

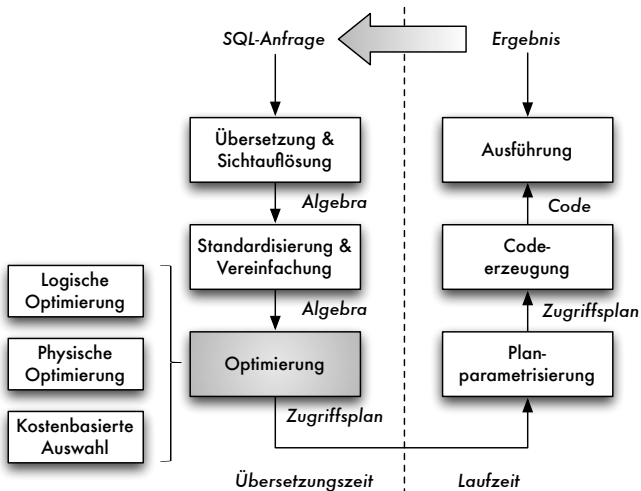
## Teil VIII

### Anfrageverarbeitung und -optimierung

# Anfrageverarbeitung und -optimierung

- 3 Überblick
- 4 Star-Join
- 5 Optimierung des GROUP BY
- 6 Berechnung des CUBE

# Überblick



# Phasen der Anfrageverarbeitung

## 1 Übersetzung und Sichtexpansion

- ▶ Im Anfrageplan arithmetische Ausdrücke vereinfachen
- ▶ Unteranfragen auflösen
- ▶ Einsetzen der Sichtdefinition

## 2 Logische oder auch algebraische Optimierung

- ▶ Anfrageplan unabhängig von der konkreten Speicherungsform umformen; etwa Hineinziehen von Selektionen in andere Operationen

## 3 Physische oder Interne Optimierung

- ▶ Konkrete Speicherungstechniken (Indexe, Cluster) berücksichtigen
- ▶ Algorithmen auswählen
- ▶ Mehrere alternative interne Pläne

## Phasen der Anfrageverarbeitung (2)

### 4 **Kostenbasierte Auswahl**

- ▶ Statistikinformationen (Größe von Tabellen, Selektivität von Attributen) für die Auswahl eines konkreten internen Planes nutzen

### 5 **Planparametrisierung**

- ▶ Bei vorkompilierten Anfragen (etwa Embedded-SQL): Ersetzen der Platzhalter durch Werte

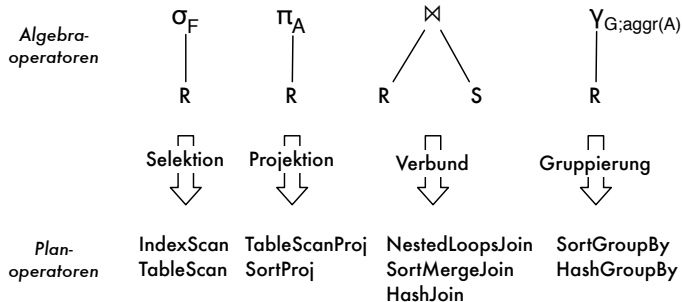
### 6 **Code-Erzeugung**

- ▶ Umwandlung des Zugriffsplans in ausführbaren Code

## Phasen der Anfrageverarbeitung (3)

- Repräsentation von Anfragen während der Verarbeitung
  - ▶ Algebraausdrücke → **Operatorbaum**
    - ★ Operatoren als Knoten
    - ★ Kanten repräsentieren Datenfluss
  - ▶ Spätere Phasen → **Zugriffs- oder Anfrageplan** (query execution plan – QEP)
    - ★ Konkrete Algorithmen als Operatorknoten

# Logische vs. physische Operatoren



- $R, S$  – Relationen
- $A$  – Attributmenge
- $F$  – Bedingung
- $G$  – Gruppierungselemente

## Optimierung von Star-Joins

- Star-Joins stellen typisches Muster für Data-Warehouse-Anfragen dar
- Typische Eigenschaften durch das Star-Schema:
  - ▶ Sehr große Faktentabelle
  - ▶ Deutlich kleinere, unabhängige Dimensionstabellen

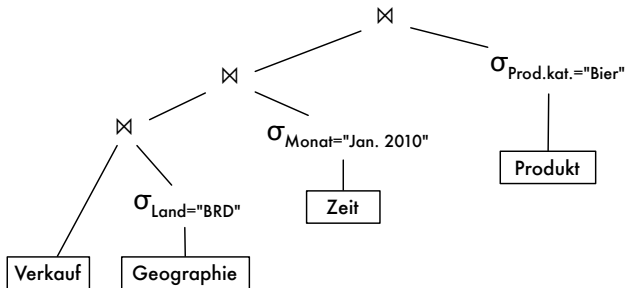
⇒ **Heuristiken klassischer relationaler Optimierer schlagen hier oft fehl!**

## Optimierung von Star-Joins (2)

- **Beispiel: Join über Faktentabelle Verkauf und die drei Dimensionstabellen Produkt, Zeit und Geographie:**
  - ▶ 4-Wege Join
  - ▶ In RDBMS üblicherweise nur paarweiser Join: Sequenz paarweiser Joins notwendig
  - ▶ 4! mögliche Join-Reihenfolgen
  - ▶ Heuristik zur Verringerung der zu untersuchenden Möglichkeiten: Joins zwischen Relationen, die nicht durch eine Join-Bedingung in der Anfrage verknüpft sind, werden nicht betrachtet

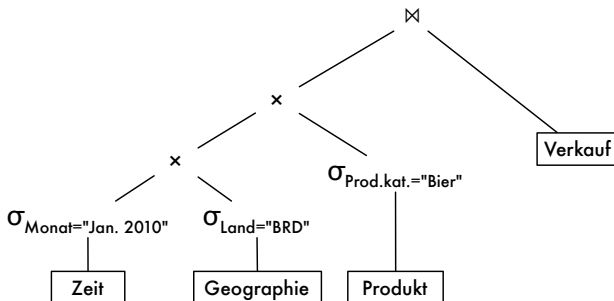
## Optimierung von Star-Joins (3)

- Heuristik liefert z.B. folgenden Ausführungsplan (Plan A):



## Optimierung von Star-Joins (4)

- Folgender Ausführungsplan (Plan B) wird üblicherweise nicht betrachtet (mit Kreuzprodukt der Dimensionstabellen):



# Star-Join: Berechnungsbeispiel

- Annahmen:

- ▶ Tabelle Verkauf: 10.000.000 Datensätze
- ▶ 10 Geschäfte in Deutschland (von 100)
- ▶ 20 Verkaufstage im Januar 2010 (von 1000 gespeicherten Tagen)
- ▶ 50 Produkte in Produktkategorie „Bier“ (von 1000)
- ▶ Gleichverteilung/gleiche Selektivität der einzelnen Ausprägungen

Plan	Operation	Anzahl Ergebnistupel
A	1. Join	1.000.000
	2. Join	20.000
	3. Join	1.000
B	1. Kreuzprodukt	200
	2. Kreuzprodukt	10.000
	3. Join	1.000

## Semi-Join von Dimensionstabellen

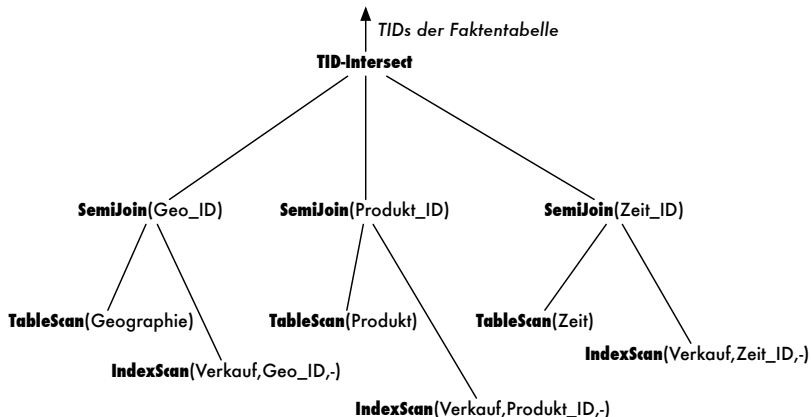
- Berechnung des Kreuzprodukts für die Dimensionstabellen nur bei ausreichend restriktiven Selektionsbedingungen für Dimensionen sinnvoll
- Vermeidung der vollständigen Berechnung des Kreuzproduktes  
↪ Einsatz des **Semi-Join**

## Semi-Join von Dimensionstabellen (2)

- 1 Auf Faktentabelle muss für jede Dimension ein einfacher B+-Baum als Index angelegt sein
- 2 Durch Semi-Joins mit den Dimensionstabellen werden zunächst die Mengen der Tupelidentifizier (TID) der in Frage kommenden Tupel bestimmt
- 3 Die Schnittmenge dieser TID-Mengen wird berechnet (z.B. durch effiziente Hauptspeicherverfahren):
  - ▶ Enthält alle TIDs der Tupel, die die Bedingungen für alle Dimensionen erfüllen
- 4 Danach Durchführung des „normalen“ paarweisen Joins

Nicht die ganze Faktentabelle geht in Join ein, sondern nur noch die tatsächlich relevanten Tupel! (im Beispiel: 1.000 statt 10.000.000 Tupel)

# Semi-Join von Dimensionstabellen (3)



# Optimierung des GROUP BY

- Besondere Berücksichtigung der Gruppierungs-/Aggregatoperationen im Rahmen der Optimierung
- Logische/Algebraische Optimierung: „Push-down“ von Gruppierungen  $\rightsquigarrow$  Reduzierung der Kardinalität der Zwischenergebnisse
  - ▶ Invariantes Gruppieren
  - ▶ Frühzeitiges Vorgruppieren
- Physische/Interne Optimierung
  - ▶ Spezielle Implementierungen für **GROUP BY**, **CUBE** und anderen OLAP-Funktionen

# Invariantes Gruppieren

- Idee: Verschieben einer Gruppierungsoperation nach „unten“ (Invarianz bzgl. Position)
- Anwendbar, wenn
  - ▶ Verbundpartner nicht direkt zum Ergebnis beiträgt (implizite Selektion)
  - ▶ Gruppierungsattribute die Rolle eines Fremdschlüssels im Verbund einnehmen
- Beispiel:

```
SELECT V.Zeit_ID, V.Geo_ID, SUM(Umsatz)
FROM Verkauf V, Zeit, Geographie
WHERE Verkauf.Zeit_ID=Zeit.Zeit_ID AND
      Verkauf.Geo_ID=Geographie.Geo_ID
      AND Jahr < 2010 AND Bundesland <> 'THÜR'
GROUP BY V.Zeit_ID, V.Geo_ID
```

# Invariantes Gruppieren (2)



## Frühzeitiges Vorgruppieren

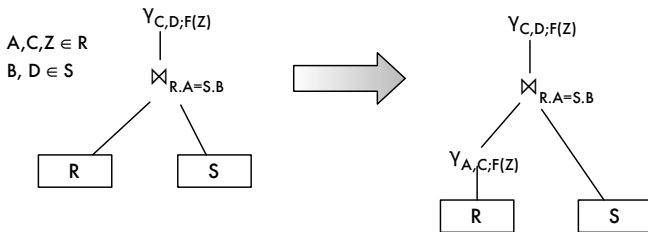
- Invariantes Gruppieren: restriktive Vorbedingung  $\rightsquigarrow$  selten anwendbar
- Idee: Einfügen eines zusätzlichen Gruppierungsoperators vor einen Verbund (ähnlich Projektion)
- Anwendbar, wenn
  - ▶ Gruppierungsbedingung enthält Verbundattribute
  - ▶ Aggregierte Attribute beziehen sich nicht auf Attribute des Verbundpartners

## Frühzeitiges Vorgruppieren (2)

- Beispiel:

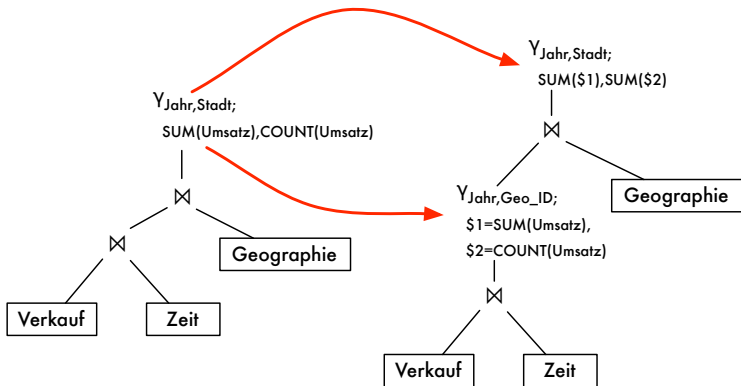
```
SELECT Jahr, Stadt, SUM(Umsatz), COUNT(Umsatz)
FROM Verkauf, Zeit, Geographie
WHERE Verkauf.Zeit_ID = Zeit.Zeit_ID AND
      Verkauf.Geo_ID = Geographie.Geo_ID
GROUP BY Jahr, Stadt
```

# Frühzeitiges Vorgruppieren (3)



## Frühzeitiges Vorgruppieren (4)

- Außerdem notwendig: Anpassung der Aggregationsfunktionen



# Implementierung des Gruppierungsoperators

- Implementierung von **GROUP BY**
- Realisierung von OLAP-Funktionen
- Berechnung von **CUBE**
- Iceberg-Cubes

# GROUP BY-Implementierungen

## ● Sortierungsbasiert

- ▶ Vorsortierung der Relation bzw. sortiertes Lesen (Index-Scan)
- ▶ Ablauf
  - 1 Sortierung
  - 2 Durchlaufen der Tupelliste
  - 3 Aggregieren der Werte und Ausgabe des Aggregatwertes bei Gruppenwechsel

## ● Hashbasiert

- ▶ Hashfunktion über Gruppierungsattribute  $h(G_1, \dots, G_n)$
- ▶ Ablauf
  - 1 Einordnen der Tupel in Hashtabelle mittels  $h(G_1, \dots, G_n)$
  - 2 Durchlaufen der Hashtabelle
  - 3 Anwendung der Aggregatfunktionen

# Realisierung von OLAP-Funktionen

- Sequenzielle Auswertung
- Für jede OLAP-Funktion:
  - ▶ Eingabedaten entsprechend **OVER()**-Klausel sortieren
  - ▶ Aggregationsfunktion anwenden
- Sortierung nach
  - ▶ Attributen der globalen Gruppierung
  - ▶ Attributen der **OVER()**-Klausel (**PARTITION BY** und **ORDER BY**)
- Bei mehreren OLAP-Funktionen in einer Anfrage
  - ▶ Sequenzielle Auswertung, d.h. eventuell wiederholtes Sortieren bei ggf. Nutzung von gemeinsamen Sortierpräfixen

## Realisierung von OLAP-Funktionen (2)

globale Gruppierungsattribute  $G_1 \dots G_n$ ,

lokale Sortierungsattribute aus **OVER**():  $O_1 \dots O_p$

Aggregatfunktion **AGG**(),

lokale Partitionierungsattribute (opt.)  $P_1 \dots P_m$ ,

untere und obere Fenstergrenze (opt.)  $W_u \dots W_o$

```
sort( $G_1, \dots, G_n, P_1, \dots, P_m, O_1, \dots, O_p$ );
```

```
while ( $t = \text{next\_tuple}()$ ) {
```

```
    if ( $t$  hat gleiche Werte bzgl.  $G_1 \dots G_n, P_1 \dots P_m$  wie letztes Tupel)
```

```
         $\text{aggrlist} := \text{concat}(\text{aggrlist}, t)$ ;
```

```
    else {
```

```
        // Partitionswechsel
```

```
        for  $i := 1$  to  $\text{length}(\text{aggrlist})$  {
```

```
            // Berechne absolute Fenstergrenzen low, high
```

```
             $\text{aggrval} := \text{AGG}(\{\text{aggrlist}[\text{low}] \dots \text{aggrlist}[\text{high}]\})$ ;
```

```
             $\text{output}(G_1, \dots, G_n, P_1, \dots, P_m, O_1, \dots, O_p, \text{aggrval})$ ;
```

```
        }
```

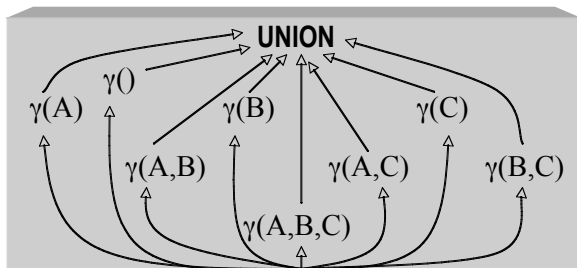
```
         $\text{aggrlist} := ()$ ;
```

```
    }
```

```
}
```

## Berechnung des CUBE: naiver Ansatz

- Separate Berechnung der einzelnen Gruppierungskombinationen
- Abschließende Vereinigung



# CUBE und Aggregationsfunktionen

- Algebraische Funktionen ermöglichen
  - ▶ Berechnung weniger detaillierter Aggregate aus höher detaillierten Aggregaten (mehr Dimensionen)
  - ▶ Partielle Ordnung („Gitter“) der **GROUP BY**-Operationen des **CUBE**
    - ★ **Aggregationsgitter**
  - ▶ **GROUP BY** ist Kind eines anderen **GROUP BY** wenn Eltern-Operation zur Berechnung der Kind-Operation genutzt werden kann → **Ableitbarkeit**

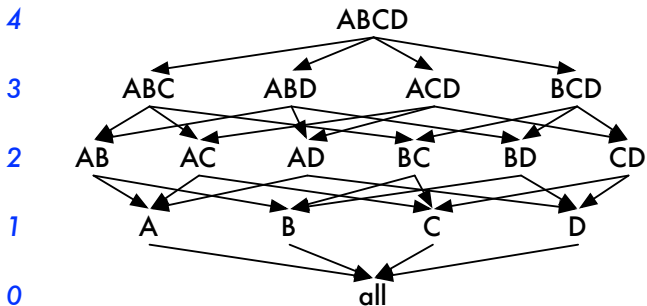
# Ableitbarkeit

- Ableitbarkeit von Gruppierungskombinationen  $G_i$
- Direkte Ableitbarkeit:
  - ▶  $G_2$  ist von  $G_1$  ableitbar, wenn
  - ▶  $G_2$  hat genau ein Attribut weniger als  $G_1$ :  $G_2 \subset G_1$  und  $|G_2| = |G_1| - 1$
  - ▶ Oder in  $G_1$  wird genau ein Attribut  $A_i$  durch  $B_i$  ersetzt, wobei gilt:  
 $A_i \rightarrow B_i$

## ⇒ Aggregationsgitter

- Ableitbarkeit:
  - ▶ Gruppierungskombination  $G_2$  ist innerhalb eines Aggregationsgitters von  $G_1$  ableitbar, wenn Pfad von  $G_1$  nach  $G_2$  existiert

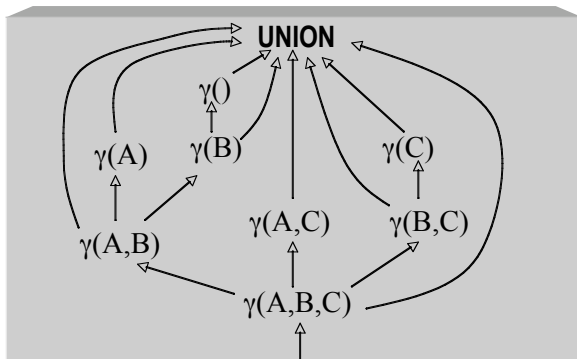
# Aggregationsgitter



# Berechnung von CUBE

- Idee:
  - ▶ Nutzung der Gitter-Sicht (Ableitbarkeit)
  - ▶ GROUP-BYs mit gemeinsamen Gruppierungsattributen können Partitionen, sortierte Bereiche etc. gemeinsam nutzen
- Verfahren
  - ▶ Basierend auf Sortierung: PipeSort
  - ▶ Basierend auf Hashing: PipeHash

# Optimierte Berechnung: Prinzip



# Optimierungspotenzial

- Smallest-parent
  - ▶ Berechnung eines GROUP-BY auf Basis des minimalen zuvor berechneten Eltern-GROUP-BY
- Cache-results
  - ▶ Zwischenspeichern (im Hauptspeicher) der Ergebnisse eines GROUP-BY für nachfolgende GROUP-BY (Bsp.:  $ABC \rightarrow AB$ )
- Amortize-scans
  - ▶ Gemeinsames Berechnen mehrerer GROUP-BY in einem Scan (Bsp.:  $ABC, ACD, ABD, BCD$  aus  $ABCD$ )
- Share-sorts
  - ▶ Für Sortier-basierte Verfahren: Zwischenspeichern und gemeinsames Nutzen von sortiertem Bereichen
- Share-partitions
  - ▶ Für Hash-basierte Verfahren: Zwischenspeichern und gemeinsames Nutzen von Partitionen

## Bedeutung der Sortierreihenfolge

- Annahme: Gruppierung auf Basis von Sortierung
  - ▶ erfordert u.U. Umsortierung
- Beispiel:

<b>Produkt</b>	<b>Jahr</b>	<b>Region</b>	<b>Verkauf</b>
Rotwein	2009	SANH	230
Rotwein	2009	THÜR	210
Rotwein	2010	SANH	200
...	...	...	...
Bier	2009	SANH	568

- ▶ Umsortierung für Gruppierung (Produkt, Region)
- Berücksichtigung der Sortierkosten in einem Kostenmodell

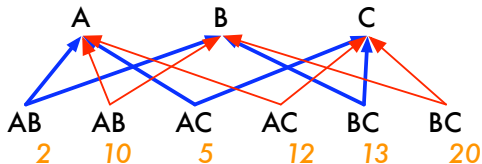
# Kostenmodell

- Kostenarten
  - ▶ *S-Kosten* (Still to sort): Berechnung von GROUP-BY  $j$  aus GROUP-BY  $i$ , wenn  $i$  noch nicht sortiert ist
  - ▶ *A-Kosten* (Already sorted): Berechnung von GROUP-BY  $j$  aus GROUP-BY  $i$ , wenn  $i$  bereits sortiert ist
- Abschätzung auf Basis der Datenverteilung, Systemparameter etc.

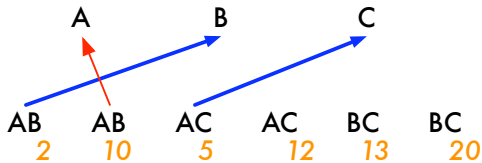
# PipeSort

- Eingabe: Suchgitter
  - ▶ Graph mit Knoten zur Repräsentation von GROUP-BY (Aggregationsgitter)
  - ▶ Gerichtete Kante verbindet GROUP-BY  $i$  mit GROUP-BY  $j$ 
    - ★  $i$  ist Elternknoten von  $j$
    - ★  $j$  kann generiert werden aus  $i$
    - ★  $j$  hat genau ein Attribut weniger als  $i$
  - ▶ Ebene  $k$  bezeichnet alle GROUP-BYs mit  $k$  Attributen
- Annotation von Kanten  $e_{ij}$  mit A- und S-Kosten
- Ausgabe: Subgraph des Suchgitters
  - ▶ Jeder Knoten ist mit einem einzigen Elternknoten verbunden
  - ▶ Bestimmt Sortierreihenfolge bei Erhalt des Pipelining
    - ★ Spezieller aufspannender Baum
- Ziel: derartiger Subgraph mit minimaler Summe der Kantenkosten

# PipeSort: Beispiel



→ Pipeline     
 → Sortierung



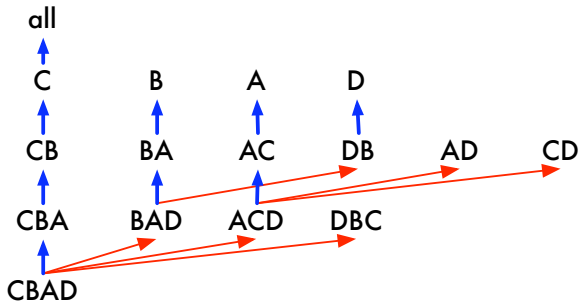
# PipeSort: Algorithmus

- Zurückführen auf bekannten Graphen-Algorithmus
- Ebenenweises Vorgehen:  $k = 0..N - 1$  ( $N$ : Anzahl der Attribute)
- Transformation der Ebene  $k + 1$  durch  $k$  Kopien jedes Knoten
- Jeder kopierte Knoten hat Verbindungen mit den gleichen Knoten wie Original
- Kantenkosten für Originalknoten:  $A(e_{ij})$ , sonst:  $S(e_{ij})$
- Graph mit minimalen Kosten suchen
  - ▶ Paarbildung und Minimierung der Gesamtkosten („ungarische“ Methode – *weighted bipartite matching problem*)

## PipeSort: Sortierreihenfolge

- Jeder Knoten  $h$  in Ebene  $k$  ist verbunden mit einem Knoten  $g$  in Ebene  $k + 1$
- $h \rightarrow g$  über  $A()$ -Kante:  $h$  bestimmt Attribut Reihenfolge für Sortierung von  $g$
- $h \rightarrow g$  über  $S()$ -Kante:  $g$  wird neu sortiert für Berechnung von  $h$

# PipeSort: Sortierplan



Relation



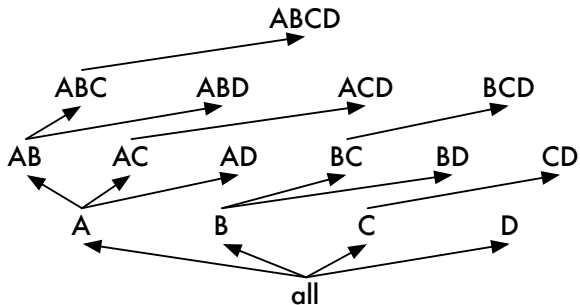
## Iceberg-Cubes

- Idee: Ausnutzung der Monotonie von Aggregationen

Falls eine Aggregationsgruppe die **COUNT**-Bedingung nicht erfüllt, dann wird diese auch nicht durch Gruppen mit zusätzlichen Attributen erfüllt.

- Ansatz: Bottom-Up-Konstruktion eines Cubes und Minimal-Support-Pruning (ähnlich Apriori)

# Iceberg Cube bottom up



# Iceberg-Cube: Berechnung

```
BottomUpCube(input, dim):
  aggregate(input);
  write(outputRec);
  for (d:=dim; d<numDims; d++) {
    C := cardinality[d]; /* Kardinalität der Dimension */
    Partition(input, d, C, dataCount[d]);
    k := 0;
    for (i:=0; i < C; i++) {
      c := dataCount[d][i]; /* Größe der Partition */
      if (c >= minsup) {
        outputRec.dim[d] := input[k].dim[d];
        BottomUpCube(input[k...k+c], d+1);
      }
      k += c;
    }
    outputRec.dim[d] = ALL;
  }
```

# Zusammenfassung

- Spezielle Charakteristika von DW-Anfragen erfordern spezifische Optimierungsmaßnahmen
- Rewriting-Techniken:
  - ▶ Verbundreihenfolge beim Star-Join
  - ▶ Push-down von Gruppierungen
- Operator-Implementierungen
  - ▶ CUBE und Iceberg-CUBE
  - ▶ OLAP-Funktionen