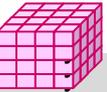


# 5. Performance-Techniken

- Einleitung
- Indexstrukturen
  - ein- vs. mehrdimensionale Indexstrukturen
  - eindimensionale Einbettung
  - Bitlisten-Indexstrukturen
- Column Stores
  - Datenkompression
- Datenpartitionierung
  - vertikale vs. horizontale Fragmentierung
  - mehrdimensionale, hierarchische Fragmentierung
- Materialisierte Sichten
  - Verwendung
  - Auswahl



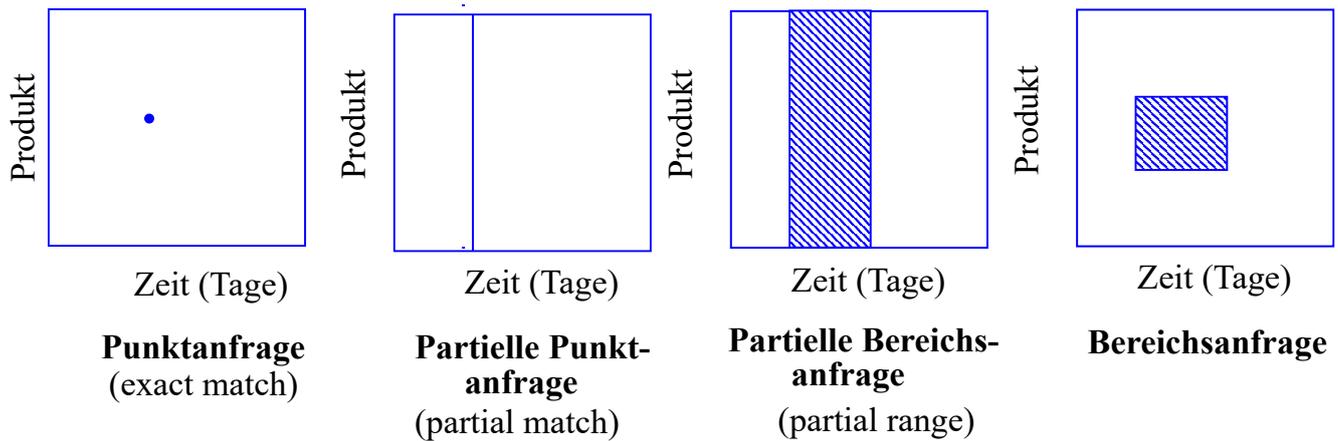
## Einleitung

- hohe Leistungsanforderungen
  - sehr große Datenmengen, vor allem Faktentabelle
  - kurze Antwortzeiten für viele Benutzer
  - mehrdimensionale Auswahlbedingungen, Gruppierung, Aggregationen, Sortierung ...
  - periodische Aktualisierung mit vielen Änderungen für DW-Tabellen und Data Marts
- Scan-Operationen auf der Faktentabelle i.a. nicht akzeptabel
  - Bsp.: 1 TB, Verarbeitungsgeschwindigkeit 50 MB/s
- Standard-Verfahren (z.B. Hash-Join) oft zu ineffizient für Star Join
- Einsatz mehrerer DW-spezifischer Performance-Techniken
  - Indexstrukturen (1-dimensional, mehrdimensional, Bit-Indizes)
  - Column Store
  - Partitionierung der Daten (Einschränkung der zu bearbeitenden Daten) und Parallelverarbeitung
  - materialisierte Sichten bzw. vorberechnete Aggregationen

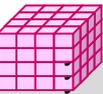


# Mehrdimensionale Anfragearten

## ■ Punkt- und Bereichsanfragen (exakt vs. partiell)



## ■ Aggregation, Gruppierung (Cube, Rollup), Sortierung ...



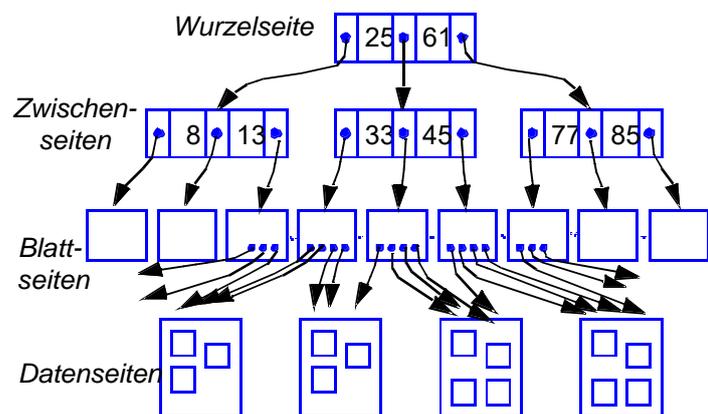
# Indexstrukturen

## ■ Optimierung selektiver Lesezugriffe: Reduzierung der für Anfrage zu lesenden Datenseiten

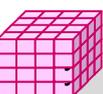
- zusätzlicher Speicherbedarf und Änderungsaufwand

## ■ Standard-Indexstruktur: **B\*-Baum**

- eindimensionaler Index (1 Attribut bzw. Attributkombination)
- Primär- oder Sekundärindex
- balanciert, gute Speicherbelegung
- mit / ohne Clusterung der Datensätze
- geringe Höhe auch bei sehr großen Tabellen
- Unterstützung von (1-dim.) Punkt- und Bereichsanfragen

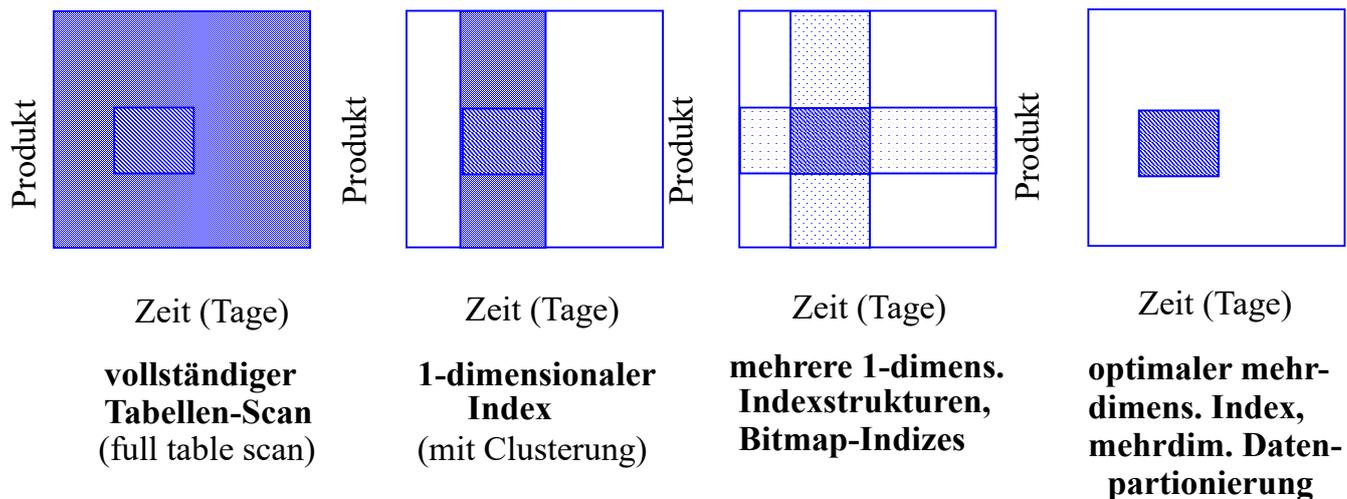


Clustered Index



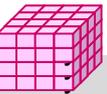
# Indexunterstützung für mehrdimens. Anfragen

## ■ Eingrenzung des Datenraumes



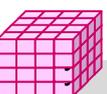
## ■ Bsp. mehrdimensionaler Indexstrukturen

- Grid File
- R-Baum, ...



## Eindimensionale Einbettung

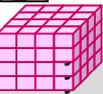
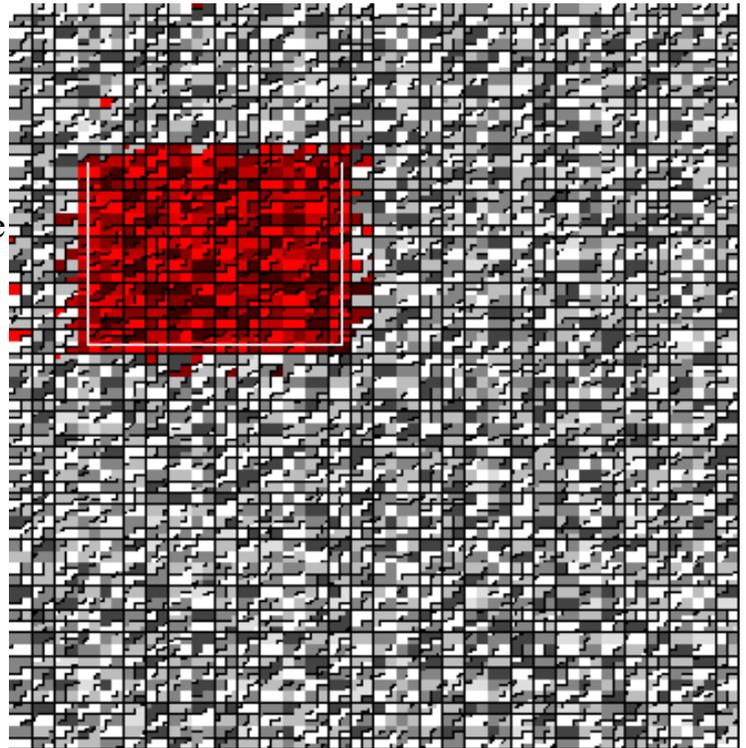
- Transformation mehrdimensionaler Punktobjekte für eindimensionale Repräsentation, z.B. mit B\*-Bäumen
- möglichst Wahrung der topologischen Struktur (Unterstützung mehrdimensionaler Bereichs- und Nachbarschaftsanfragen)
- Ansatz
  - Partitionierung des Datenraums  $D$  zunächst durch gleichförmiges Raster
  - eindeutige Nummer pro Zelle legt Position in der totalen Ordnung fest
  - Reihenfolge bestimmt eindimensionale Einbettung: *space filling curve*
- Zuordnung aller mehrdimensionalen Punktobjekte einer Zelle zu einem Bucket (Seite)
- jede Zelle kann bei Bedarf separat (und rekursiv) unter Nutzung desselben Grundmusters weiter verfeinert werden





# UB-Baum (2)

- exakte mehrdimensionale Anfragen gehen auf 1 Seite
- Bereichssuche (Begrenzung durch zwei Eckpunkte LO und RU)
  - bestimme Z-Wert (Z-Region) zu LO
  - werte Suchprädikat auf alle Sätze in Z-Region aus
  - berechne nächsten Bereich der Z-Kurve innerhalb Anfragebereich
  - wiederhole die beiden vorherigen Schritte bis Endadresse der Z-Region größer ist als RU (diesen Punkt also enthält)
- UB-Baum-Realisierung im Rahmen des relationalen DBS TransBase
- Animationen / Publikationen unter <http://mistral.in.tum.de>



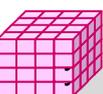
## Bitlisten-Indizes

- herkömmliche Indexstrukturen ungeeignet für Suchbedingungen geringer Selektivität
  - z.B. für Dimensionsattribute mit wenigen Werten (Geschlecht, Farbe, Jahr ...)
  - pro Attributwert sehr lange Verweislisten (TID-Listen) auf zugehörige Sätze
  - nahezu alle Datenseiten zu lesen
- Standard-Bitlisten-Index (**Bitmap Index**):
  - Index für Attribut A umfasst eine Bitliste (Bitmap, Bitvektor)  $B_{A_j}$  für jeden der k Attributwerte  $A_1 \dots A_k$
  - Bitliste umfasst 1 Bit pro Satz (bei N Sätzen Länge von N Bits)
  - Bitwert 1 (0) an Stelle i von  $B_{A_j}$  gibt an, dass Satz i Attributwert  $A_j$  aufweist (nicht aufweist)
  - 1-dimensionaler Index, jedoch flexible Kombinierbarkeit

*Kunde*

KID	Geschlecht	Lieblingsfarbe
122	W	Rot
123	M	Rot
124	W	Weiß
125	W	Blau
...	...	...

Blau 0001100010001100001100000000  
 Rot 1100000000010010000010000001  
 Weiß 0010000001100000010000011110  
 Grün 0000011100000001100001100000 ...



# Bitlisten Indizes (2)

## ■ Vorteile

- effiziente AND-, OR-, NOT-Verknüpfung zur Auswertung mehrdimensionaler Suchausdrücke
- effiziente Unterstützung von Data-Warehouse-Anfragen (Joins)
- geringer Speicherplatzbedarf bei kleinen Wertebereichen ( $k < 100$ )

## ■ Beispielanfrage

Select ... WHERE A1=c1 AND A2=c2 AND A3=c3

100 Millionen Sätze zu je 200 B; pro Teilbedingung Selektivität 1%

ohne Index: Relationengröße 20 GB; Verarbeitung dauert bei 50 MB/s: 400 s

mit Bitlisten:

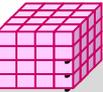
Bitlistengröße 12,5 MB (für c1, c2 und c3) -> 37,5 MB; Einlesen dauert ca 1 s

AND-Verknüpfung führt zu 1% von 1% von 1% von  $10^8$  Sätzen (Bits) = 100 Sätzen

Einlesen von 100 Sätzen (je ca 10 ms): 1 s

Gesamtaufwand:  $1s+1s=2s$  vs 400 s -> Verbesserung um Faktor 200

## ■ Erweiterungen erforderlich für Bereichsanfragen, große Wertebereiche, hierarchische Dimensionen



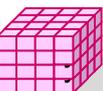
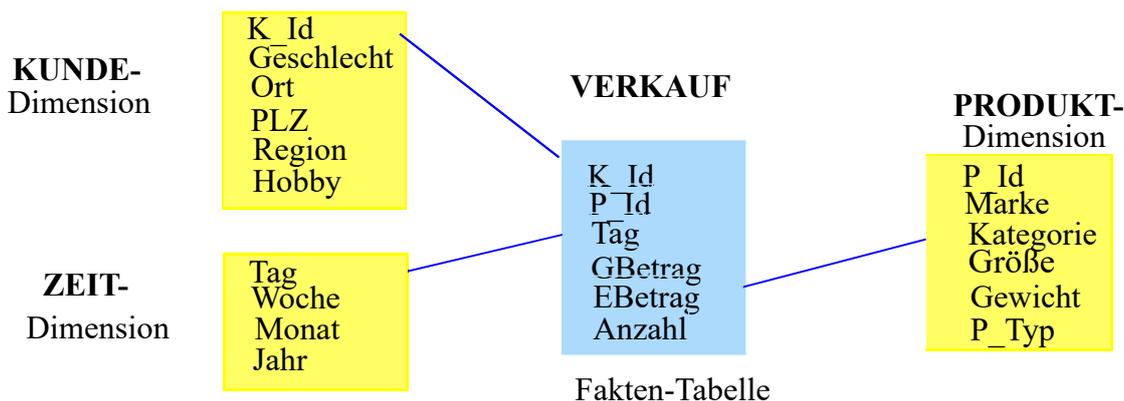
# Bitlisten-Join-Index

## ■ Dimensionsbedingungen vor allem im Rahmen von Star Joins

## ■ vollständige Scans auf Fakten-Tabelle zu verhindern -> Nutzung von Bitlisten-Indizes zur Bestimmung der relevanten Fakten-Tupel

## ■ Bitlisten-Join-Index

- Bitlisten-Index für Dimensions-Attribut auf der Fakten-Tabelle
- Bitliste enthält Bit pro Fakten-Tupel: entspricht vorberechnetem Join
- flexible Kombinierbarkeit für unterschiedliche Anfragen
- Auswertung der Suchbedingungen auf Indizes ermöglicht minimale Anzahl von Datenzugriffen auf die Fakten-Tabelle



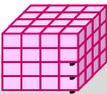
# Bitlisten-Join-Index (2)

- Beispielanfrage: Smartphone-Umsatz und Anzahl verkaufter Geräte im April mit männlichen Kunden

```
select sum (GBetrag), sum (Anzahl)
from VERKAUF v, KUNDE k, PRODUKT p, ZEIT z
where v.K_Id = k.K_Id and v.P_Id = p.P_Id and v.Tag = z.Tag
      and z.Monat = "April" and p.Kategorie = „Smartphone“
      and k.Geschlecht = "M"
```

- Nutzung von 3 Bitlisten-Join-Indizes

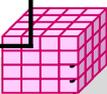
Monat	Kategorie	Geschlecht
Januar 0001100010001 ...	TV 1001000000010 ...	M 1010010001001 ...
Febr 0000011100000 ...	Notebook 0100000001000 ...	W 0111101110110 ...
März 1000000000010 ...	Smartphone 0010100010101 ...	
April 0110000001100 ...	Tablet 0000011100000 ...	
...	...	



## Bereichskodierte Bitlisten-Indizes

- Standard-Bitlisten erfordern für Bereichsanfragen Auswertung vieler Bitlisten
- Bereichskodierte Bitlisten-Indizes:
  - in Bitliste zu Wert *w* bedeutet 1-Bit für einen Satz, dass der Attributwert *kleiner oder gleich w* ist
  - Bitliste für Maximalwert kann eingespart werden (da nur „1“ gesetzt sind)
- Für jede Bereichsanfrage max. 2 Bitlisten auszuwerten
  - Q1: Monat <= „Juni“:
  - Q2: Monat „März“ bis „Juni“:

ID	Monat	Standard-Bitlisten									Bereichskodierte Bitlisten								
		Jan	Feb	Mär	Apr	Mai	Jun	Jul	...	Dez	Jan	Feb	Mär	Apr	Mai	Jun	...	Nov	Dez
122	Mai	0	0	0	0	1	0	0	...	0	0	0	0	0	1	.....	1		
123	März	0	0	1	0	0	0	0	...	0	0	0	1	1	....				1
124	Januar	1	0	0	0	0	0	0	...	0	1	1	1	1	....				1
125	März	0	0	1	0	0	0	0	...	0	0	0	1	1	....				1
126	Februar	0	1	0	0	0	0	0	...	0	0	1	1	1	....				1
127	Dezember	0	0	0	0	0	0	0	...	1	0	0	0	0	...			0	1
128	April	0	0	0	1	0	0	0	...	0	0	0	0	1	....				1
...	...	...									...								



# Bereichskodierte Bitlisten-Indizes (2)

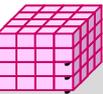
## ■ Bereichsanfragen: Beispiele

- Bereich  $A < x \leq E$ :
- Bereich  $x \leq E$ :
- Bereich  $x > A$ :

## ■ Punktanfrage (Gleichheitsbedingung)

- erfordert (meist) Lesen von 2 Bitlisten vs. 1 Bitliste bei Standard-Bitlisten-Index
- nur Juni:  $B(\text{Juni}) \text{ AND}$

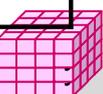
## ■ Speicherplatzreduzierung möglich durch verallgemeinerte *Intervall-Kodierung*



# Intervallkodierte Bitlisten-Indizes

- jede Bitliste repräsentiert Wertezugehörigkeit zu bestimmtem Intervall fester Länge von der Hälfte des Gesamtwertebereichs
  - Beispiel  $I1 = [\text{Jan}, \text{Jun}]$ ,  $I2 = [\text{Feb}, \text{Jul}]$ ,  $I3 = [\text{Mär}, \text{Aug}]$ ,  
 $I4 = [\text{Apr}, \text{Sep}]$ ,  $I5 = [\text{Mai}, \text{Okt}]$ ,  $I6 = [\text{Jun}, \text{Nov}]$ ,  $I7 = [\text{Jul}, \text{Dez}]$
- für jede Punkt- und Bereichsanfrage max. 2 Bitlisten zu lesen  
 z.B. Bereich März bis September:  $I3 \text{ OR } I4$   
 Monat = Februar:
- etwa halbiertes Speicheraufwand

ID	Monat	Standard-Bitlisten									Intervallkodierte Bitlisten						
		Jan	Feb	Mär	Apr	Mai	Jun	Jul	...	Dez	I1	I2	I3	I4	I5	I6	I7
122	Mai	0	0	0	0	1	0	0	...	0	1	1	1	1	1	0	0
123	März	0	0	1	0	0	0	0	...	0	1	1	1	0	0	0	0
124	Januar	1	0	0	0	0	0	0	...	0	1	0	0	0	0	0	0
125	März	0	0	1	0	0	0	0	...	0	1	1	1	0	0	0	0
126	Februar	0	1	0	0	0	0	0	...	0	1	1	0	0	0	0	0
127	Dezember	0	0	0	0	0	0	0	...	1	0	0	0	0	0	0	1
128	April	0	0	0	1	0	0	0	...	0	1	1	1	1	0	0	0
...	...	...															...



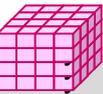
# Kodierte Bitlisten-Indizes

- Speicherplatzersparnis durch logarithmische Kodierung der k möglichen Attributwerte (encoded bitmap indexing)
  - Standardverfahren: pro Satz ist nur in einer der k Bitlisten das Bit gesetzt
  - jede der k Wertemöglichkeiten wird durch  $\log_2 k$  Bits kodiert  $\Rightarrow$  nur noch  $\log_2 k$  Bitlisten
  - hohe Einsparungen bei großen k (z.B. Kunden, Produkte)
    - k=1 Million erfordert 20 statt 1 Million Bitlisten

		<b>Kodierung</b>	<b>F<sub>1</sub></b>	<b>F<sub>0</sub></b>	
Blau	0001100010001 ...	Blau	0	0	2 Bitvektoren für 4 Farben F <sub>1</sub> 0 0 1 0 0 1 ... F <sub>0</sub> 1 1 0 0 0 1 ...
Rot	1100000000010 ...	Rot	0	1	
Weiß	0010000001100 ...	Weiß	1	0	
Grün	0000011100000 ...	Grün	1	1	

## ■ Auswertung von Suchausdrücken

- höherer Aufwand bei nur 1 Bedingung ( $\log_2 k$  Bitlisten statt 1 abzuarbeiten)  
Farbe = Blau:
- bei mehreren Bedingungen wird auch Auswertungsaufwand meist reduziert
- Blau oder Rot:



# Kodierungsvarianten

## ■ Mehrkomponenten-Bit-Indizes

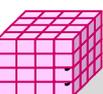
- Zerlegung von Attributwerten in mehrere Komponenten und separate Kodierung
- Wahl der Komponenten erlaubt Kompromiss zwischen Speicheraufwand (# Bitlisten) und Zugriffsaufwand

## ■ Beispiel 1

- Produkt-Nr (0..999) =  $x * 100 + y * 10 + z$  mit x,y,z aus 0..9
- 30 Bitlisten statt 1000 (10 bei logarithmischer Kodierung)
- Auswertung eines Produktes:

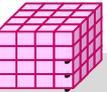
## ■ Beispiel 2

- hierarchische Kodierung zB von Produkten innerhalb Produktart und Produktgruppe
- zB 5/6/8 Bits für Produktgruppe/Produktart/Produkt (Muster gggggaaaaaappppppppp)  
führt zu 19 Bitlisten für max.  $32 * 64 * 256 = 500K$  Produkte
- Auswertungen für Produktgruppe und Produktart ohne eigene Bitlisten



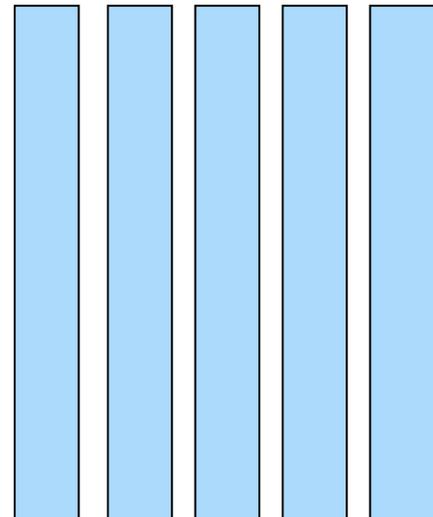
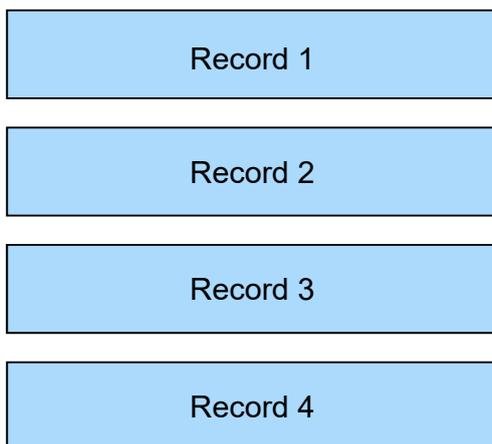
# 5. Performance-Techniken

- Einleitung
- Indexstrukturen
  - ein- vs. mehrdimensionale Indexstrukturen
  - Bitlisten-Indexstrukturen
- Column Stores
  - Datenkompression
- Datenpartitionierung
  - vertikale vs. horizontale Fragmentierung
  - mehrdimensionale, hierarchische Fragmentierung
- Materialisierte Sichten
  - Verwendung
  - Auswahl



## Column Stores

- spaltenweise statt zeilenweise Speicherung von DB/DW-Inhalten
  - v.a. im Rahmen von In-Memory-Architekturen genutzt
- Beispiel-Systeme:
  - Sybase IQ
  - Vertica (kommerzielle Version von C-Store), Infobright ICE, Monet
  - SAP HANA, IBM Blu ...



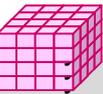
# Vorteile / Nachteile

## ■ Vorteile Column Store

- I/O-Einsparungen falls nur wenige Attribute benötigt
- effiziente Aggregationsmöglichkeiten
- OLAP-orientiert
- oft deutlich bessere Trefferraten im CPU-Cache durch Fokussierung auf relevante Daten

## ■ Nachteile Column Store (-> Vorteile Row Store)

- ungünstig für Operationen, die (fast) alle Attribute von Tupeln betreffen, z.B. für Änderungen / effizientes Einfügen neuer Tupel
- weniger günstig für OLTP und wenn nicht alle Daten im Hauptspeicher

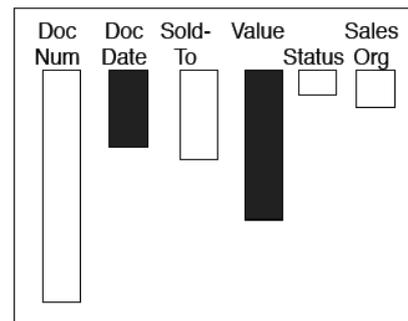
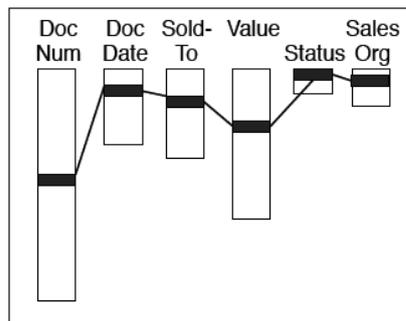


## Anfragen – Column vs. Row Store \*

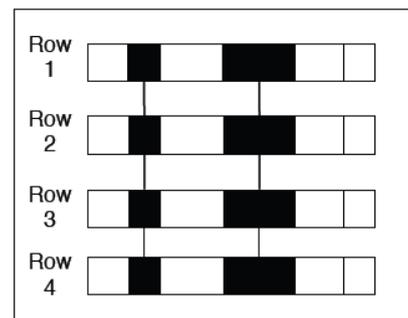
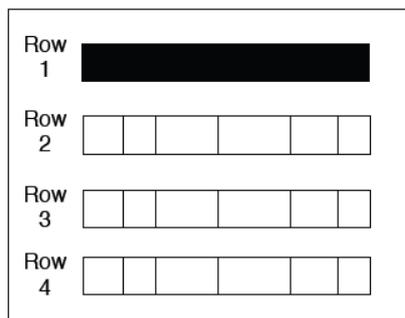
```
SELECT *
FROM Sales Orders
WHERE Document Number = '95779216'
```

```
SELECT SUM(Order Value)
FROM Sales Orders
WHERE Document Date > 2009-01-20
```

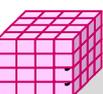
Column Store



Row Store

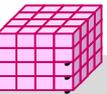


\* Quelle: Hasso Plattner: Enterprise Applications – OLTP and OLAP – Share One Database Architecture



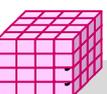
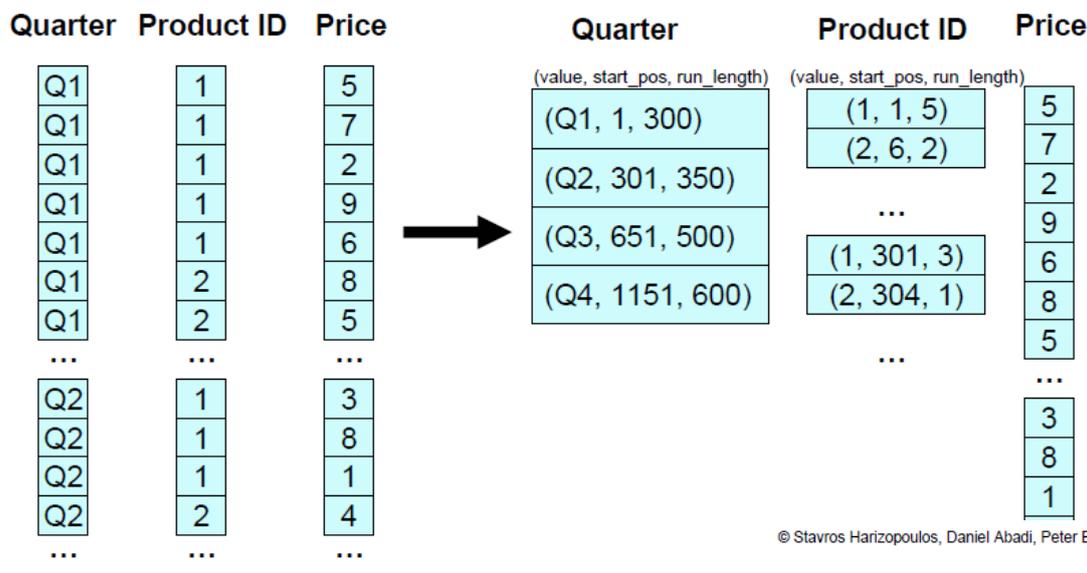
# Datenkompression

- Column Stores nutzen oft komprimierte Speicherung von Attributwerten
  - reduzierter Speicherbedarf
  - reduzierter I/O-Aufwand
- leichtgewichtige Kompression mit geringem Aufwand zur Dekomprimierung
  - wünschenswert: Auswertungen auf komprimierten Werten selbst
- Varianten:
  - Run Length Encoding (RLE)
  - Bit Vector Encoding
  - Delta Coding
  - Wörterbuch-Kodierung
  - etc.
- pro Spalte kann das beste Kodierungsverfahren gewählt werden



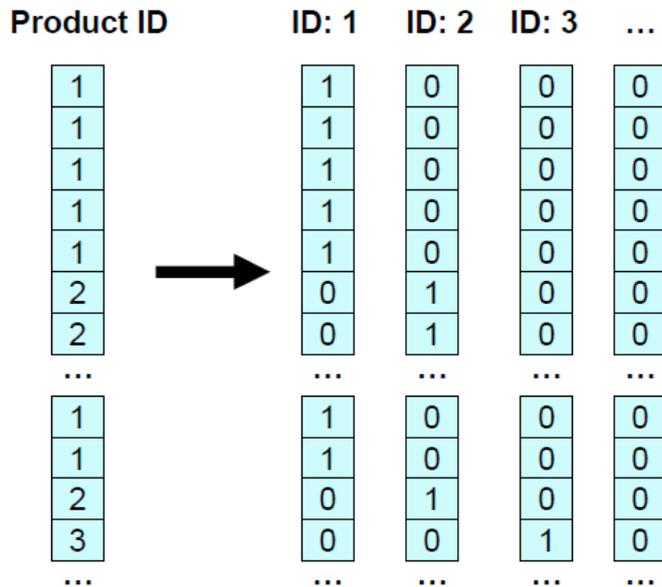
## Run Length Encoding (RLE) / Lauflängenkodierung

- Zusammenfassung von Nachbarn mit gleichem Wert  
(Wert, Startposition, #Vorkommen)
- effizient bei langen Folgen gleicher Werte
  - wird durch Sortierung der Attributwerte pro Spalte unterstützt
- einfache Dekodierung
- Auswertungen auch auf komprimierten Werten möglich

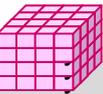


# Bit Vector Encoding

- Spalte wird für k mögliche Attributwerte durch k Bitvektoren repräsentiert
  - Bit 1 (0) für Bitliste j an Position i bedeutet, dass Satz i den Wert j (nicht) hat
- speichergünstig bei wenigen Attributwerten
  - kann mit RLE kombiniert werden (lange 1- bzw. 0-Folgen)
- effiziente Auswertungen von Selektionen

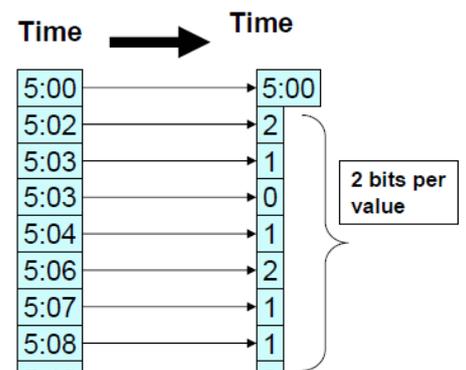


© Stavros Harizopoulos, Daniel Abadi, Peter Boncz (2009)

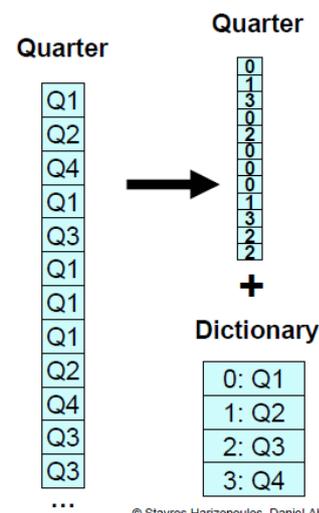


## Weitere Kompressionstechniken

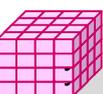
- Delta / Difference Coding
  - Speicherung der Differenz zu Vorgängerwerten
  - oft kompakter, v.a. bei Sortierung
- Wörterbuch-Kodierung
  - erfordert keine Sortierung
  - günstig v.a. bei wenigen und langen Attributwerten
  - Auswertung auf komprimierten Daten möglich
  - Lookup zur Dekomprimierung



Dictionary		IndexVector	
pos	value	pos	value
0	Aachen	0	0
1	Karlsruhe	1	0
2	Leipzig	2	0
3	Münster	3	1
		4	0
		5	0
		6	2
		7	3

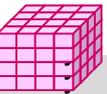


© Stavros Harizopoulos, Daniel Abadi, Peter Boncz (2009)



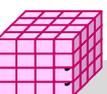
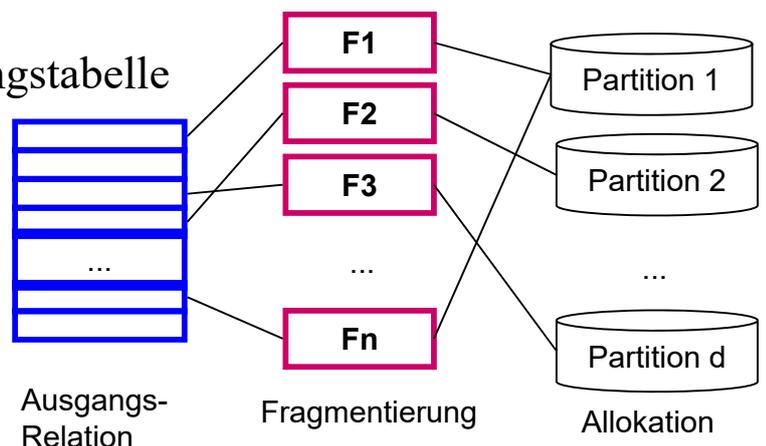
# 5. Performance-Techniken

- Einleitung
- Indexstrukturen
  - ein- vs. mehrdimensionale Indexstrukturen
  - Bitlisten-Indexstrukturen
- Column Stores
  - Datenkompression
- Datenpartitionierung
  - vertikale vs. horizontale Fragmentierung
  - mehrdimensionale, hierarchische Fragmentierung
- Materialisierte Sichten
  - Verwendung
  - Auswahl



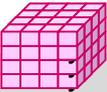
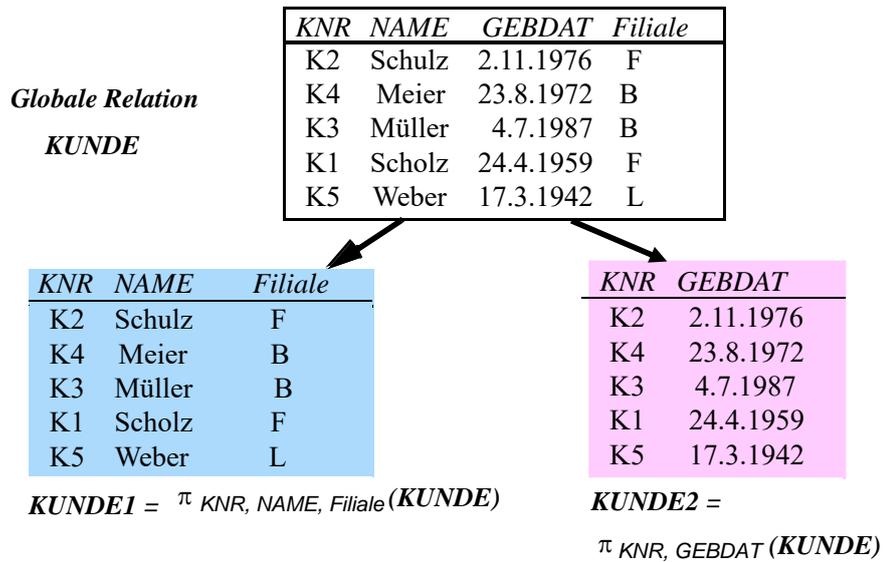
## Datenpartitionierung

- Partitionierung: logische Zerlegung von Relationen
  - *Fragmentierung*: Bestimmung der Verteilungseinheiten
  - *Allokation*: Zuordnung der Fragmente zu Partitionen (Rechnerknoten, Externspeicher)
- Fragmentierung (Zerlegung):
  - horizontal vs. vertikal
  - Vollständigkeit der Zerlegung
  - Rekonstruierbarkeit der Ursprungstabelle
- Ziele
  - Reduzierung des Verarbeitungsumfangs
  - Unterstützung von Parallelverarbeitung
  - Lastbalancierung



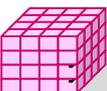
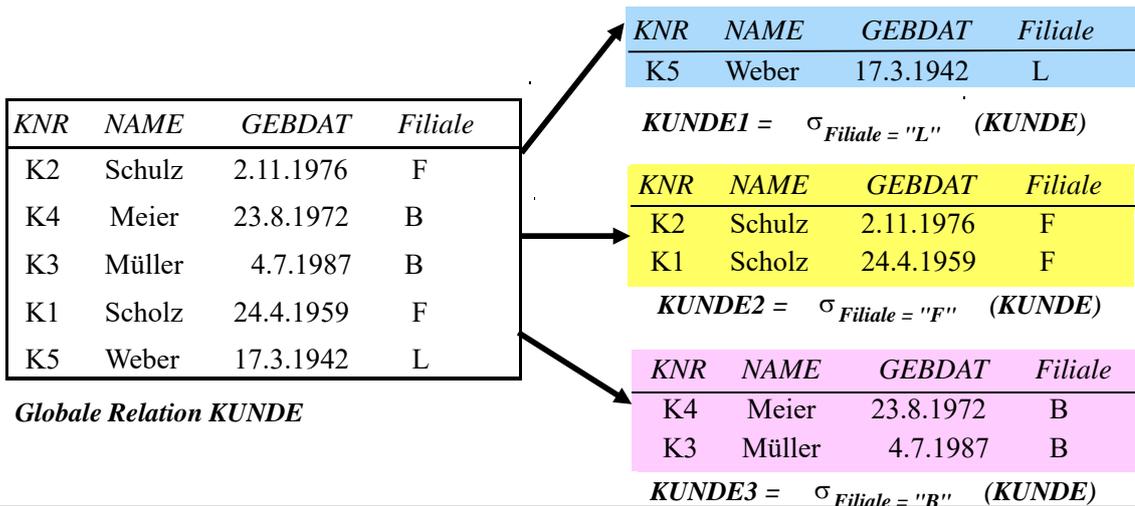
# Vertikale Fragmentierung

- spaltenweise Aufteilung von Relationen
- Definition der Fragmentierung durch Projektion
- Vollständigkeit:
  - jedes Attribut in wenigstens 1 Fragment enthalten
- verlustfreie Zerlegung:
  - Primärschlüssel i.a. in jedem Fragment enthalten
  - JOIN-Operation zur Rekonstruktion des gesamten Tupels
- Arbeitersparnis durch Auslagern selten benötigter Attribute in eigene Fragmente



# Horizontale Fragmentierung

- zeilenweise Aufteilung von Relationen
- Definition der Fragmentierung durch Selektionsprädikate  $P_i$  auf der Relation:
  - $R_i := \sigma_{P_i}(R)$  ( $1 \leq i \leq n$ )
  - Vollständigkeit: jedes Tupel ist einem Fragment eindeutig zugeordnet
  - Fragmente sind disjunkt:  $R_i \cap R_j = \{\}$  ( $i \neq j$ )
  - Verlustfreiheit: Relation ist Vereinigung aller Fragmente:  $R = \cup R_i$  ( $1 \leq i \leq n$ )
- Anfragen auf Fragmentierungsattribut werden auf Teilmenge der Daten begrenzt
- Parallelverarbeitung unterschiedlicher Fragmente

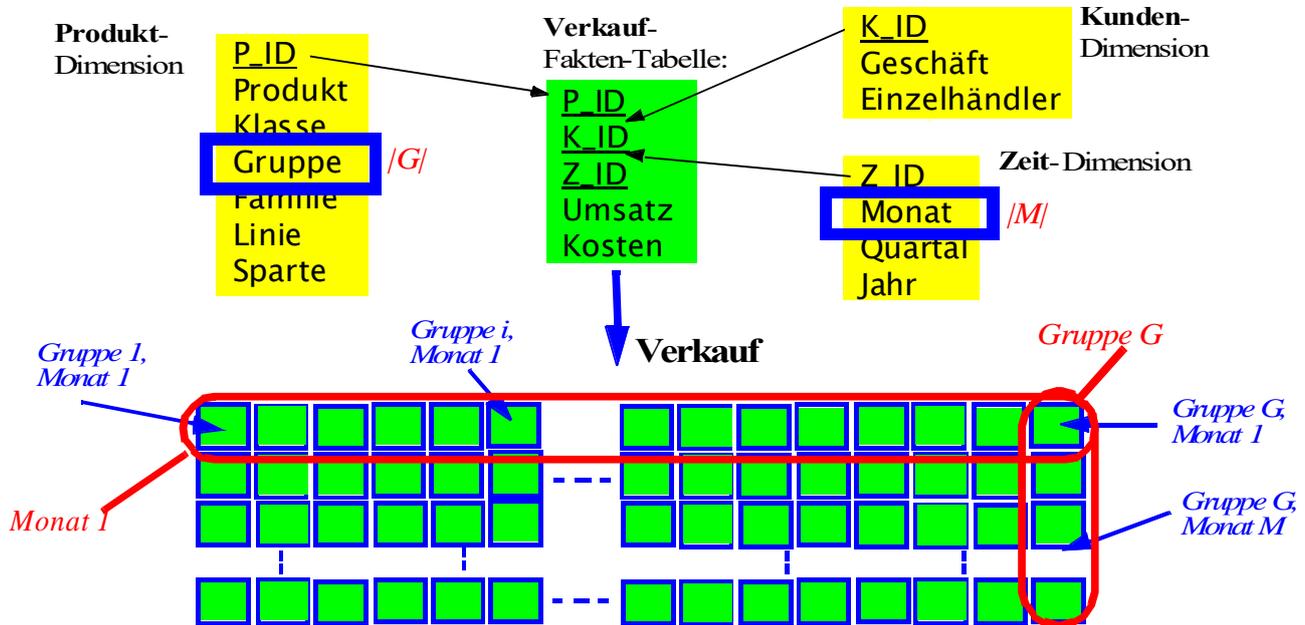


# Multi-dimensionale, hierarchische Fragmentierung (MDHF)\*

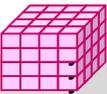
- Horizontale Bereichsfragmentierung auf *mehreren* Attributen (reihenfolge-unabhängig)

- Auswahl höchstens eines Attributs pro Dimension als Fragmentierungsattribut(e)
- Fragment-Einschränkung für alle Anfragen bzgl. Fragmentierungsdimensionen

Beispiel: 2-dimensionale Fragmentierung  $F_{GruppeMonat}$  der Faktentabelle **Verkauf**

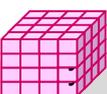
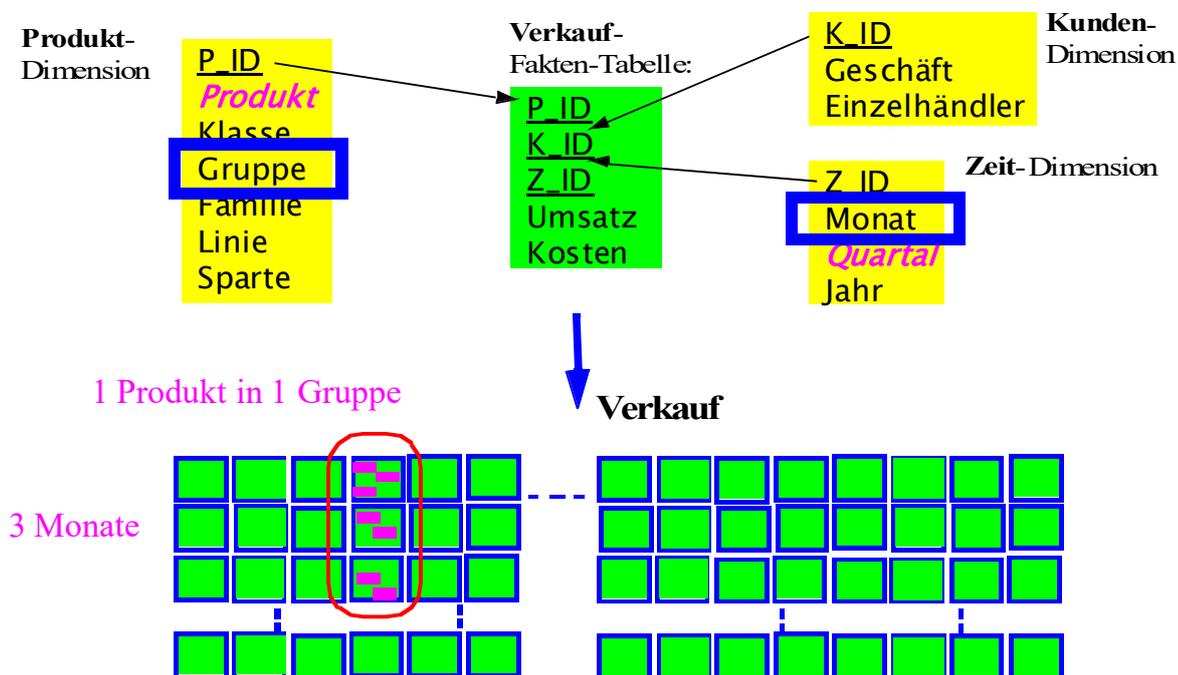


\* Stöhr, T., Märtens, H., Rahm, E.: *Multi-Dimensional Database Allocation for Parallel Data Warehouses* Proc. VLDB, 2000,



## Sternschema-Anfrage unter MDHF

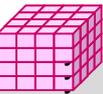
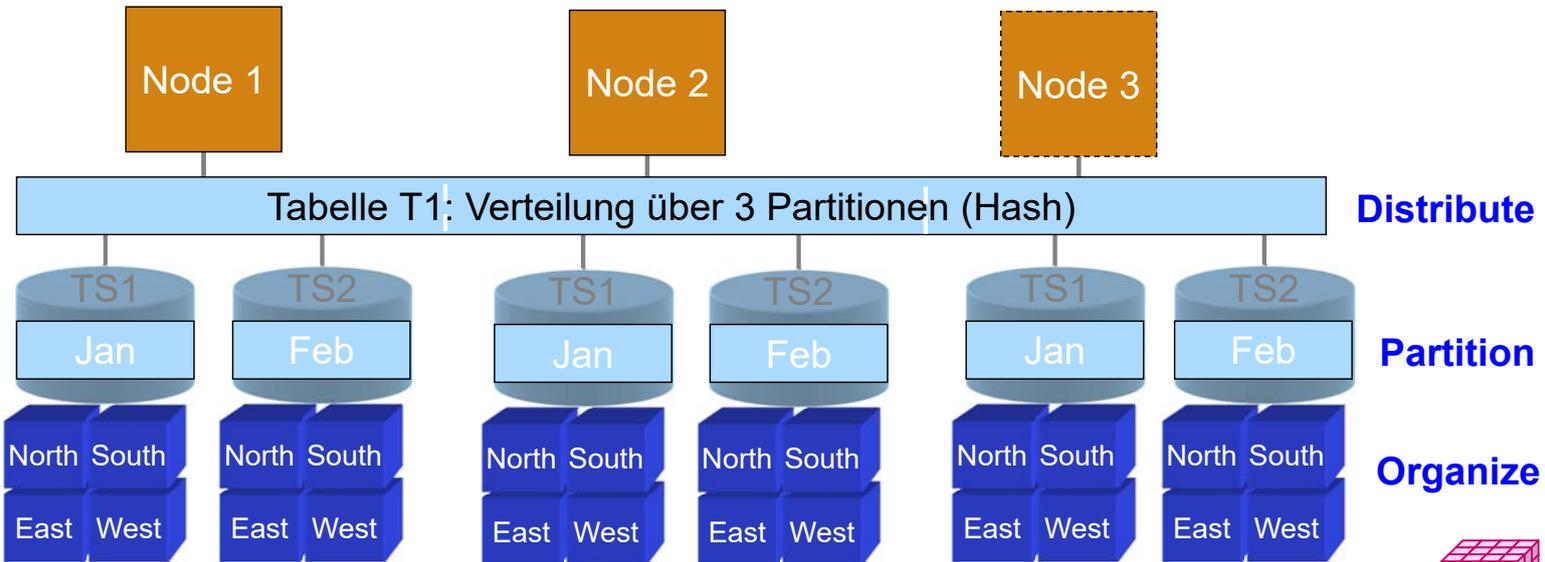
- Zugriff oberhalb und unterhalb des Fragmentierungsebene (Anfrage auf *Quartal* und *Produkt*)
- nur 3 Fragmente



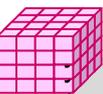
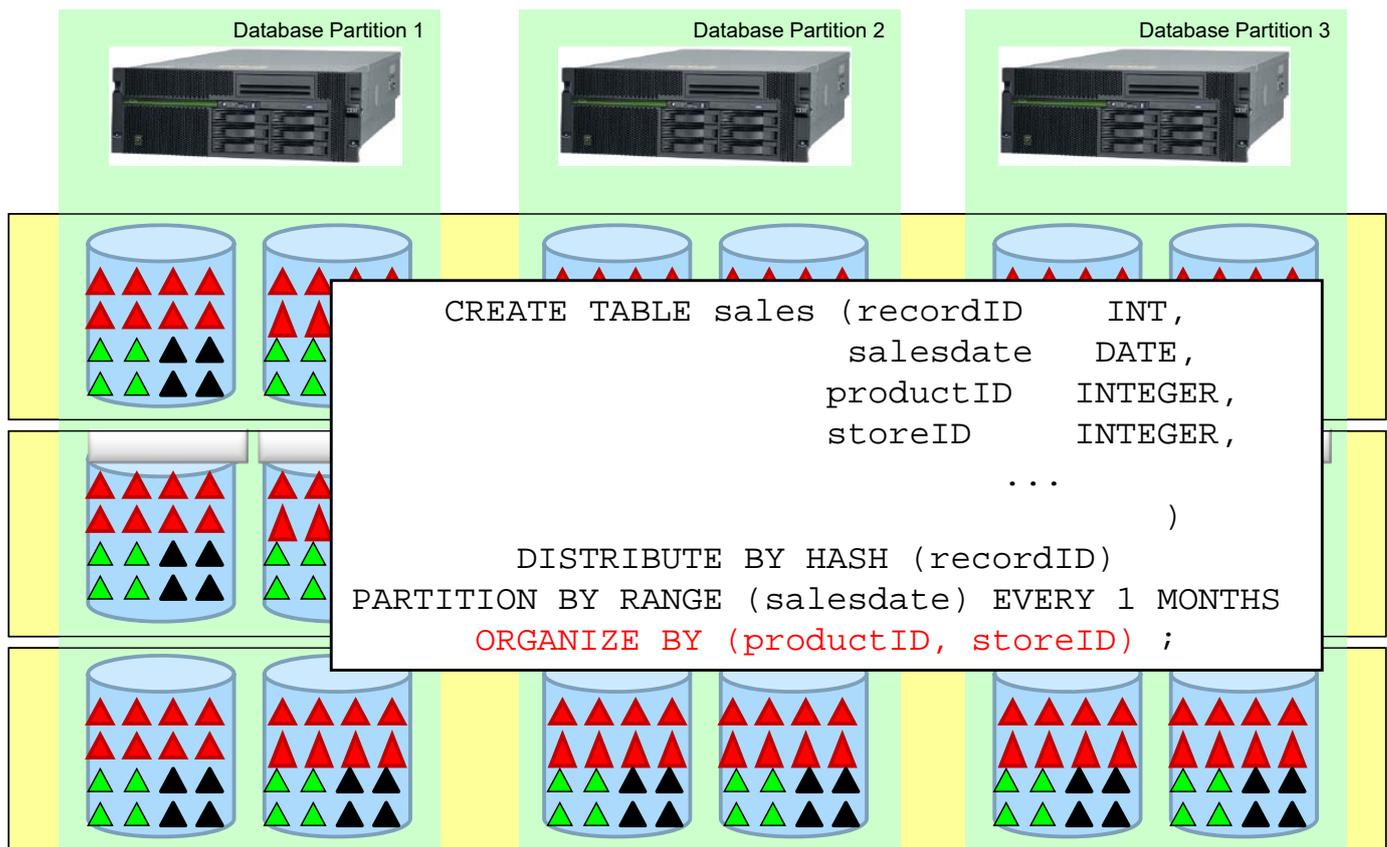
# Datenallokation in DB2

## ■ 3 –stufige Vorgehensweise

- DISTRIBUTE BY HASH - Tabellen-Partitionierung zwischen Knoten
- PARTITION BY RANGE – partitionsinterne Fragmentierung von Tabellen
- ORGANIZE BY DIMENSIONS – Clustering innerhalb von Fragmenten (mehrdimensionales Clustering, MDC)

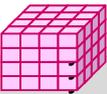


# DB2 Multi-Dimensional Clustering



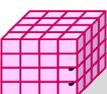
# 5. Performance-Techniken

- Einleitung
- Indexstrukturen
  - ein- vs. mehrdimensionale Indexstrukturen
  - Bitlisten-Indexstrukturen
- Column Stores
  - Datenkompression
- Datenpartitionierung
  - vertikale vs. horizontale Fragmentierung
  - mehrdimensionale, hierarchische Fragmentierung
- Materialisierte Sichten
  - Verwendung
  - Auswahl



## Materialisierte Sichten

- Unterstützung in kommerziellen DBS: Oracle, DB2, SQL Server
  - materialized views, summary tables, ...
- explizite Speicherung von Anfrageergebnissen, z.B. Aggregationen, zur Beschleunigung von Anfragen
- sehr effektive Optimierung für Data Warehousing
  - häufig ähnliche Anfragen (Star Queries)
  - Lokalität bei Auswertungen
  - relativ stabiler Datenbestand
- Realisierungs-Aspekte
  - Verwendung von materialisierten Sichten für Anfragen (Query-Umformulierung, query rewrite)
  - Auswahl der zu materialisierenden Sichten: statisch vs. dynamisch (Caching von Anfrageergebnissen)
  - Aktualisierung materialisierter Sichten



# Verwendung materialisierter Sichten

- Einsatz von materialisierter Sichten transparent für den Benutzer
  - DBS muss während Anfrageoptimierung relevante materialisierte Sichten automatisch erkennen und verwenden können (Anfrageumstrukturierung)
  - umgeformte Anfrage muss äquivalent zur ursprünglichen sein (dasselbe Ergebnis liefern)

## ■ Beispiel

### Anfrage Q

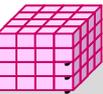
```
select sum (v.GBetrag) AS Umsatz
from VERKAUF v, PRODUKT p, ZEIT z
where v.Tag = z.Tag and v.P_Id = p.P_Id
and z.Monat = "April" and
p.Kategorie = "Smartphone"
```

### mat. Sicht M1

```
create materialized view M1 (K, M, S, A) AS
select p.Kategorie, z.Monat,
       SUM (GBetrag), SUM (Anzahl)
from VERKAUF v, PRODUKT p, ZEIT z
where v.Tag = z.Tag and v.P_Id = p.P_Id
group by cube (p.Kategorie, z.Monat)
```

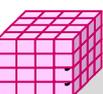
### modifizierte Anfrage Q'

```
select S AS Umsatz
from M1
where M='April' and K="Smartphone"
```



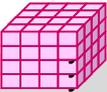
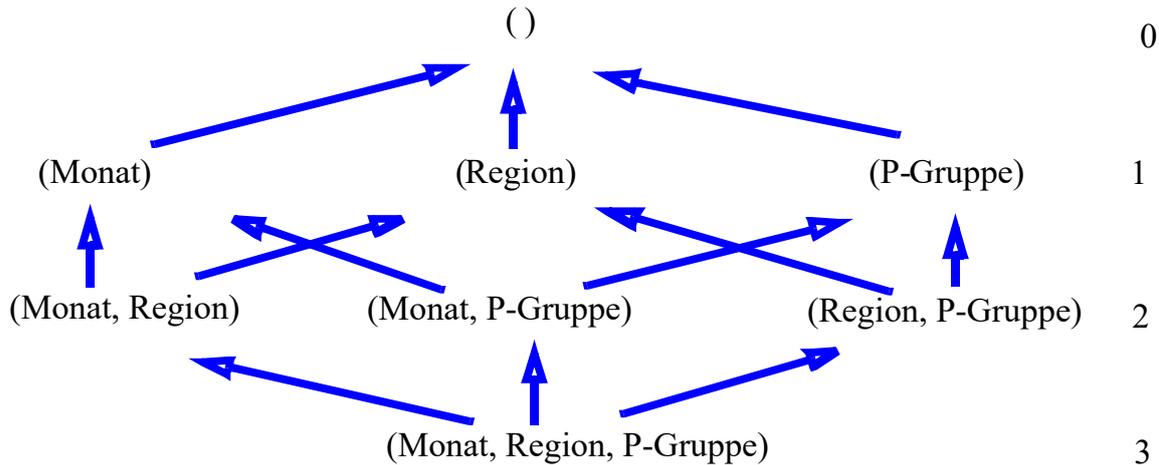
# Auswahl von materialisierter Sichten

- Optimierungs-Tradeoff:
  - Nutzen für Anfragen vs. erhöhte Speicherungs-/Aktualisierungskosten
- statische Bestimmung durch DBA oder Tool
  - keine Berücksichtigung aktueller Anfragen
  - keine Änderung bis zur nächsten Warehouse-Aktualisierung
- dynamische Auswahl: Caching von Anfrageergebnissen
  - Nutzung von Lokalität bei Ad-Hoc-Anfragen
  - günstig bei interaktiven Anfragen, die aufeinander aufbauen (z.B. Rollup)
  - komplexe Verwaltung, z.B. Verdrängungsentscheidungen für variabel große Ergebnismengen in Abhängigkeit von Referenzierungshäufigkeit, Kosten der Neuberechnung etc.



# Statische Auswahl materialisierter Sichten

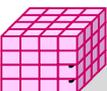
- Auswahl vorzuberechnender Aggregationen des Aggregationsgitters
  - *Aggregationsgitter*: azyklischer Abhängigkeitsgraph, der anzeigt, für welche Kombinationen aus Gruppierungsattributen sich Aggregierungsfunktionen (SUM, COUNT, MAX, ...) direkt oder indirekt aus anderen ableiten lassen
- vollständige Materialisierung aller Kombinationen i.a. nicht möglich
  - #Gruppierungskombinationen wächst exponentiell mit Anzahl von Gruppierungsattributen  $n$
  - möglichst optimale Teilmenge zu bestimmen



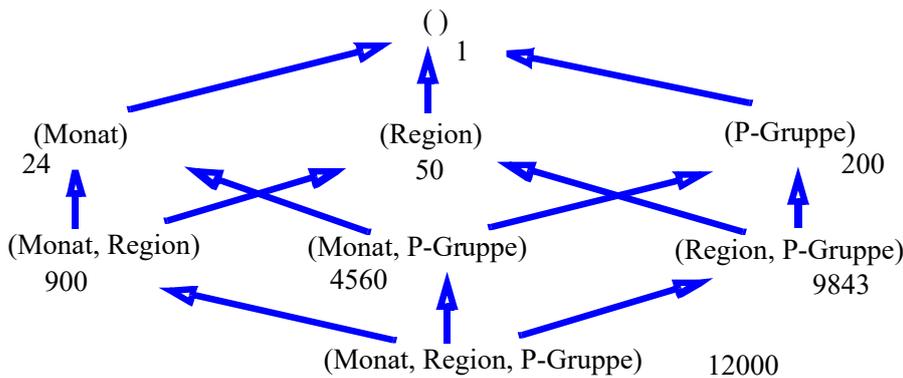
## Statische Auswahlheuristik\*

- Annahmen
  - gleiche Nutzungswahrscheinlichkeit pro Cuboid (Kombination von Dimensionsattributen)
  - Aufwand sei proportional zu Anzahl zu berechnender Sätze/Aggregate
- Heuristik für vorgegebenes Limit für Speicheraufwand
  - pro Kombination von Dimensionsattributen wird Summe der Einsparungen berechnet, die sie für andere nicht materialisierte Kombinationen bewirkt
  - in jedem Schritt wird die Kombination ausgewählt, die die größte Summe an Einsparungen zulässt, solange der maximal zugelassene Speicheraufwand nicht überschritten ist

\*Harinarayan/Rajaraman/Ullman, *Implementing Data Cubes Efficiently*. Proc. Sigmod 1996



# Statische Auswahlheuristik: Beispiel



Limit sei 50% Zusatzaufwand  
(bezogen auf Kardinalität der Detaildaten)

- Vollauswertung erfordert  $12.000 + 15.578 = 27.578$  Sätze (+ 130%)
- Beschränkung erforderlich

## ■ Schritt 1: maximale Einsparung für *Monat/Region*

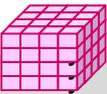
- nur 900 statt 12.000 Werte auszuwerten
- Nutzung für 4 Knoten (Einsparung  $4 * 11.100$ ): (Monat, Region), (Monat), (Region), ()

## ■ Schritt 2:

- *Monat*: zusätzliche Einsparung  $2 * (900 - 24) = 1752$
- *Monat, P-Gruppe*: zusätzliche Einsparung  $2 * (12.000 - 4.560) = 14.880$
- resultierender Overhead:  $900 + 4560 = 5460$

## ■ Schritt 3:

- *P-Gruppe*: zusätzliche Einsparung  $(4.560 - 200) + (900 - 200) = 5.060$



# Zusammenfassung

## ■ mehrdimensionale vs. 1-dimensionale Indexstrukturen

## ■ Bitlisten-Indizes

- effizient kombinierbar für mehrdimensionale Auswertungen / Star-Joins
- Bereichs- und Intervallkodierung für Bereichsanfragen
- kodierte Bitlisten-Indizes für große Dimensionen

## ■ Partitionierung

- vertikale oder horizontale Zerlegung von Relationen zur Reduzierung des Arbeitsaufwandes und Unterstützung von Parallelverarbeitung
- mehrdimens. horizontale Fragmentierung ermöglicht umfassende Einsparungen

## ■ materialisierte Sichten

- große Performance-Vorteile durch Vorberechnung von Anfragen/Aggregationen
- dynamische vs. statische Auswahl an materialisierten Sichten

## ■ Column Stores

- hohe I/O-Einsparungen für OLAP, insbesondere mit komprimierten Datenwerten
- effiziente Aggregationsmöglichkeiten
- in Kombination mit In-Memory-DWH verlieren andere Optimierungen an Bedeutung

