

Data Warehousing und Data Mining

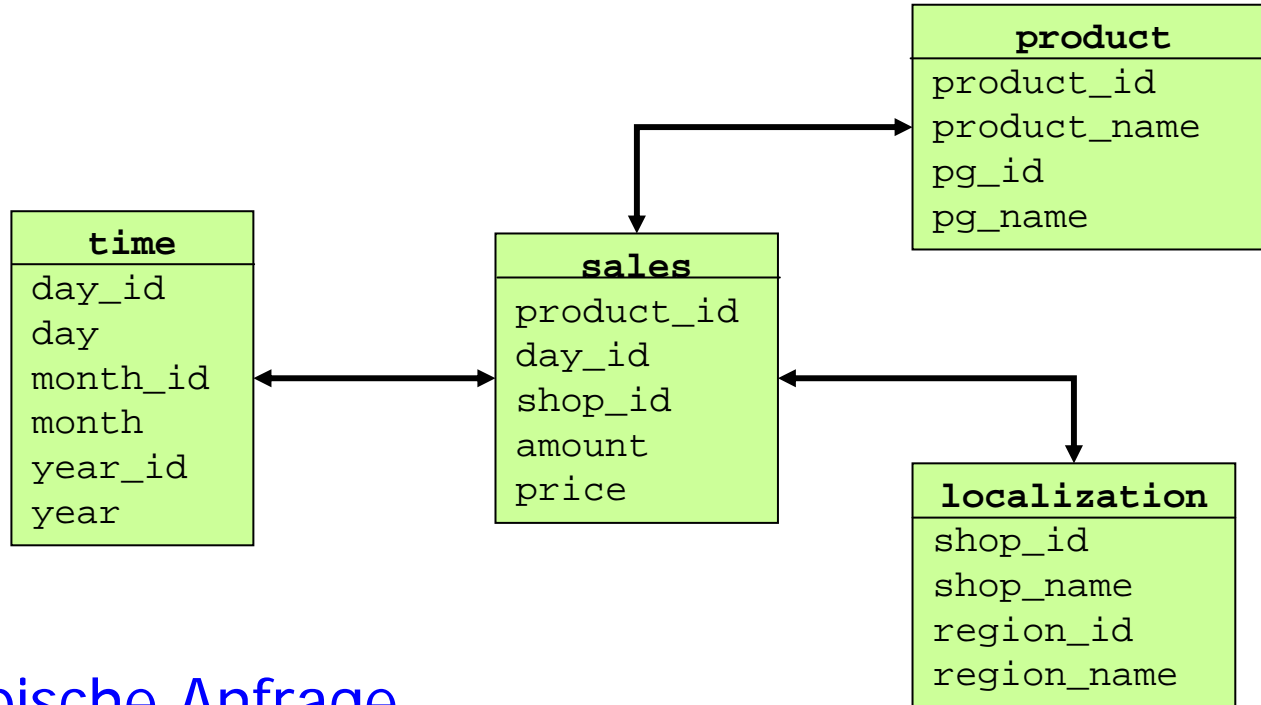
Implementierung von OLAP
Operationen

Ulf Leser

Wissensmanagement in der
Bioinformatik



Anfragen an Star-Schema



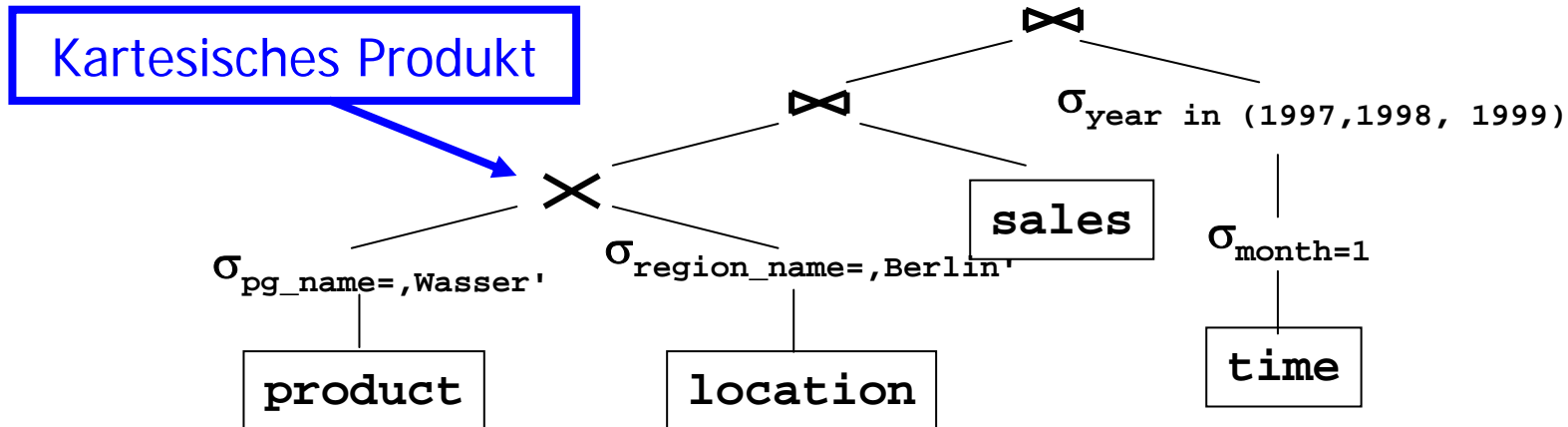
- **Typische Anfrage**

- Aggregation und Gruppierung
- Bedingungen auf den Werten der Dimensionstabellen
- Joins zwischen Dimensions- und Faktentabelle

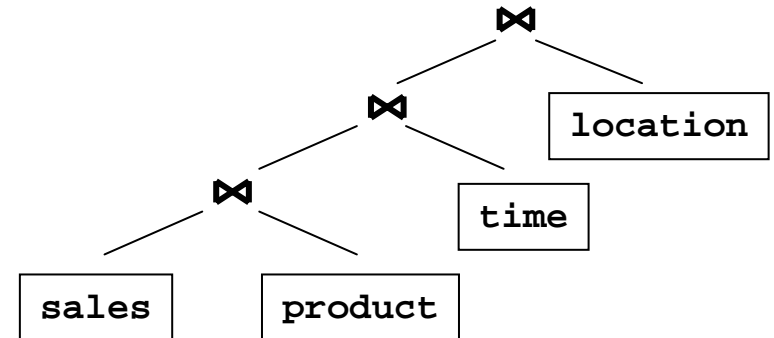
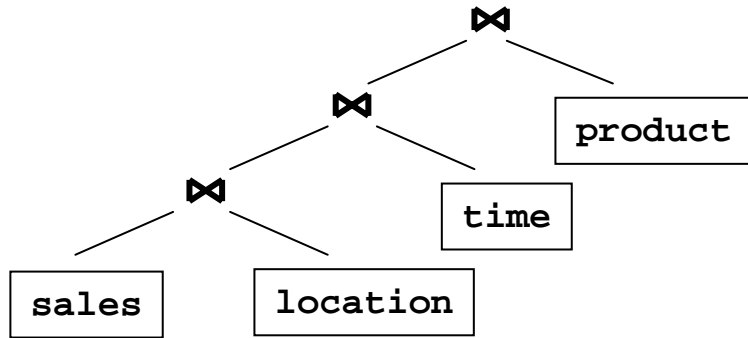
Heuristiken

```
SELECT T.year, sum(amount*price)
FROM sales S, product P, time T, localization L
WHERE P.pg_name=,'Wasser' AND
      P.product_id = S.product_id AND
      T.day_id = S.day_id AND
      T.year in (1997, 1998, 1999) AND
      T.month = ,1' AND
      L.shop_id = S.shop_id AND
      L.region_name=,'Berlin'
GROUP BY T.year
```

- Typisches Vorgehen
 - Auswahl des Planes nach Größe der Zwischenergebnisse
 - Keine Beachtung von Plänen, die **kartesisches Produkt** enthalten



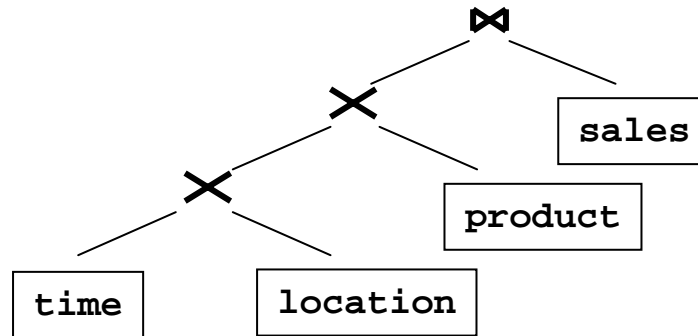
Reihenfolge zählt



	Zwischen- ergebnis
1. Join (m / 15)	6.666.666
2. Join ($ J_1 * 3 / 120$)	166.666
3. Join ($ J_2 / 50$)	3.333

	Zwischen- ergebnis
1. Join (m / 50)	2.000.000
2. Join ($ J_1 * 3 / 120$)	50.000
3. Join ($ J_2 / 15$)	3.333

Plan mit kartesischen Produkten



	Zwischenergebnis
1. time x location (3*20 * 100)	6.000
2. ... x product (P ₁ * 20)	120.000
3. ... ⋈ sales	3.333

- Es gibt mehr „Zellen“ als Verkäufe
- Nicht an jedem Tag wird jedes Produkt in jedem Shop verkauft

Star-Join in Oracle 7

- Strategie
 1. Kartesisches Produkt aller Dimensionstabellen
 2. Zugriff auf Faktentabelle über Index
 - Hohe Selektivität für Anfrage wichtig
 - Zusammengesetzter Index auf allen FKs muss vorhanden sein
 - Sonst „nur“ kleinere Zwischenergebnisse, dann teurer Scan
- Weiterer Vorteil des kartesischen Produkts
 - „Berichtsform“: Auch leere Würfelzellen sollen ins Ergebnis
 - Äquivalent zu Outer-Joins
 - Werden durch das kartesische Produkt alle gebildet
- Aber: Das ist **nicht immer gut**
 - Daten für 3 Monate, 10 Jahre, 5 Regionen, 10 Produktgruppen
 - Größe des kartesischen Produkts?
 - $3 * 20 * 10 * 5 * 100 * 10 * 20 = 60.000.000$

Star-Join in Oracle 8i – 9i

- Neue Star-Join Strategie seit Oracle 8i
- Benutzung komprimierter **Bitmapindexe**
- Phasen
 1. Berechnung aller FKs in Faktentabelle gemäß Dimensionsbedingungen einzeln für jede Dimension
 2. Anlegen/laden von bitmapped Join-Indexten auf allen FK Attributen der Faktentabelle
 3. Merge (AND) aller Bitmapindexe
 4. Direkter Zugriff auf Faktentabelle über TID
 5. Join **nur der selektierten Fakten** mit Dimensionstabellen zum Zugriff auf Dimensionswerte
- Zwischenergebnisse sind nur Bitlisten

Beispiel – Schritt 2 & 3

```
SELECT T.year, sum (amount * price)
FROM sales S, time T
WHERE S.day_id IN
```

```
(SELECT day_id FROM time
WHERE year in (1997, 1998, 1999) AND month=,1`)
```

```
AND S.shop_id IN
```

```
(SELECT shop_id FROM localization WHERE region_name=,Berlin`)
```

```
AND S.product_id IN
```

```
(SELECT product_id FROM product WHERE pg_name=,Wasser`)
```

```
AND S.day_id = T.day_id
```

```
GROUP BY T.year
```

2400 Bitarrays
Davon ~60 gewählt
Mit OR verknüpfen

1500 Bitarrays
Davon ~100 gewählt
Mit OR verknüpfen

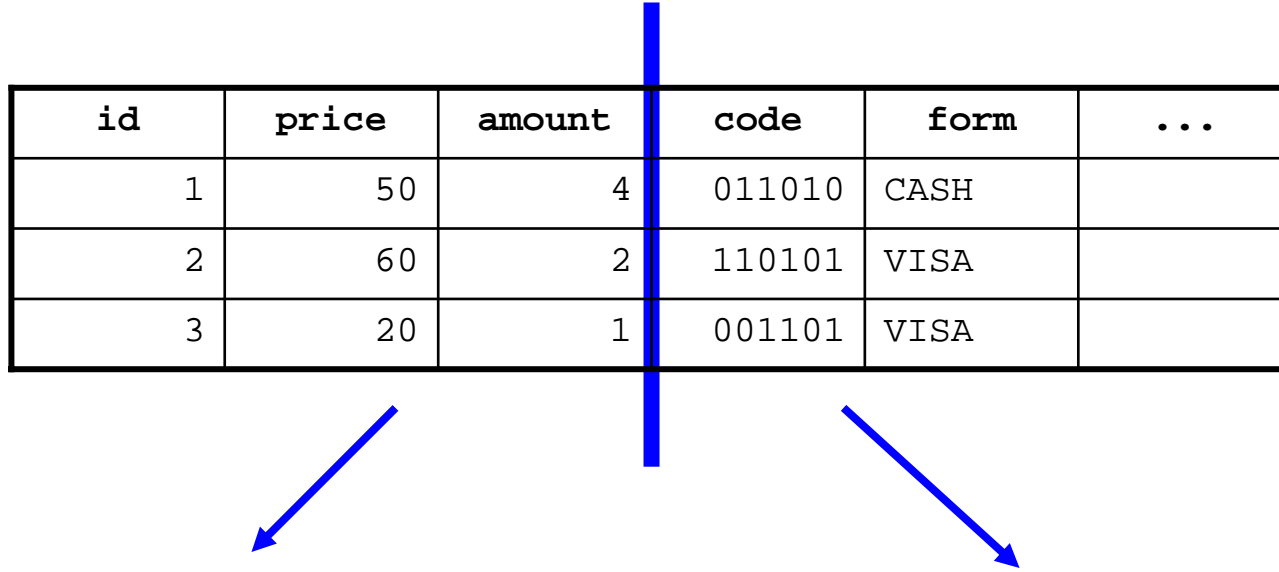
1000 Bitarrays
Davon ~20 gewählt
Mit OR verknüpfen

AND

Weitere Verfeinerung: Bloom-Filter

- Star-Join schwierig, wenn Bitarrays nicht in Hauptspeicher passen
 - Performancegewinn geht verloren
- Idee: Bloom-Filter
 - „Unschärfe“ Schritte 2-3
 - Aufbau einer maximal großen Bitmap B
 - Maximal bzgl. Hauptspeichergröße
 - Auswahl einer Dimension D
 - Die mit der besten (also geringsten) Selektivität
 - Wahl einer Hashfunktion h : TIDs nach B
 - Mehrere TIDs werden auf das selbe Bit in B abgebildet

Vertikale Partitionierung



id	price	amount	code	form	...
1	50	4	011010	CASH	
2	60	2	110101	VISA	
3	20	1	001101	VISA	

id	price	amount
1	50	4
2	60	2
3	20	1

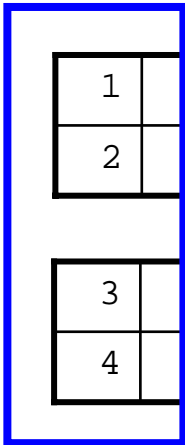
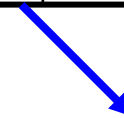
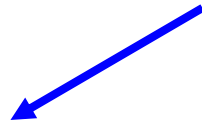
id	code	form	...
1	011010	CASH	
2	110101	VISA	
3	001101	VISA	

Bewertung

- „Zerstört“ semantische Einheiten – die Tupel
- Zusammenfassen erfordert (teure) Joins
- Geeignete Technik zur „Auslagerung“ von ...
 - Selten benutzten Attributen
 - Attributen, die häufiger als andere verändert werden
 - Und deshalb zu Reorganisation / Row Splits führen
 - BLOBs, LONGS, etc.
- Herstellung von Transparenz?
 - Definition eines Views
 - Benötigt immer zusätzliche Joins
 - I.d.R. Performanceverschlechterung

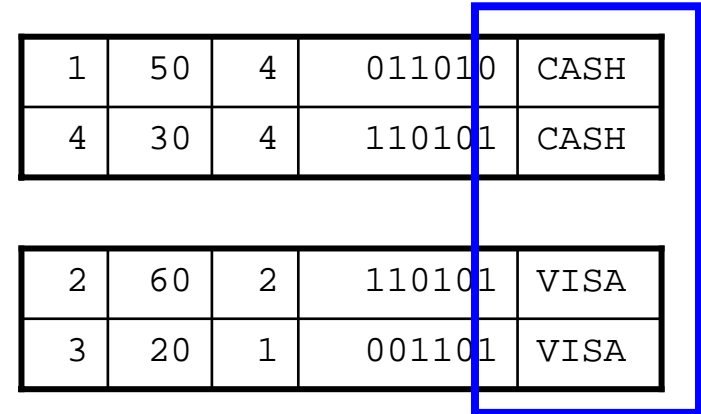
Horizontale Partitionierung

id	price	amount	code	form	...
1	50	4	011010	CASH	
2	60	2	110101	VISA	
3	20	1	001101	VISA	
4	30	4	110101	CASH	



1	50	4	011010	CASH
2	60	2	110101	VISA

3	20	1	001101	VISA
4	30	4	110101	CASH



1	50	4	011010	CASH
4	30	4	110101	CASH

2	60	2	110101	VISA
3	20	1	001101	VISA

Horizontale Partitionierung

- Wichtige Erweiterung der meisten kommerziellen RDBMS der letzten Jahre
- Hauptvorteile
 - **Datenmanagement** – Partitionen als eigenständige Datenbankobjekte
 - **Parallele Verarbeitung**
 - Scans können Partitionen auslassen (**Partition pruning**)

Prinzip

```
CREATE TABLE sales_range
  (salesman_id  NUMBER(5),
   salesman_name VARCHAR2(30),
   sales_amount NUMBER(10),
   sales_date   DATE)
PARTITION BY RANGE(sales_date)
( PARTITION t21 VALUES LESS THAN(TO_DATE('02/01/2000','DD/MM/YYYY')),
  PARTITION t22 VALUES LESS THAN(TO_DATE('03/01/2000','DD/MM/YYYY')),
  PARTITION t23 VALUES LESS THAN(TO_DATE('04/01/2000','DD/MM/YYYY')),
  PARTITION t24 VALUES LESS THAN(TO_DATE('05/01/2000','DD/MM/YYYY'))
);
```

- Partitionen sind eigene Datenbankobjekte (Tabellen)
 - Eigene DDL Kommandos
 - Müssen explizit angelegt werden
- Einmal angelegt, ist ihre **Existenz für Benutzer transparent**
 - Wie bei Indexen
- Die letzte Partition kann als Grenze **MAXVALUE** haben
 - „Specifying a value other than MAXVALUE for the highest partition bound imposes an implicit integrity constraint on the table. „

Partition Pruning

- Bei Anfragen mit **Bedingung auf Partitionierungsattribut**
- Punktanfragen
 - Direkte Auswahl relevanter Partition
 - Partition ähnlich einer Ebene in B*-Baum
 - **Keine Performanceverbesserung**
- Bereichsanfragen
 - Direkte Auswahl relevanter Partitionen
 - Auch Daten liegen partitioniert vor (nicht nur TIDs)
 - Ähnlichkeit zu Index-Organized Tables
 - **Erhebliche Performanceverbesserung**
- Unterstützte Prädikate
 - Pruning mit Listen/Bereichspartitionierung nur bei =, <, >, IN
 - Pruning mit Hashpartitionierung nur bei =, IN

Parallelität

- Arten von Parallelität

- Interquery

- Verteilen von Anfragen auf Prozessoren / Knoten
 - Keine Beschleunigung einzelner Queries
 - Behandlung auf Session-Ebene (Dedicated Server / MTS)

- Intraquery

- Aufbrechen der Query in parallel ausgeführte Teilanfragen
 - Query wird leicht verändert auf unterschiedlichen Datenbereichen ausgeführt
 - Anfrage läuft parallel auf unterschiedlichen Partitionen

Inhalt dieser Vorlesung

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

SQL und OLAP

- Übersetzung MDDM in Star- oder Snowflake Schema
- Triviale Operationen
 - Auswahl (slice, dice): Joins und Selects
 - Verfeinerung (drill-down): Joins und Selects
- Einfache Operation
 - Aggregation um eine Stufe: Group-by
- Schwieriger
 - Hierarchische & multidimensionale Aggregationen
 - Analytische Funktionen (attributlokal, sequenzbasiert)
- Sind beide in SQL-92 möglich, aber nur **kompliziert auszudrücken und ineffizient** in der Bearbeitung

Hierarchische Aggregation –2-

Alle Verkäufe der Produktgruppe „Wein“ nach Tagen,
Monaten und Jahren

Benötigt UNION und eine Query pro Klassifikationsstufe

```
SELECT    T.day_id, sum(amount*price)
FROM      sales S, product P
WHERE     P.pg_name=„Wein“ AND
```

```
GROUP BY T.day_id
SELECT    T.month_id, sum(amount*price)
FROM      sales S, product P, time T
WHERE     P.pg_name=„Wein“ AND
```

```
GROUP BY T.month_id
SELECT    T.year_id, sum(amount*price)
FROM      sales S, product P, time T
WHERE     P.pg_name=„Wein“ AND
          P.product_id = S.product_id AND
          T.day_id = S.day_id
GROUP BY T.year_id
```

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()`

Cube Operator

- d Dimensionen, jeweils eine Klassifikationsstufe
 - Jede Dimension kann in Gruppierung enthalten sein oder nicht
 - 2^d Gruppierungsmöglichkeiten
- Herkömmliches SQL
 - Viel Schreibarbeit
 - 2^d Scans der Faktentabelle
- CUBE Operator
 - Berechnung der Summen von sämtlichen Kombinationen der Argumente (Klassifikationsstufen)
 - Summen werden durch „ALL“ repräsentiert
 - Keine Beachtung von Hierarchien
 - Durch Schachtelung mit ROLLUP erreichbar

SQL Analytical Functions

- Erweiterung in SQL3 zur flexibleren Berechnung von **Aggregaten und Rankings**
 - Ausgabe der Summe Verkäufe eines Tages zusammen mit der Summe Verkäufe des Monats **in einer Zeile**
 - Rang der Summe Verkäufe eines Monats im Jahr über alle Jahre
 - Vergleich der Summe Verkäufe eines Monats zum **gleitenden Dreimonatsmittel**
- OLAP Funktionen
 - Erscheinen in der SELECT Klausel
 - Berechnen für **jedes Tupel des Stroms der Groupby-Query** einen Wert
 - Diese Werte können von neuen, in der SELECT Klausel angegebenen Partitionierungen und Sortierungen abhängen
 - Damit kann man unabhängige Gruppierungen in einer Zeile des Ergebnisses verwenden
 - CUBE etc. erzeugen immer neue Tupel

OVER() Klausel

- Ausgabe der Summe Verkäufe eines Tages im Verhältnis zu den Gesamtverkäufen

```
SELECT      S.day_id, sum(amount) AS day_sum, day_sum/T.all_sum
FROM        sales S,
            (SELECT sum(amount) AS all_sum
             FROM sales S) T
GROUP BY    S.day_id;
```

- Kompliziert zu schreiben und potentiell **ineffizient**
 - Warum müssen wir zweimal über **sales** iterieren?
- Besser: **over() Klausel**

```
SELECT      S.day_id, sum(amount) AS day_sum,
            day_sum/ (sum(amount) OVER())
FROM        sales S
GROUP BY    S.day_id;
```

- OVER() ohne Parameter iteriert über alle Tupel

OVER() mit lokaler Partitionierung

- OVER() gestattet die Angabe **attributlokaler Partitionen**
- Ausgabe der Summe Verkäufe eines Tages im Verhältnis zu Verkäufen des Jahres

```
SELECT      S.day_id, sum(amount) AS day_sum,  
            sum(amount) OVER(PARTITION BY YEAR(S.day_id)) AS year_sum,  
            day_sum/year_sum AS ratio  
FROM        sales S,  
GROUP BY    S.day_id;
```

SQL Funktion
YEAR()

day_id	day_sum
1.1.1999	500
2.1.1999	500
3.1.1999	1.000
1.2.1999	1.500
2.2.1999	1.500
1.1.2000	600
5.5.2000	400
1.10.2001	4.000

Sommers

day_id	day_sum	year_sum	Ratio
1.1.1999	500	5.000	0.10
2.1.1999	500	5.000	0.10
3.1.1999	1.000	5.000	0.20
1.2.1999	1.500	5.000	0.30
2.2.1999	1.500	5.000	0.30
1.1.2000	600	1.000	0.60
5.5.2000	400	1.000	0.40
1.10.2001	4.000	4.000	1.00

Inhalt dieser Vorlesung

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

GROUP BY

```
SELECT    region, product, SUM(amount)
FROM      sales, ...
...
GROUP BY  region, product
```

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



Hashbasierte Implementierung

- Implementierung über Hashing
 - Lese Tupelstrom und verwende Wert von **G als Index in Hashtabelle**
 - Benötigt eine geeignete Hashfunktion – am Anfang ist unklar, wie viele verschiedene Werte von G es gibt
 - Erzeugt einen Bucket pro Wert von G
 - Iteriere über alle Buckets und wende Aggregatfunktion an
- Problem: Evt. müssen **alle Buckets im Hauptspeicher** gehalten werden
 - Bei distributiven (und algebraischen) Funktionen muss nur ein (wenige) Wert pro Bucket gehalten werden
 - Bei holistischen Funktionen müssen wir **effektiv alle Tupel im Speicher** halten
 - Denn man weiß nicht, wann das letzte Tupel einer Partition kommt

Sortierung

- Implementierung durch Sortierung
 - **Sortiere den Tupelstrom** nach den Werten von G
 - Im Hauptspeicher oder mit externer Sortierung
 - Lies den sortieren Tupelstrom
 - Tupeln puffern oder sofort aggregieren
 - Wenn sich Werte von G ändern beginnt eine neue Partition
 - Ggf: Berechne Aggregatfunktion
 - **Pipelining**: Wert kann sofort weitergereicht werden
- Vorteil: Benötigt im schlimmsten nur so viel Platz wie die größte Partition
 - Bei distributiven Funktionen sogar **nur einen Wert**
- Nachteil: Erfordert Sortierung

Fazit

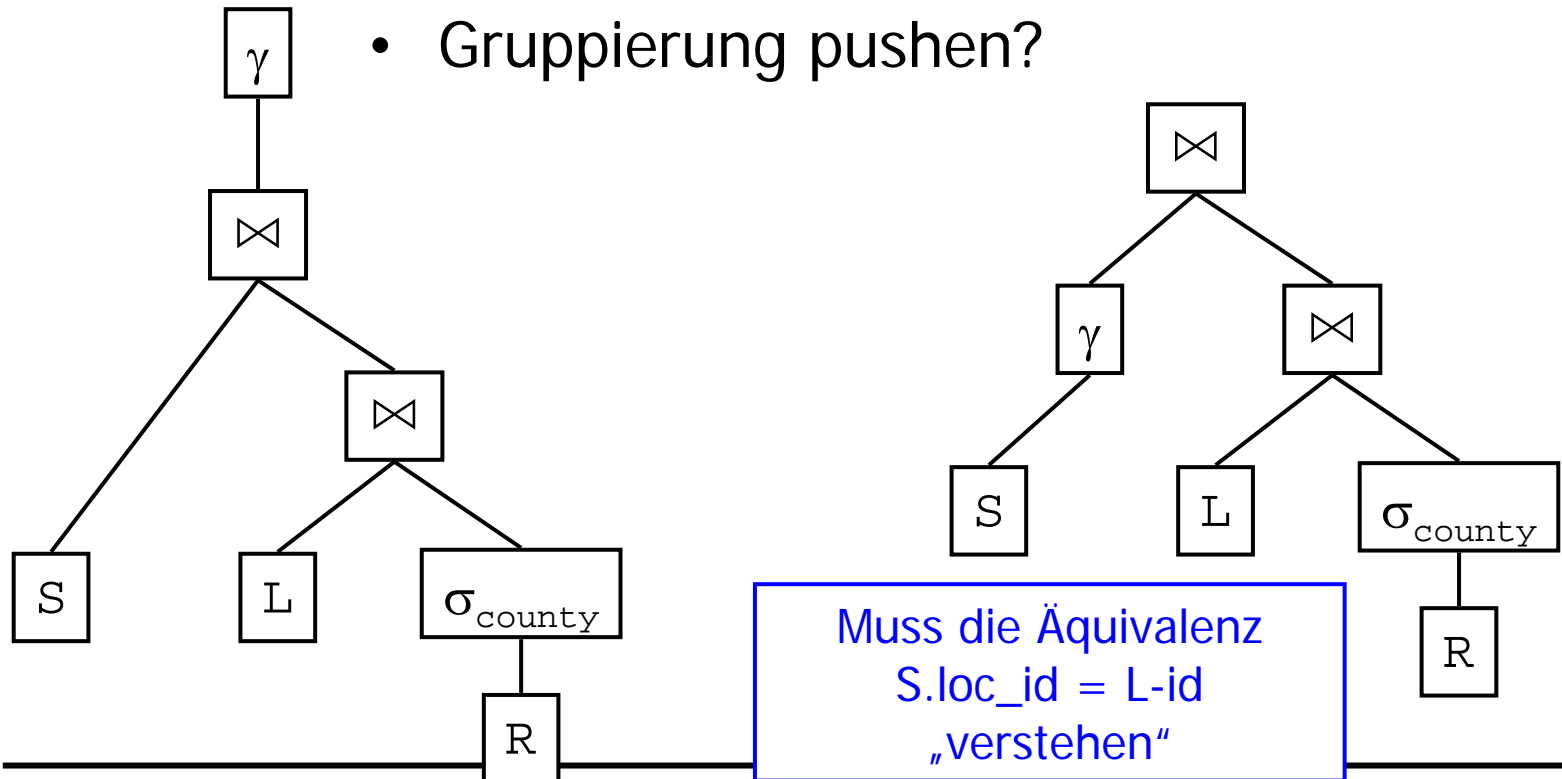
- Wenn alle Buckets in den Hauptspeicher passen – Hashen
- Wenn nicht – sortieren
- Aber: Wie kann man das vorher **abschätzen**?
 - Wichtigstes Indiz ist die Aggregatfunktion selber
 - Außerdem die Anzahl erwarteter Partitionen
 - Naive Abschätzung
 - COUNT DISTINCT einzeln auf alle Attribute in G machen
 - Anzahl Werte von G = Produkt der Anzahl verschiedener Werte über alle Attribute
 - **Worst Case**: Nimmt an, dass alle Kombinationen vorkommen
 - Ignoriert Abhängigkeiten zwischen Werten
 - Abzählen
 - COUNT DISTINCT auf G
 - Dann kann man (fast) gleich die Gruppierung ausrechnen
 - Besser: **Sampling**

Algebraische Optimierung

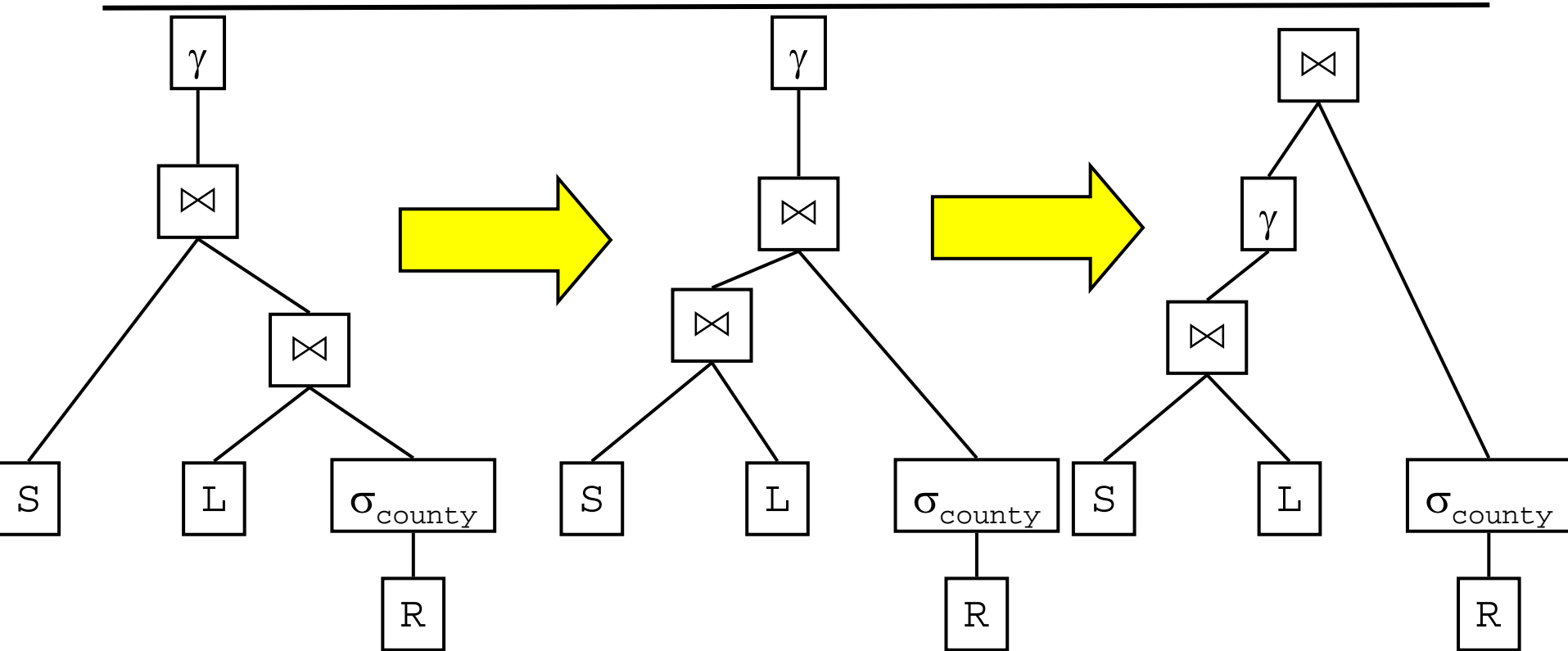
- Bisher nehmen wir an, dass ein GROUP BY als vorletzte Operation ausgeführt wird
 - Vor Sortierung (und Auswertung HAVING)
- Aber: **Gruppierung reduziert die Anzahl Tupel**
- Daher ist es manchmal sinnvoll, Gruppierung zu pushen
- Beispiel (Snowflake-Schema)
 - ```
SELECT S.product_id, L.id, SUM(amount)
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
```



# Alternative

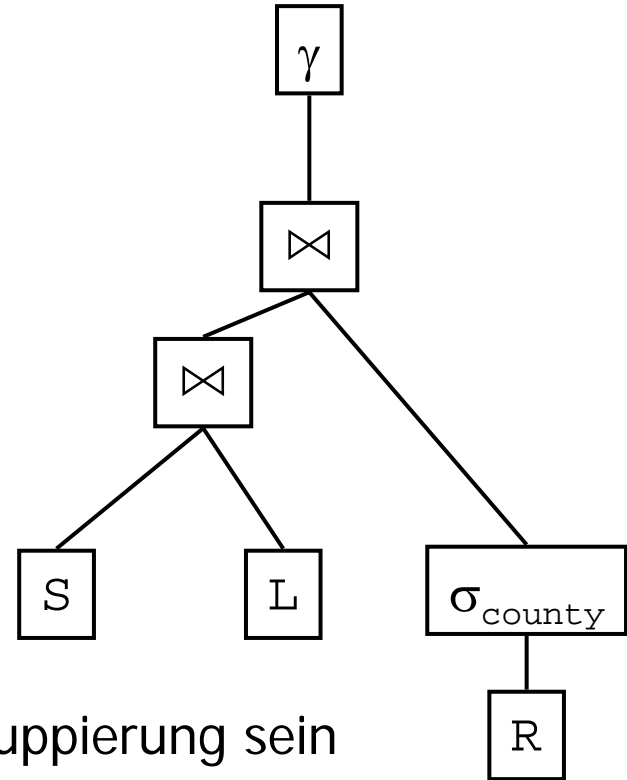


- Bedingung: **L.region\_id** muss im Ergebnis der Gruppierung sein
  - Z.B Umschreiben in `SELECT s.product_id, L.id, max(l.region_id)`
- Pushen oder nicht? **Kostenbasierte Entscheidung**
  - Der Join  $S \bowtie L$  ist größer im dritten Plan, da der Filter auf ‚BRD‘ erst später erfolgt
  - Dafür gibt es weniger Tupel im ersten Zwischenergebnis

# Zweites Beispiel

```
SELECT S.product_id, L.typ, 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.typ
```

- Annahme: Tabelle `location` hat ein **Attribut** `typ`
- Gruppierung pushen?
  - Geht nicht: Fremdschlüssel (`R.id`) kann nicht mehr im Ergebnis der Gruppierung sein
  - Nachträgliche Filterung auf `'BRD'` gelingt nicht
- Pushen nur möglich, wenn
  - **Alle Joinattribute** sind in der Gruppierung
  - Kein Attribute andere Tabellen sind in der Gruppierung



# Präaggregation

---

- Manchmal kann man Aggregatfunktionen nicht vollständig pushen, aber **Präaggregationen** einführen
  - Mehr Operationen
  - Aber die reduzieren die Größe der Zwischenergebnisse

- Beispiel

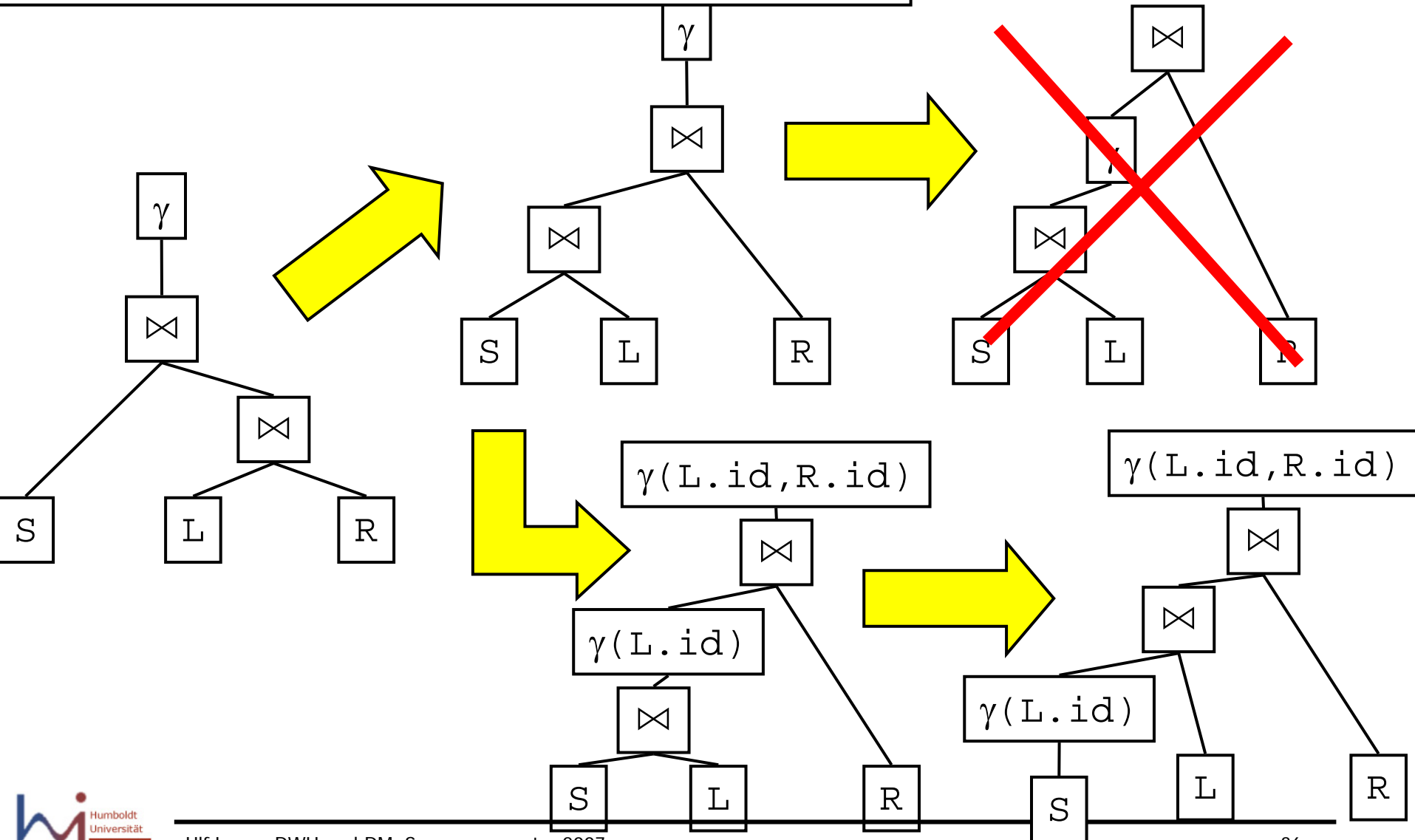
```
- SELECT L.id, R.id,
 SUM(amount), COUNT(amount)
FROM sales S, location L, region R
WHERE S.loc_id = L.id AND
 L.region_id = R.id
GROUP BY L.id, R.id
```

# Präaggregation

```

SELECT L.id, R.id, 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 L.id, R.id

```



# Stimmt das?

---

- Identisch?

- ```
SELECT  L.id, R.id,
        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 L.id, R.id
```
- ```
SELECT J.id, R.id
 SUM(X), COUNT(Y)
FROM region R,
 (SELECT L.id as lid, MAX(R.id) as rid
 SUM(amount) as X, COUNT(amount) as Y
 WHERE S.loc_id = L.id
 GROUP BY L.id) J
WHERE J.rid = R.id
GROUP BY J.lid, R.id
```

# Aggregatfunktionen anpassen

---

- Identisch?

- ```
SELECT  L.id, R.id,
        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 L.id, R.id
```
- ```
SELECT J.id, R.id
 SUM(X), SUM(Y)
FROM region R,
 (SELECT L.id as lid, MAX(R.id) as rid
 SUM(amount) as X, COUNT(amount) as Y
 WHERE S.loc_id = L.id
 GROUP BY L.id) J
WHERE J.rid = R.id
GROUP BY J.lid, R.id
```

# Inhalt dieser Vorlesung

---

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

# Berechnung analytischer Funktionen

---

- Analytische Funktionen enthalten attributlokale Partitionierungen und Sortierungen
- Diese betreffen den gesamten Tupelstrom
- I.d.R. ist daher pro OVER() Klausel eine Neusortierung des gesamten Tupelstroms notwendig
  - Teuer (eventuell Ausnutzung gleicher Sortierpräfixe)
  - SELECT Klausel wird „sequentiell“ ausgewertet

```
SELECT S.day_id, s.shop_id, sum(amount) AS ds_sum,
 sum(amount) OVER(PARTITION BY YEAR(S.day_id)) AS year_sum,
 ds_sum/year_sum AS ratio1,
 sum(amount) OVER(PARTITION BY S.shop_id) AS shop_sum,
 ds_sum/shop_sum AS ratio2
FROM sales S,
GROUP BY S.day_id, S.shop_id;
```

# Inhalt dieser Vorlesung

---

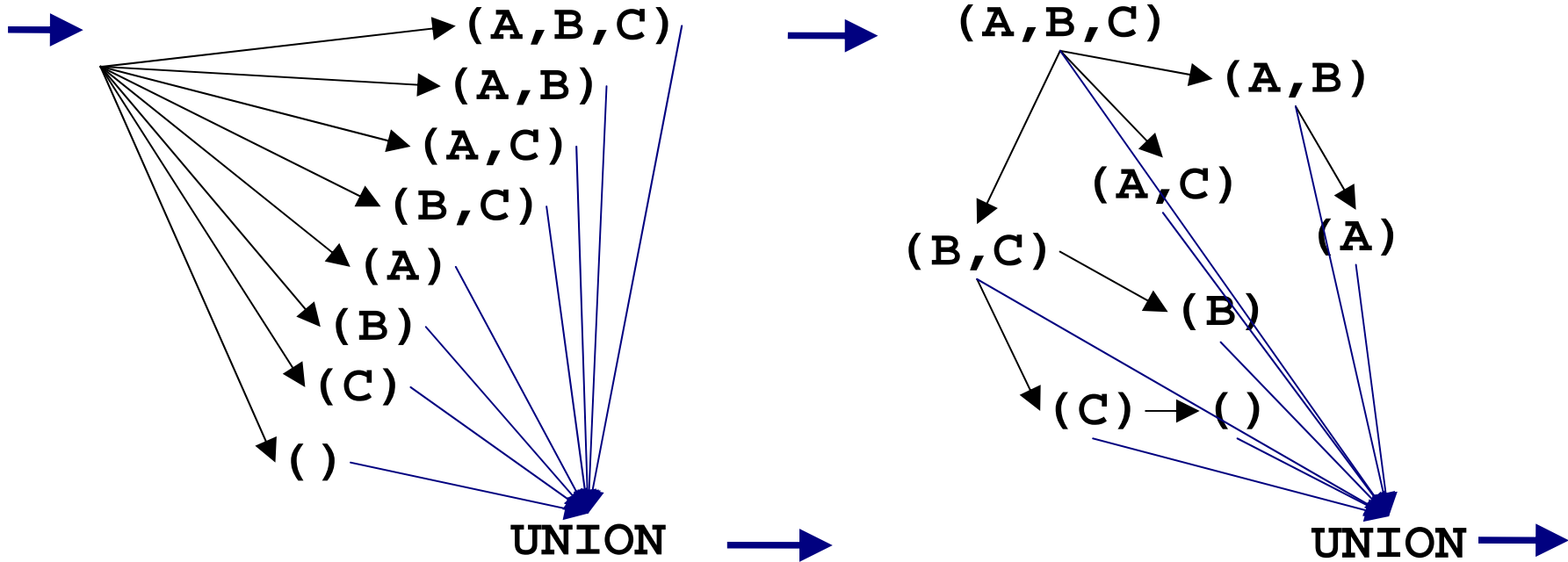
- Wiederholung: OLAP Operationen
- Implementierung der Gruppierung
- Implementierung analytischer Funktionen
- **Implementierung von Cube & Iceberg-Cube**
  - Problem und Potential
  - PipeSort Algorithmus
  - Iceberg Cubes

# Problem und Potential

---

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

# Realisierung CUBE



Naiver Ansatz

Ausnutzen von  
(direkter) Ableitbarkeit

# Szenario

---

- Definition  
*Eine Gruppierung  $G$  ist aus einer Gruppierung  $H$  ableitbar, geschrieben  $H \rightarrow G$ , wenn  $G \subseteq H$*
- Beispiel: Alles ist aus (ABC) ableitbar
- Ableitbarkeit will man ausnutzen, aber: Da es exponentiell viele Gruppierungen gibt, gehen die auf keinen Fall alle in den Hauptspeicher
- Vorgehen
  - Gruppierung mit höchster Auflösung wurde berechnet und liegt auf der Platte
  - In welcher Reihenfolge berechnet man am besten die weiteren Gruppierungen?
  - Abgeschlossene und laufende Gruppierungen konkurrieren dabei um begrenzt verfügbaren Hauptspeicher
    - Wann kann man Gruppierungen verdrängen?

# Mögliche Tricks

---

## 1. Smallest-parent

- Berechne Gruppierung aus dem **Elternteil mit den wenigsten Tupeln**
- Beispiel: (A) kann aus (AB), (AC) oder (ABC) berechnet werden
  - (ABC) hat sicher am meisten Tupel, aber  $|AB| \geq |AC|$
- 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 sortieren)
- Gewinn: Weniger IO

## 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)
- Gewinn: Weniger Scans, weniger IO

# Mögliche Tricks 2

---

## 4. 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:  $(ABC) \rightarrow (AB) \rightarrow (A) \rightarrow ()$ , aber nicht  $(ABC) \rightarrow (BC)$
- Gewinn: Keine zusätzlichen Sortierungen (und Gruppierung mit Sortierung war speicherplatzeffizienter als Hashen)

## 5. Share-partitions

- Wenn eine hashbasierte Gruppierung aus Speichermangel in Partitionen zerlegt 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

# Inhalt dieser Vorlesung

---

- Wiederholung: OLAP Operationen
- Implementierung der Gruppierung
- Implementierung analytischer Funktionen
- Implementierung von Cube & Iceberg-Cube
  - Problem und Potential
  - PipeSort Algorithmus
  - Iceberg Cubes

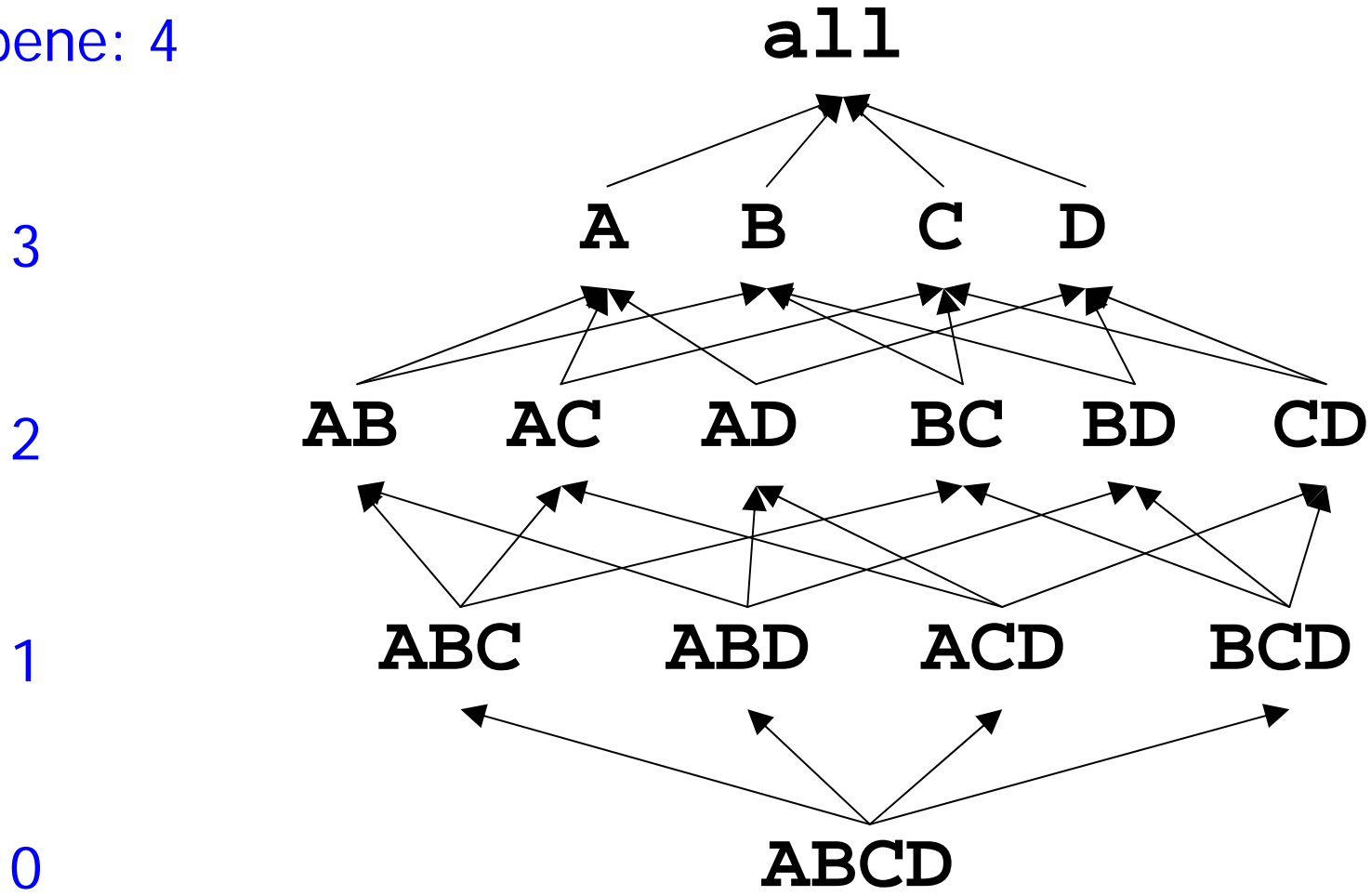
# PipeSort [AAD+96]

---

- Die verschiedenen Tricks 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
  - ...
- Algorithmen müssen sich auf einige der Tricks konzentrieren
- Beispiel: **PipeSort**
  - Eingabe ist das **Aggregationsgitter** der Gruppierung
  - Berechnet Reihenfolge und Sortierung der Gruppierungen
  - Benutzt **Share-Sorts und Smallest-Parent**
  - Benötigt Schätzungen über Kardinalität der Gruppierungen
    - Für Smallest-Parent

# Aggregationsgitter

Ebene: 4



# Grundidee

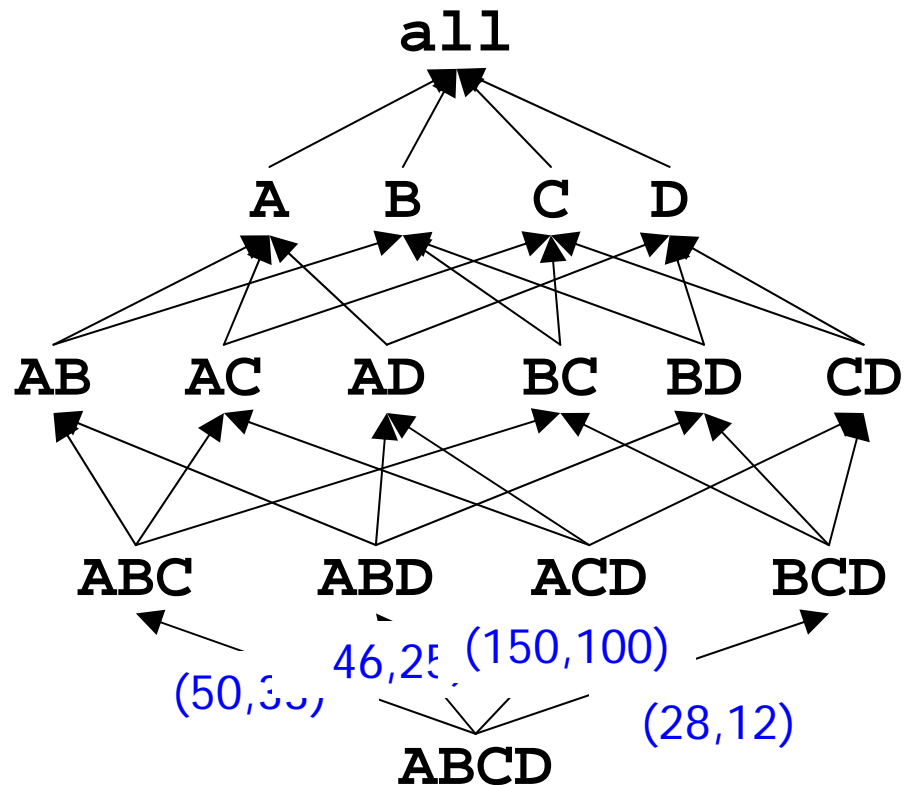
---

- Bei den Kosten der Berechnung gehen mit ein
  - **Kosten für Sortierung** – wenn notwendig
  - **Kosten für Gruppierung** – abhängig von der Größe des Eltern
- Kosten werden an Kanten im Gitter geschrieben
- PipeSort geht **ebenenweise** durch das Gitter
  - Start bei der Gruppierung mit der höchsten Auflösung
  - Jede Gruppierung wird von einem unmittelbaren Vorfahr abgeleitet
  - Das ist eine Heuristik zur Komplexitätsreduktion
- Für jeden Ebenenwechsel wird eine Lösung durch **biartites Matching** berechnet
  - Global ist das i.d.R. nicht optimal
  - PipeSort ist Greedy

# 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

Hier ist noch keine Sortierung festgelegt!



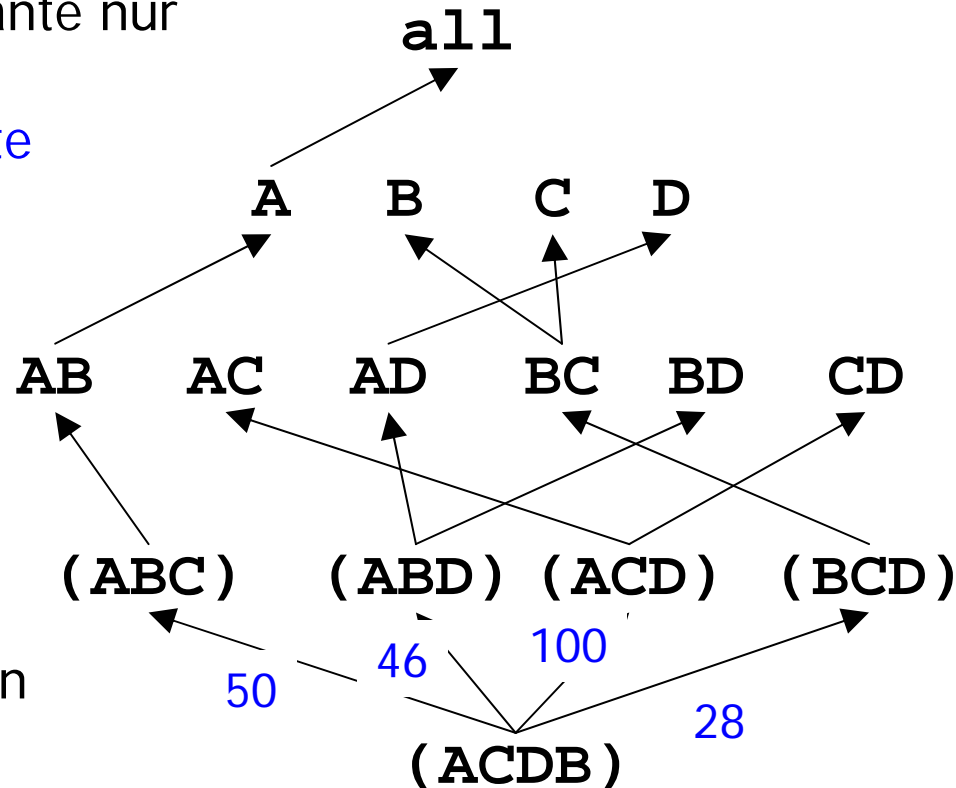
# Ziel

- Wir wollen den Subgraphen finden, für den gilt
  - Jeder Knoten hat nur eine eingehende Kante
  - Jeder Knoten hat eine festgelegte Sortierung
  - Damit ergibt sich für jede Kante nur noch ein Gewicht
  - Der **Subgraph soll die kleinste Kantensumme** haben

- Natürlich NP-hard

- Bemerkung

- Da nur direkte Ableitungen ausgenutzt werden, kann **nur ein Nachfahre die ganze Sortierung „erben“**
- Partielle Sortierungen können auch berücksichtigt werden

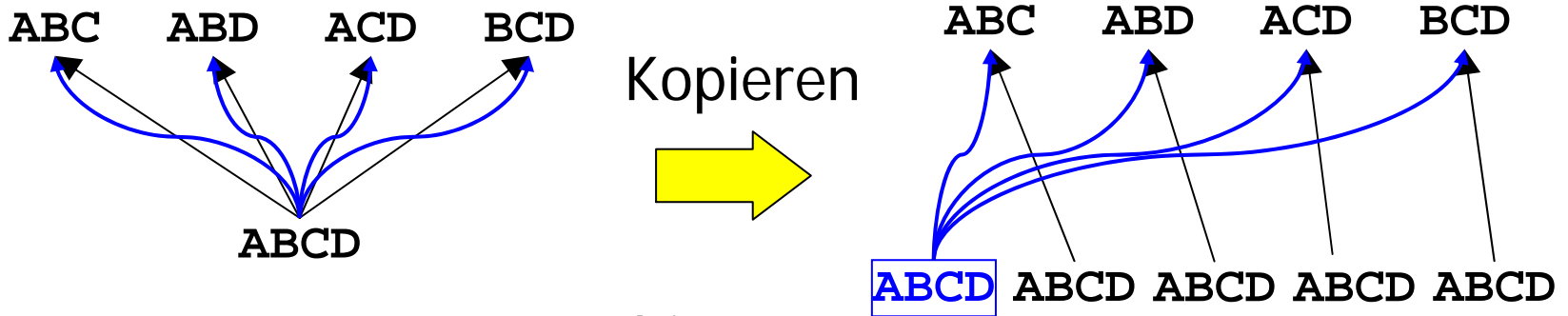


# Ebene für Ebene

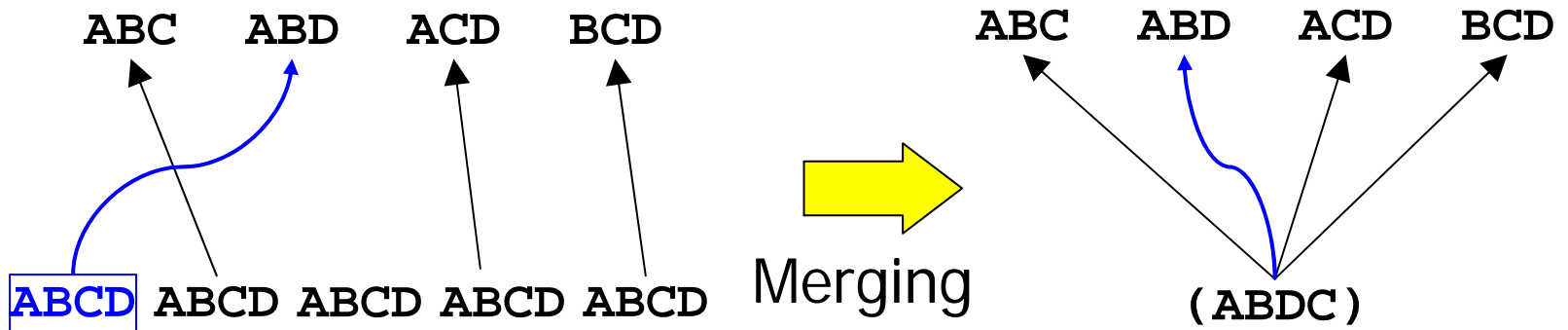
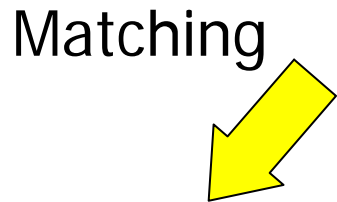
---

- Betrachte jeden Ebenenübergang im gewichteten Gitter
- Wir suchen eine optimale Teillösung, die
  - Für jeden Kindknoten genau **eine eingehende Kante** hat
  - Die **Sortierung aller Elternknoten** festlegt
- Reduktion auf **Bipartites Matching**
  - „Kopiere“ Elternknoten
    - $\forall$  Elternknoten mit  $n$  ableitbaren Gruppen, jeweils mit A/S Kanten
    - Lege 1 Kopie an, die alle A-Kanten behält
    - Lege  $n$  Kopien an, die jeweils eine S-Kante behalten
  - Dies ist ein bipartiter Graph
  - Suche ein **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 aller Eltern durch ausgehende A-Kante fest

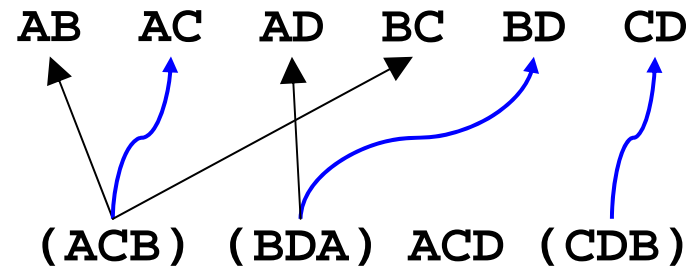
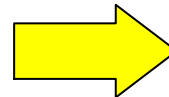
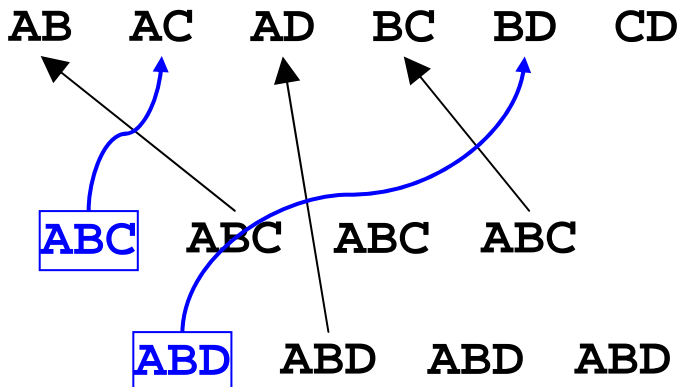
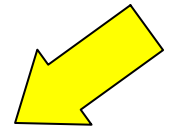
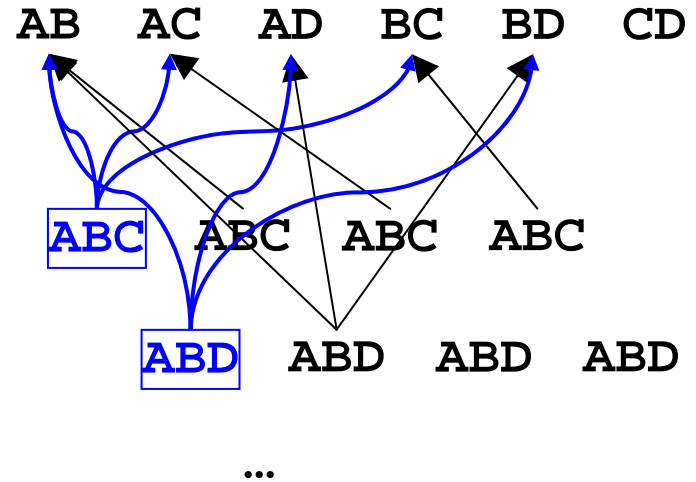
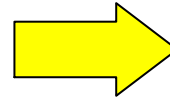
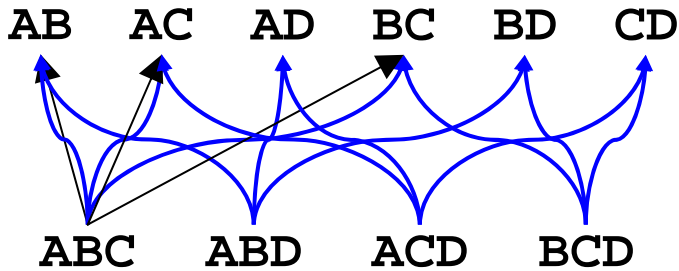
# Beispiel – Level 1



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

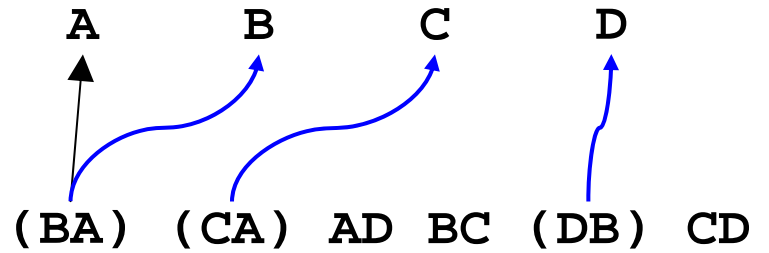
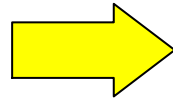
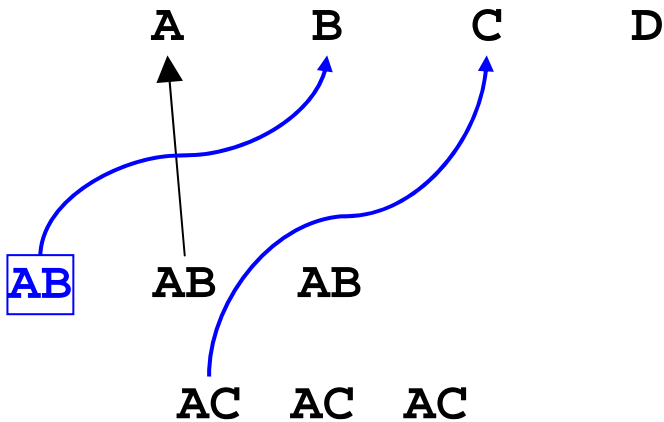
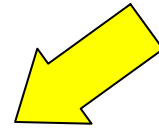
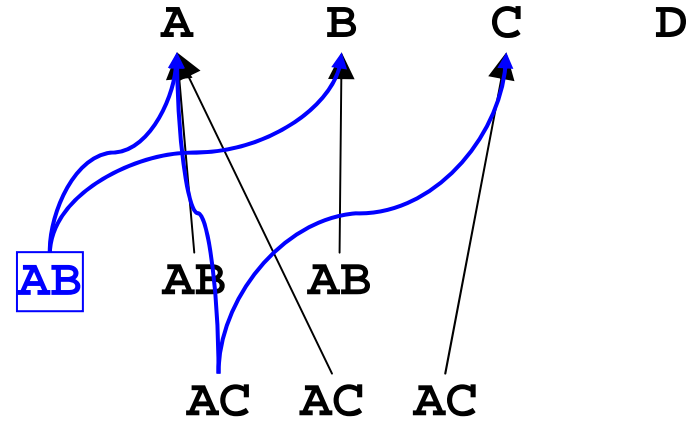
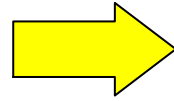
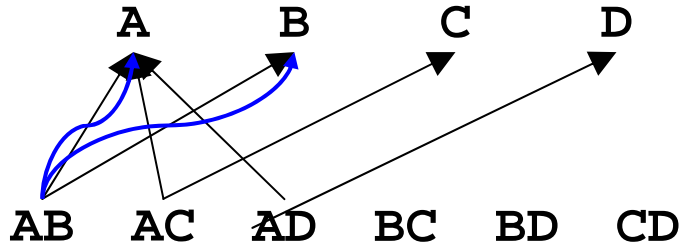


# Beispiel – Level 2

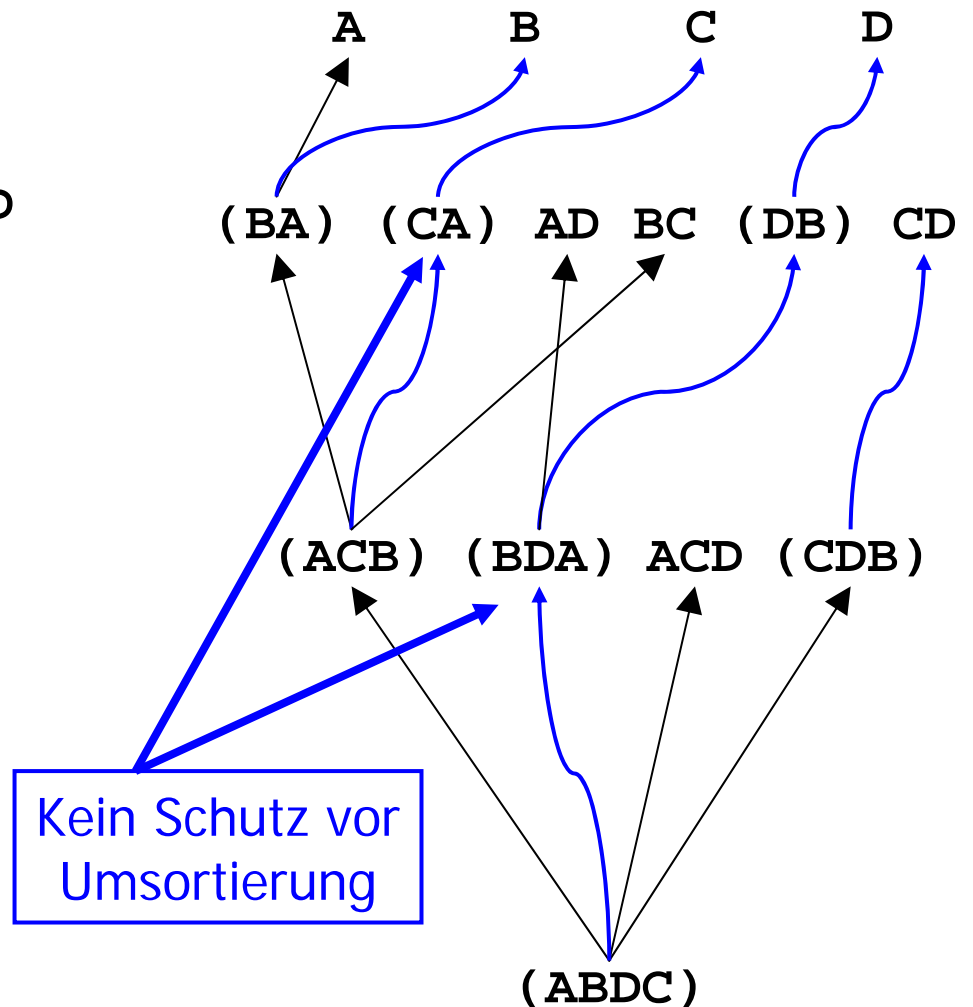
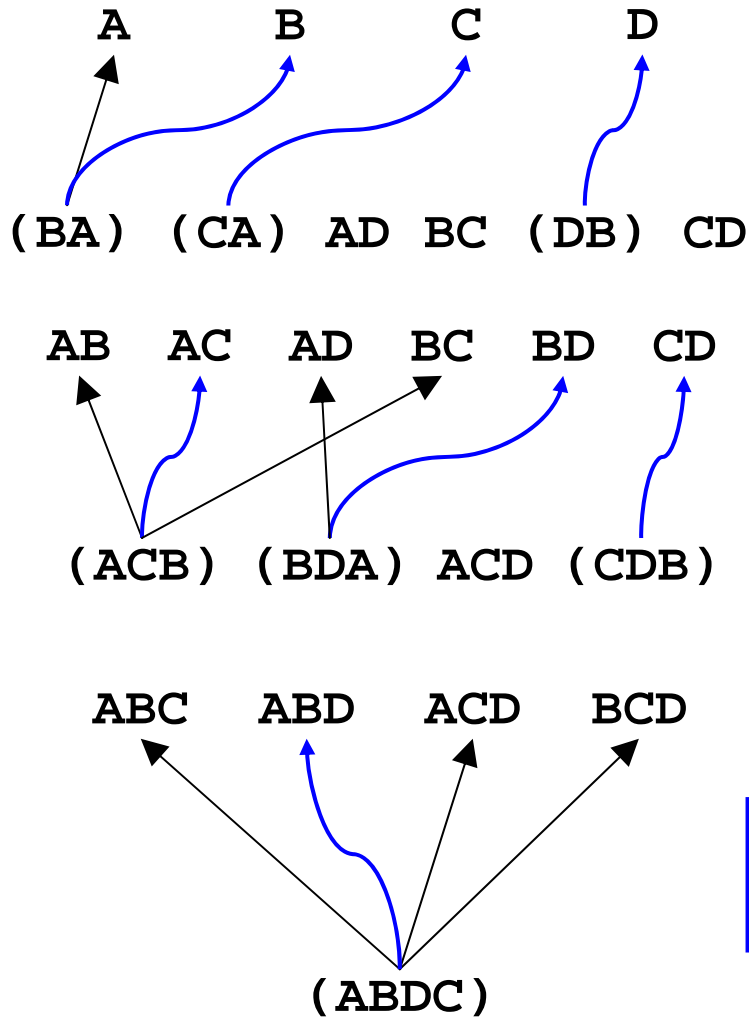


Von kleinen Eltern kann die Ableitung auch mit Umsortierung billiger sein als von großen Eltern

# Level 3

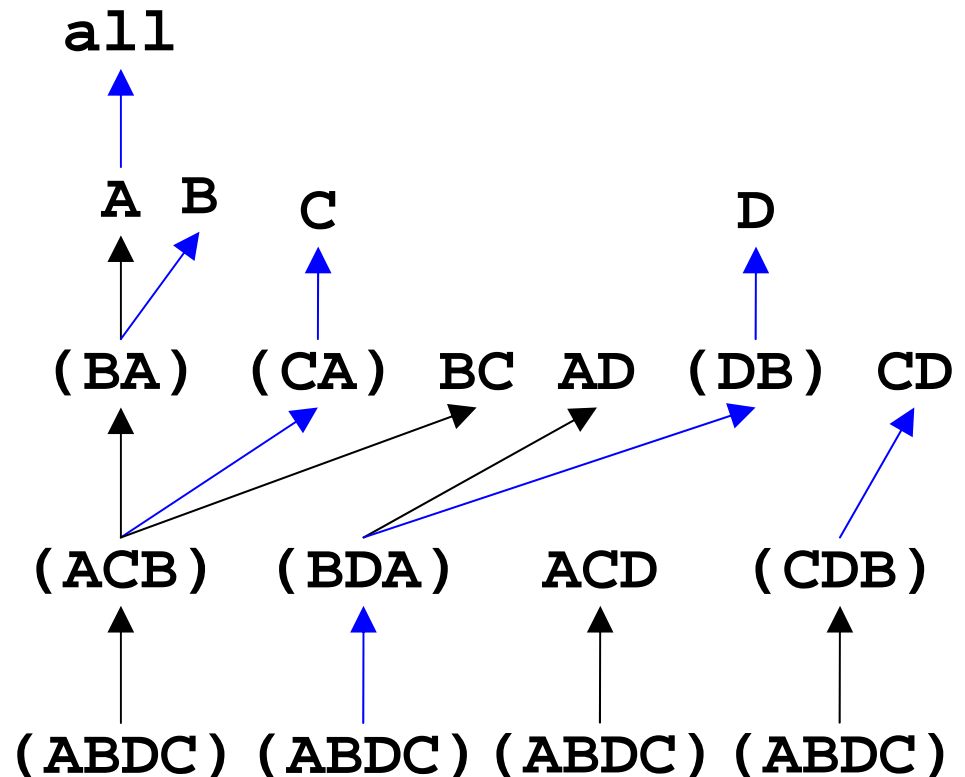
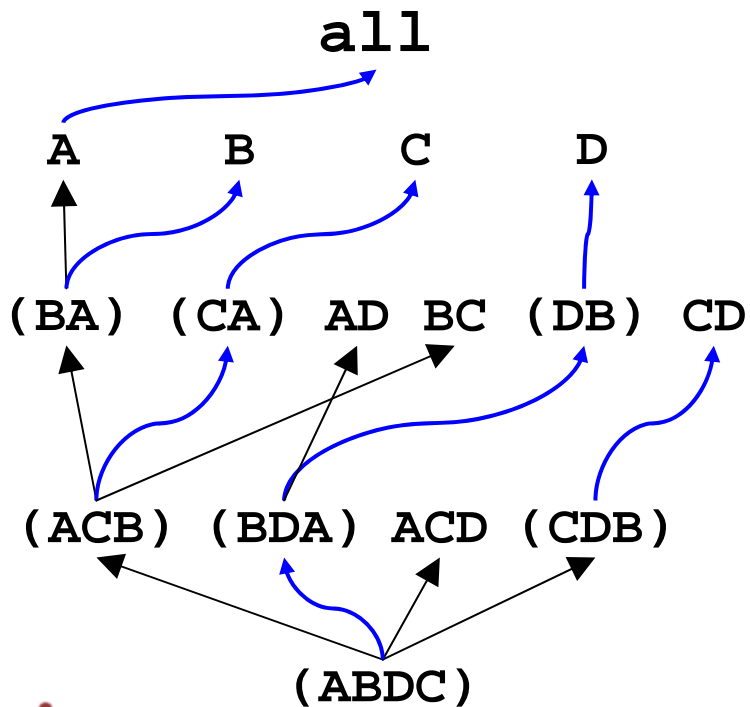


# Ergebnis



# Berechnung

- Ebenenübergänge werden von unten nach oben (bottom-up) festgelegt
- Dann werden Teilbäume in jeweils einem Lauf berechnet



# Bewertung

---

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

# Iceberg Cubes

---

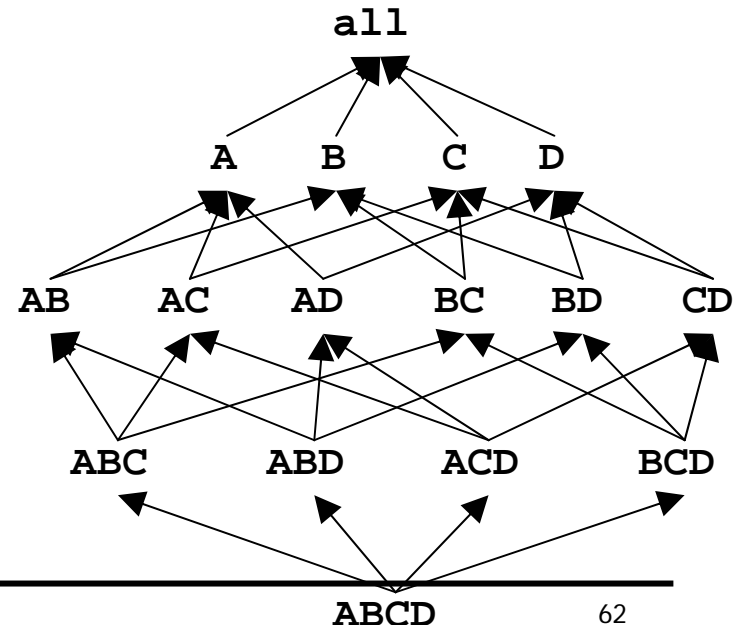
- Immer noch: Es gibt exponentiell viele Gruppierungen in einem CUBE Operator
  - 10 Dimensionen a 10 Ausprägungen = 1024 Gruppierungen mit zwischen 1 und  $10^{10}$  Partitionen
  - „High dimensionality“ – die allermeisten Kombinationen müssen leer sein
    - So viel kann man gar nicht verkaufen
- Iceberg Cubes
  - Oftmals interessieren nur Aggregate oberhalb einer gewissen Grenze
  - ```
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

- Möglichkeit 1: CUBE normal berechnen, „leere“ Partitionen dann filtern
 - Ineffizient
- Beobachtung
 - (Gilt nicht für alle Aggregatfunktionen!)
 - Wenn für eine Partition P einer Gruppierung G und Aggregatfunktion f gilt, dass $f(P) < t$, dann muss für alle Partitionen P' jeder Gruppierung G' mit $G \subseteq G'$ und $P \subseteq P'$ gelten, dass $f(P') < t$
 - A-Priori Eigenschaft
 - Durch die Hinzunahme weiterer Attribute in die Partitionierung werden die Partitionen höchstens kleiner
 - Das kann man zum **Prunen** ausnutzen

Beispiel

- Gruppierungsattribute (year, product)
- Partitionen (1990, Wedding), (1991, Wedding), ... (2007, Mitte)
- Wenn die Verkäufe 1991 im Wedding 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) muss man also Partitionen, die (1991, Wedding) enthalten, nicht berechnen
- Problem: Das ist „Falschrump“: von grob zu fein



Construction of Iceberg Cubes [BR99]

- Baut die Gruppierungen von ALL nach (ABCD)
- Iteriere über alle Dimensionen d
 - Partitioniere nach d
 - Iteriere über alle Partitionen P
 - Wenn $\text{COUNT}(P) < t$: Wegwerfen
 - Sonst; Gib $(P, \text{COUNT}(P))$ aus; Steige rekursiv ab
 - Bilde alle Kombinationen (D, \dots)
- Beobachtung
 - Benötigt viele Scans der gesamten Datenbasis (äußere Schleife)
 - Sinnvoll: Stufenweises Sortieren
 - 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.