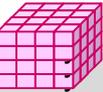


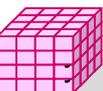
5. Performance-Techniken

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



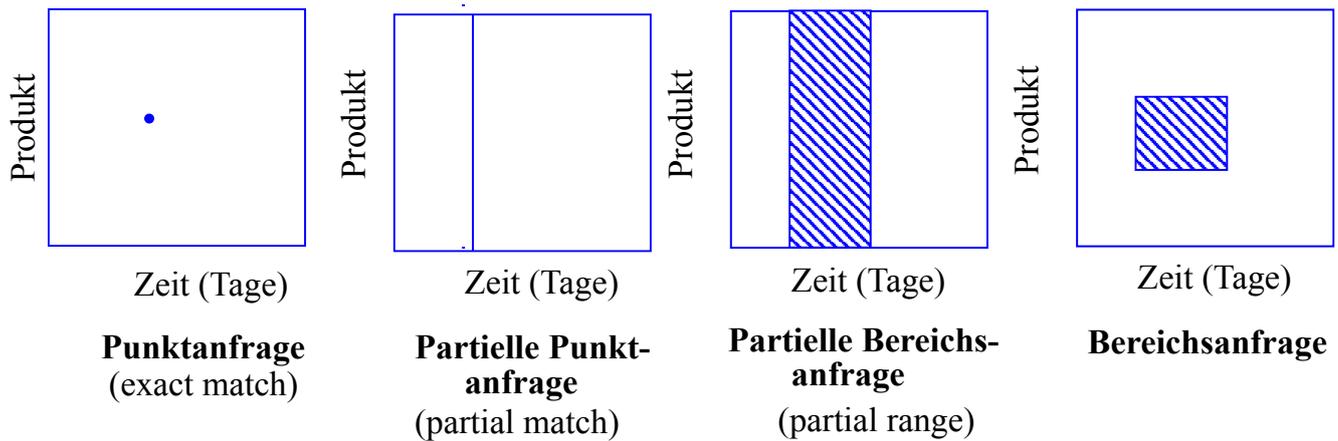
Einleitung

- hohe Leistungsanforderungen
 - sehr große Datenmengen, vor allem Faktentabelle
 - kurze Antwortzeiten für viele Benutzer
 - mehrdimensionale Auswahlbedingungen, Gruppierung, Aggregationen, Sortierung ...
 - periodische Aktualisierung mit sehr vielen Änderungen (ETL, Aktualisierung der DW-Tabellen, Hilfsstrukturen, Cubes)
- 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 Performance-Techniken unter spezieller Nutzung von DW-Charakteristika
 - Indexstrukturen (1-dimensional, mehrdimensional, Bit-Indizes)
 - materialisierte Sichten bzw. vorberechnete Aggregationen
 - Partitionierung der Daten (Einschränkung der zu bearbeitenden Daten) und Parallelverarbeitung
 - Column Store

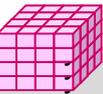


Mehrdimensionale Anfragearten

■ Punkt- und Bereichsanfragen (exakt vs. partiell)



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



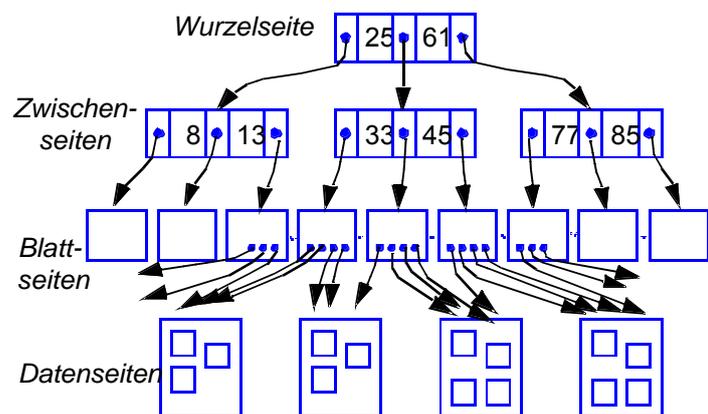
Indexstrukturen

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

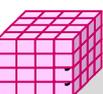
- zusätzlicher Speicherbedarf und Änderungsaufwand

■ 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 direktem und sortiert-sequentiellen Zugriffen

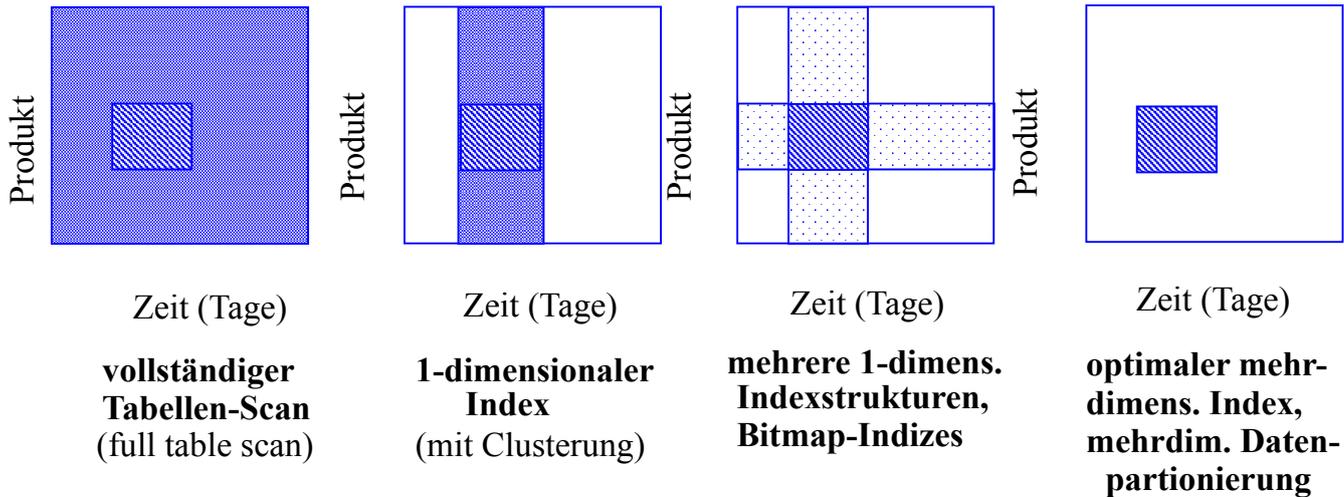


Clustered Index



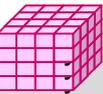
Indexunterstützung für mehrdimens. Anfragen

■ Eingrenzung des Datenraumes



■ Bsp. mehrdimensionaler Indexstrukturen

- Grid File
- R-Baum, ...



Bitlisten-Indizes

■ herkömmliche Indexstrukturen ungeeignet für Suchbedingungen geringer Selektivität

- z.B. für Attribute (Dimensionen) mit nur wenigen Werten (Geschlecht, Farbe, Jahr ...)
- pro Attributwert sehr lange Verweislisten (TID-Listen) auf zugehörige Sätze
- nahezu alle Datenseiten zu lesen

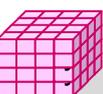
■ Standard-Bitlisten-Index (**Bitmap Index**):

- 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)

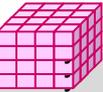
■ Vorteile

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

■ Beispielanfrage

Select ... WHERE A1=C1 AND A2=C2 AND A3=C3
100 Millionen Sätze; pro Teilbedingung Selektivität 1%

- Erweiterungen erforderlich für Bereichsanfragen, große Wertebereiche, hierarchische Dimensionen

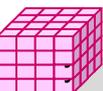
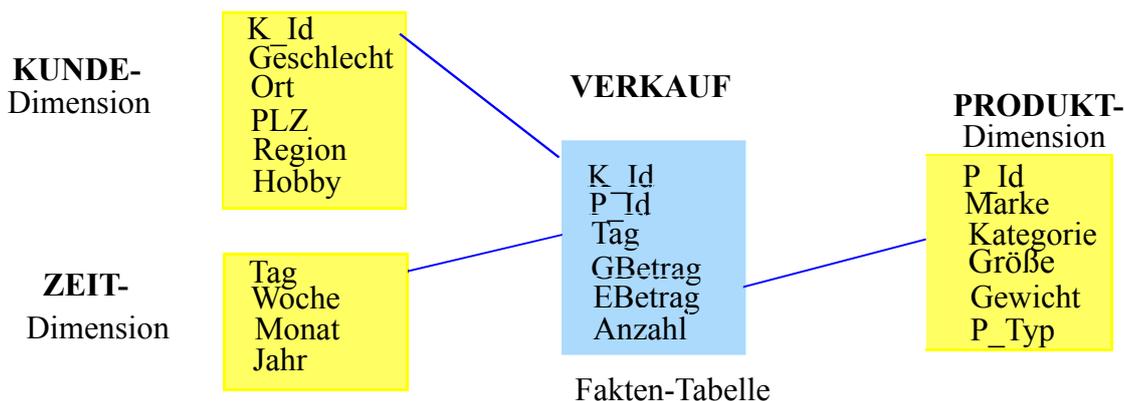


Bitlisten-Join-Index

- 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



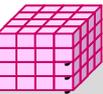
Bitlisten-Join-Index (2)

- Beispielanfrage: Video-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 = "Video"
      and k.Geschlecht = "M"
```

- Nutzung von 3 Bitlisten-Join-Indizes

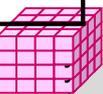
Monat	Kategorie	Geschlecht
Januar 0001100010001 ...	TV 1001000000010 ...	M 1010010001001 ...
Febr 0000011100000 ...	DVD 0100000001000 ...	W 0111101110110 ...
März 1000000000010 ...	Video 0010100010101 ...	
April 0110000001100 ...	Stereo 0000011100000 ...	
...	...	



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

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											
123	März	0	0	1	0	0	0	0	...	0											
124	Januar	1	0	0	0	0	0	0	...	0											
125	März	0	0	1	0	0	0	0	...	0											
126	Februar	0	1	0	0	0	0	0	...	0											
127	Dezember	0	0	0	0	0	0	0	...	1											
128	April	0	0	0	1	0	0	0	...	0											
...																			



Bereichskodierte Bitlisten-Indizes (2)

■ Bereichsanfragen: Beispiele

– Bereich $A < x \leq E$: $\text{not } B(A) \text{ AND } B(E)$

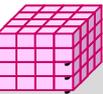
– Bereich $x \leq E$: $B(E)$

– Bereich $x > A$: $\text{not } B(A)$

■ Punktanfrage (Gleichheitsbedingung) erfordert (meist) Lesen von 2 Bitlisten vs. 1 Bitliste bei Standard-Bitlisten-Index

nur Juni: $B(\text{Juni}) \text{ AND NOT } B(\text{Mai})$

■ 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: $I2 \text{ AND NOT } I3$

■ 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							
125	März	0	0	1	0	0	0	0	...	0							
126	Februar	0	1	0	0	0	0	0	...	0							
127	Dezember	0	0	0	0	0	0	0	...	1							
128	April	0	0	0	1	0	0	0	...	0							
...															



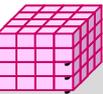
Kodierte Bitlisten-Indizes

- 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 => nur noch $\log_2 k$ Bitlisten
 - hohe Einsparungen bei großen k (z.B. Kunden, Produkte) **k=1 Million - >20 Bitlisten**

		Kodierung	F_1	F_0	
Blau	0001100010001 ...	Blau	0	0	2 Bitvektoren für 4 Farben F_1 0 0 1 0 0 1 ... F_0 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: not F1 and not F0
- bei mehreren Bedingungen wird auch Auswertungsaufwand meist reduziert
- Blau oder Rot: NOT F1



Kodierungsvarianten

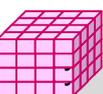
■ Mehrkomponenten-Bit-Indizes

- Zerlegung von Attributwerten in mehrere Komponenten und separate Kodierung
- Wahl der Komponenten erlaubt Kompromiss zwischen Speicheraufwand (# Bitlisten) und Zugriffsaufwand
- Bsp.: Produkt-Nr (0..999) = $x * 100 + y * 10 + z$ mit x,y,z aus 0..9

■ Hierarchische Dimensionskodierung

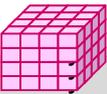
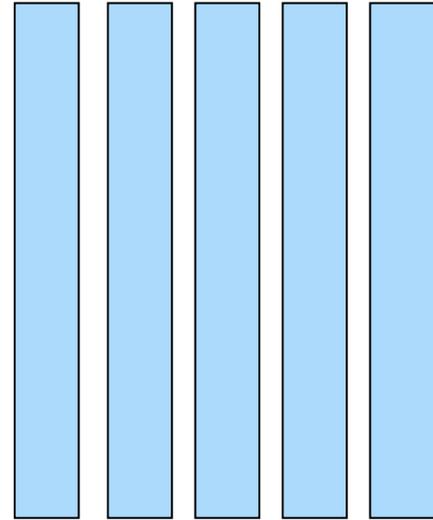
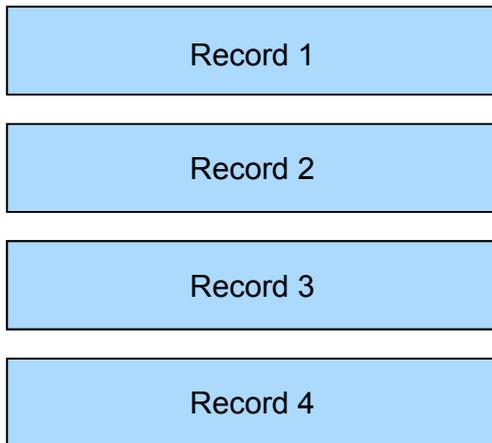
- Vermeidung separater Bitlisten für jede Dimensionsebene
-> hierarchische Kodierung mit 1 Bitlisten-Index pro Dimension
- Verwendung konkatenierter Schlüssel-IDs und separate Kodierung
- Beispiel: Kodierung einer Produkthierarchie mit ca. 50000 Produkten

	<i>BEREICH</i>	<i>FAMILY</i>	<i>GRUPPE</i>	<i>PNR</i>	Gesamt
#Elemente Gesamt	8	96	1536	49,152	49152
#Elemente pro Vorgänger	8	12	16	32	
#Bits für Kodierung (\log_2)	3	4	4	5	16
Beispiel-Bitmuster	bbb	ffff	gggg	ppppp	bbbffffggggppppp



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 ...

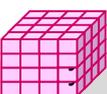


Vorläufer (Sybase IQ): Projektions-Index

- separate Speicherung der Attributwerte ausgewählter Attribute (vertikale Partitionierung)

	PNR	ANR	W-ORT	GEHALT	Projektions-Index GEHALT
1	12345	K02	L	45000	45000
2	23456	K02	M	51000	51000
3	34567	K03	L	48000	48000
4	45678	K02	M	55000	55000
5	56789	K03	F	65000	65000
6	67890	K12	L	50000	50000

- starke E/A-Einsparungen verglichen mit Zugriff auf vollständige Sätze
- Bsp.: Berechnung von Durchschnittsgehalt, Umsatzsumme ...
- effektive Einsatzmöglichkeit in Kombination mit Bitlisten-Index-Auswertungen



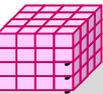
Vorteile / Nachteile

■ 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)

- noch relativ neue Technologie
- 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

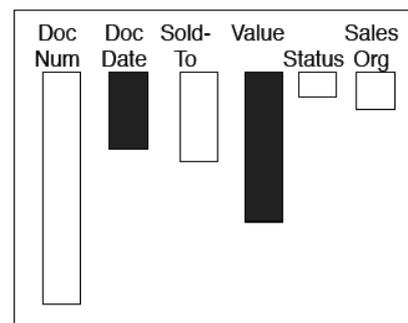
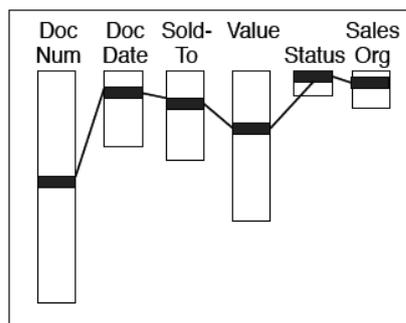


Anfragen – Column vs. Row Store *

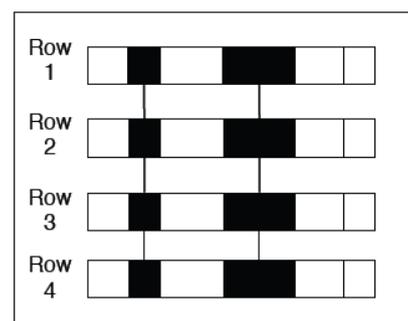
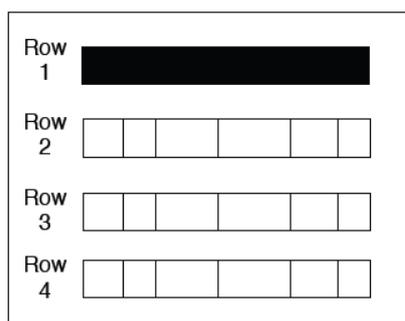
```
SELECT *
FROM Sales Orders
WHERE Document Number = '95779216'
```

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

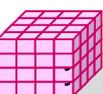
Column Store



Row Store

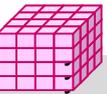


* Quelle: Hasso Plattner: Enterprise Applications – OLTP and OLAP – Share One Database Architecture



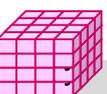
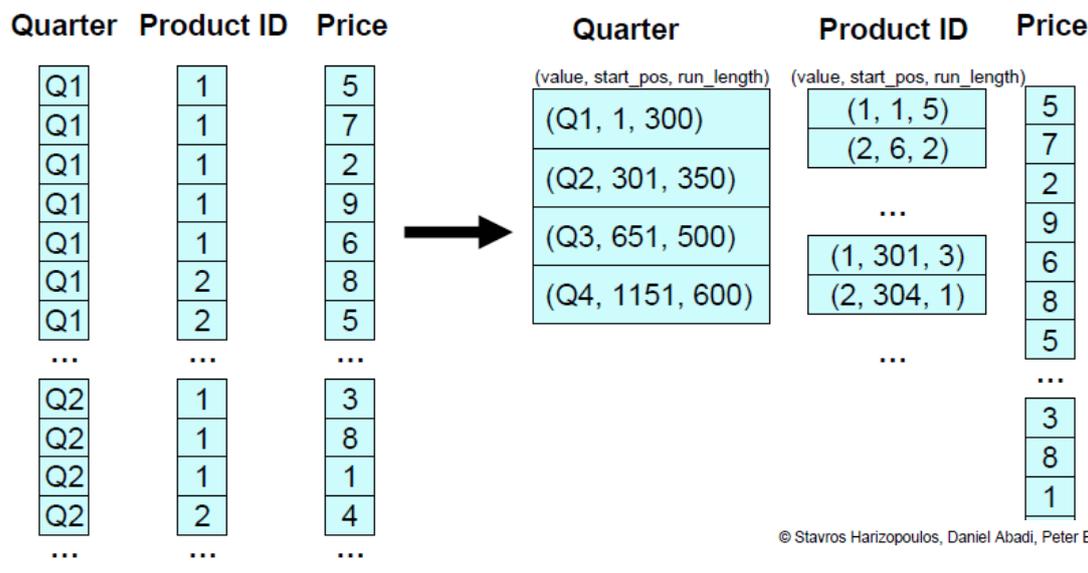
Datenkompression

- 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
- Zahlreiche Varianten
 - Run Length Encoding (RLE)
 - Bit Vector Encoding
 - Delta Coding
 - Wörterbuch-Kodierung
 - etc.
- pro Spalte kann das beste Kodierungsverfahren gewählt werden



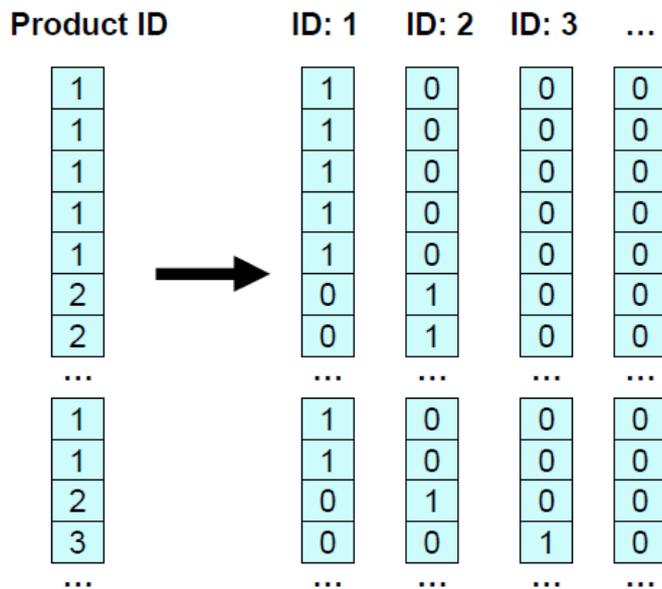
Run Length Encoding (RLE) / Lauflängenkodierung

- 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

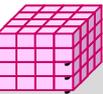


Bit Vector Encoding

- 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

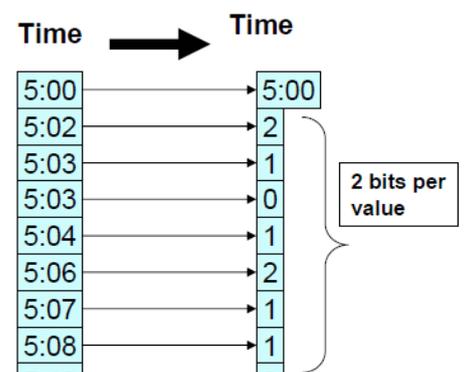


© Stavros Harizopoulos, Daniel Abadi, Peter Boncz (2009)

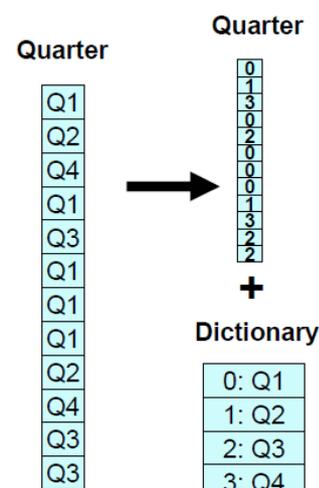


Weitere Kompressionstechniken

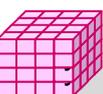
- Delta / Difference Coding
 - Speicherung der Differenz zu Vorgängerwerten
 - Oft kompakter, v.a. bei Sortierung
- Wörterbuch-Kodierung
 - Erfordert keine Sortierung
 - Günstig v.a. bei wenigen und langen Attributwerten
 - Auswertung auf komprimierten Daten möglich
 - Lookup zur Dekomprimierung



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

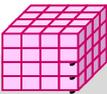
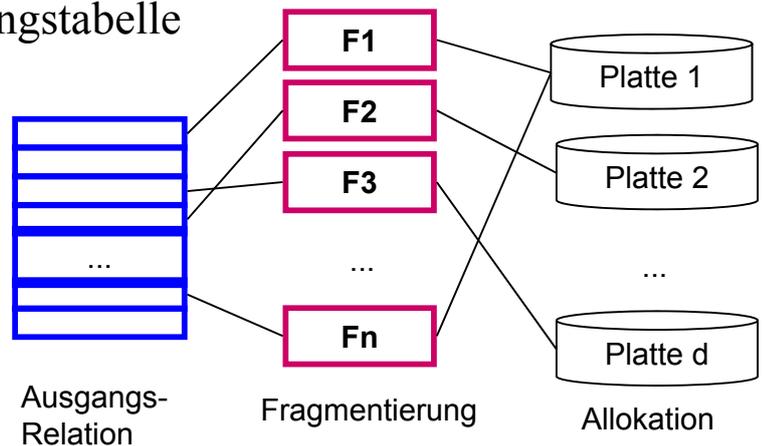


© Stavros Harizopoulos, Daniel Abadi, Peter Boncz (2009)



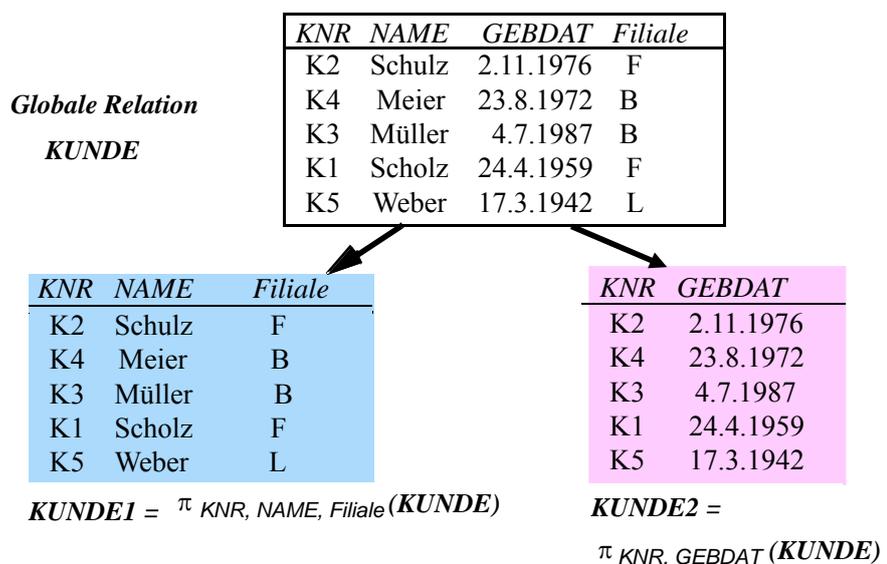
Datenpartitionierung

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

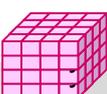


Vertikale Fragmentierung

- Spaltenweise Aufteilung von Relationen
- Definition der Fragmentierung durch Projektion
- Vollständigkeit:
 - jedes Attribut in wenigstens 1 Fragment enthalten

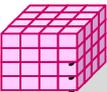
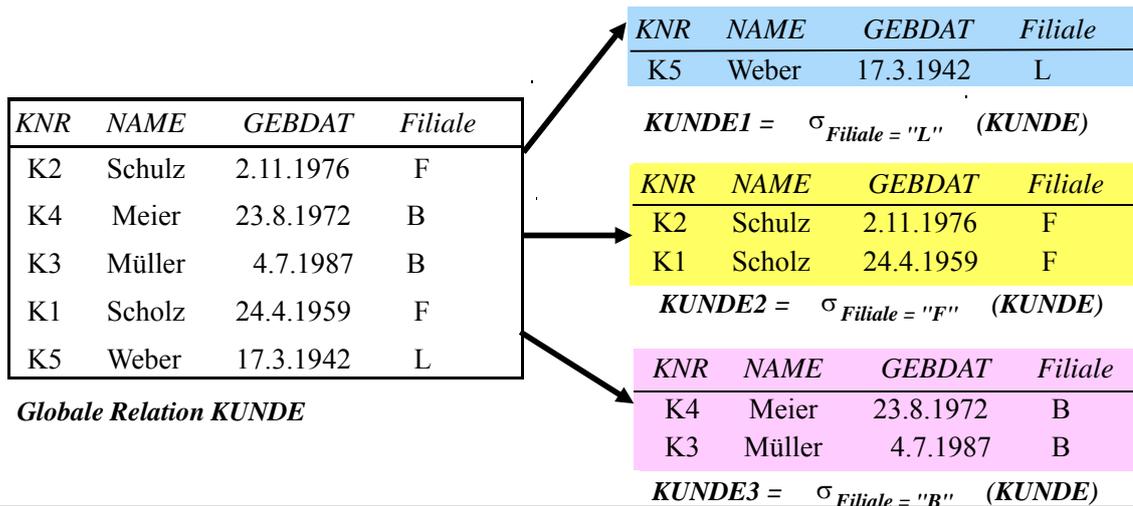


- Verlustfreie Zerlegung:
 - Primärschlüssel i.a. in jedem Fragment enthalten
 - JOIN-Operation zur Rekonstruktion des gesamten Tupels
- Arbeitersparnis durch Auslagern selten benötigter Attribute in eigene Fragmente



Horizontale Fragmentierung

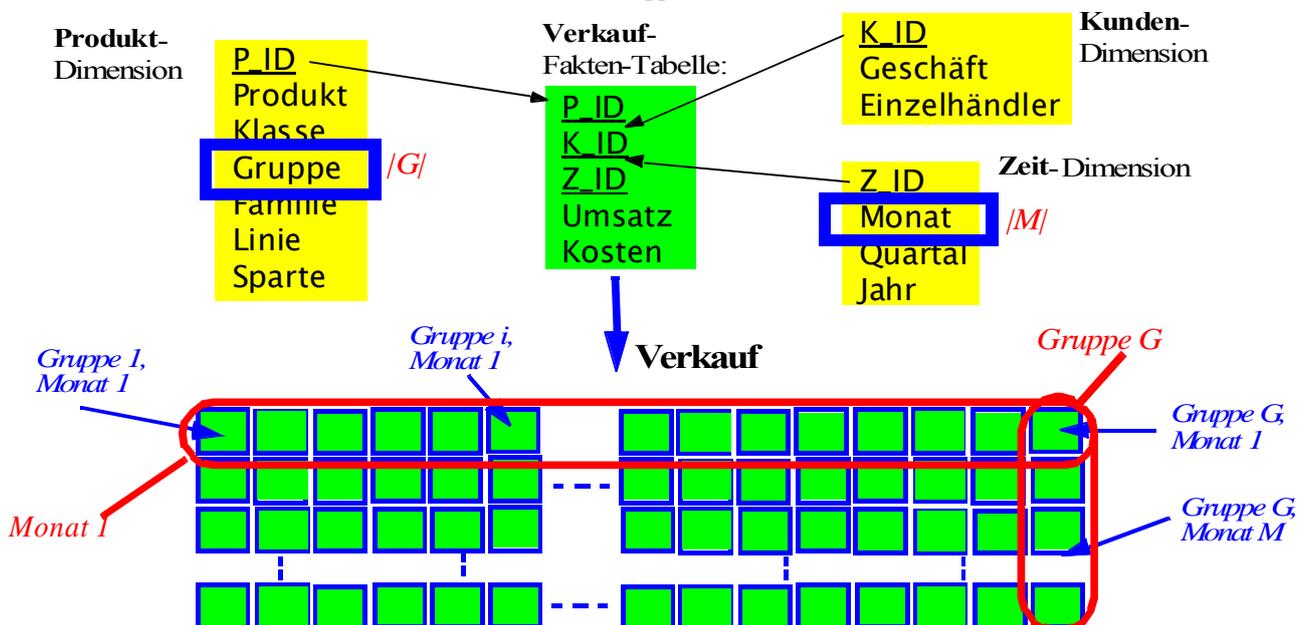
- Zeilenweise Aufteilung von Relationen
- Definition der Fragmentierung durch Selektionsprädikate P_i auf der Relation:
 - $R_i := \sigma_{P_i}(R)$ ($1 \leq i \leq n$)
 - Vollständigkeit: jedes Tupel ist einem Fragment eindeutig zugeordnet
 - Fragmente sind disjunkt: $R_i \cap R_j = \{\}$ ($i \neq j$)
 - 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



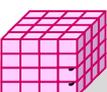
Multi-dimensionale, hierarchische Fragmentierung (MDHF)*

- Horizontale Bereichsfragmentierung auf *mehreren* Attributen (reihenfolgeunabhängig)
 - 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**

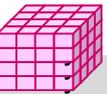
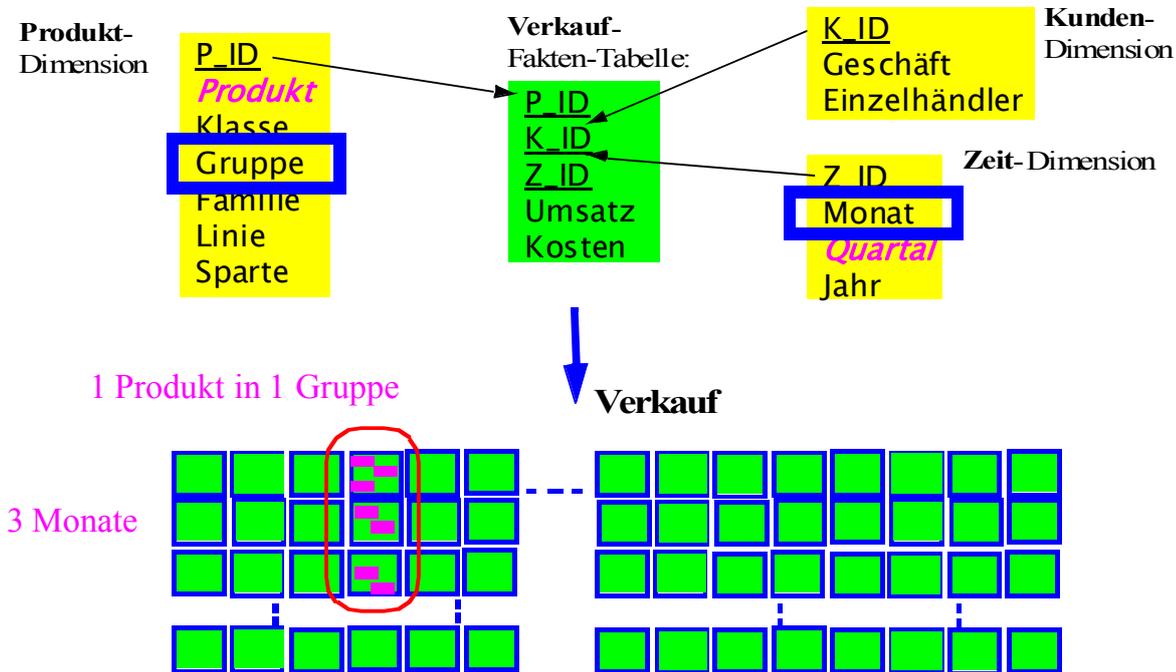


* Stöhr, T., Märtens, H., Rahm, E.: *Multi-Dimensional Database Allocation for Parallel Data Warehouses* Proc. VLDB, 2000,



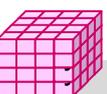
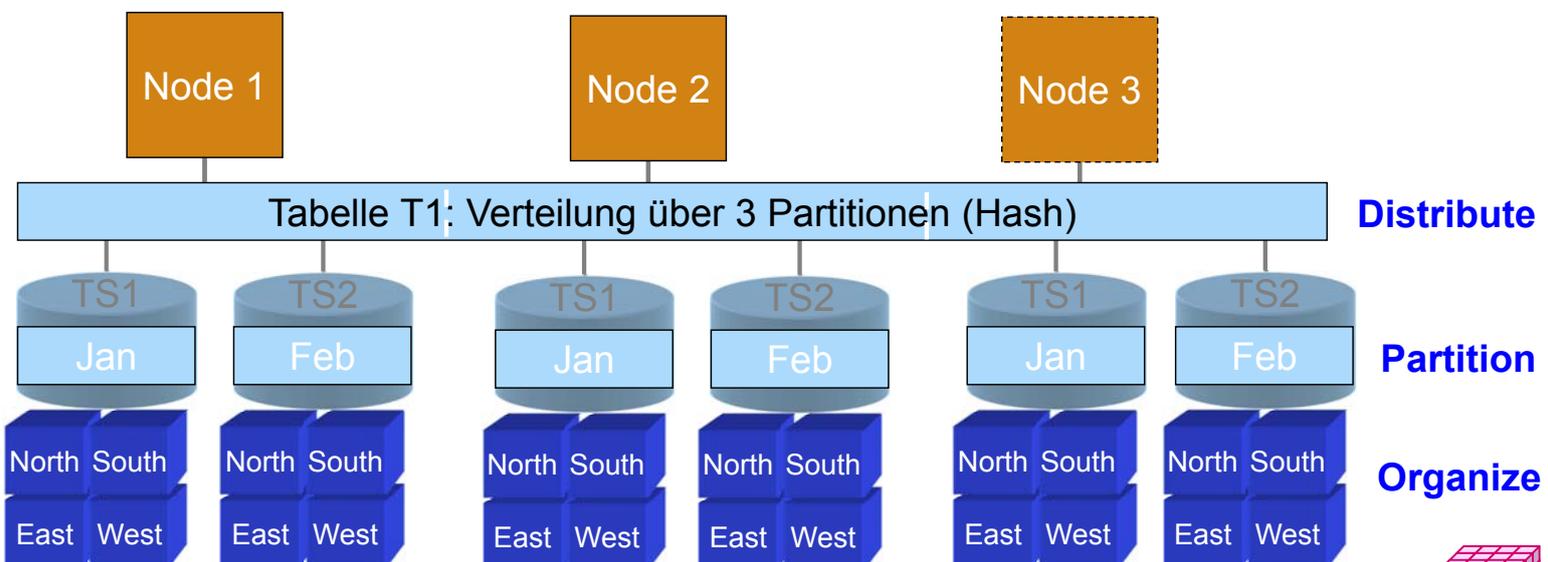
Sternschema-Anfrage unter MDHF

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

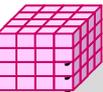
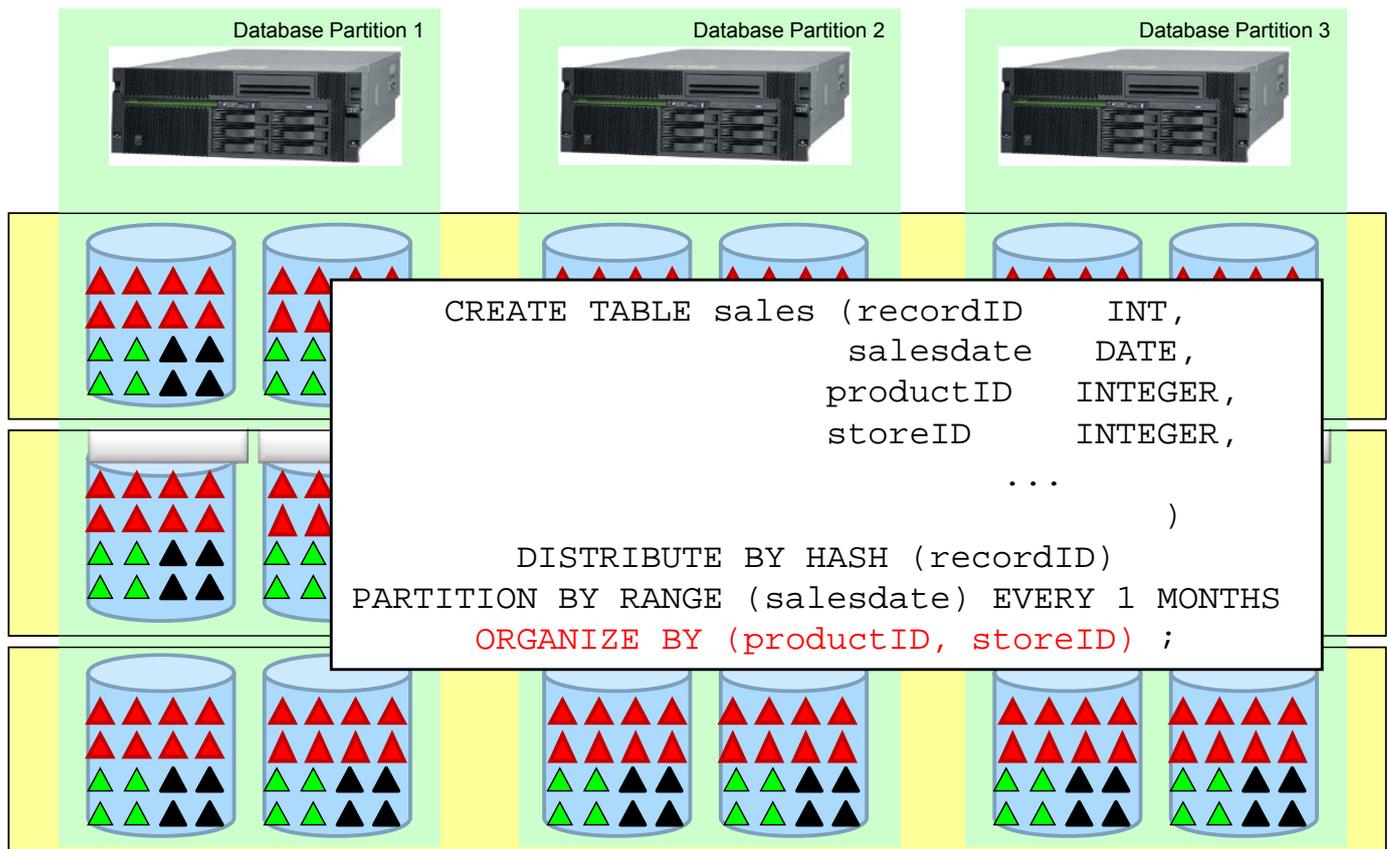


Datenallokation in DB2

- 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)

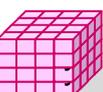


DB2 Multi-Dimensional Clustering



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

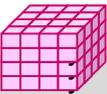
```
select sum (v.GBetrag)
from VERKAUF v, PRODUKT p, ZEIT z
where v.Tag = z.Tag and v.P_Id = p.P_Id
and z.Monat = "Jun04" and
p.Kategorie = "Video"
```

modifizierte Anfrage Q'

```
select
```

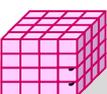
mat. Sicht M1

```
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)
```



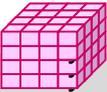
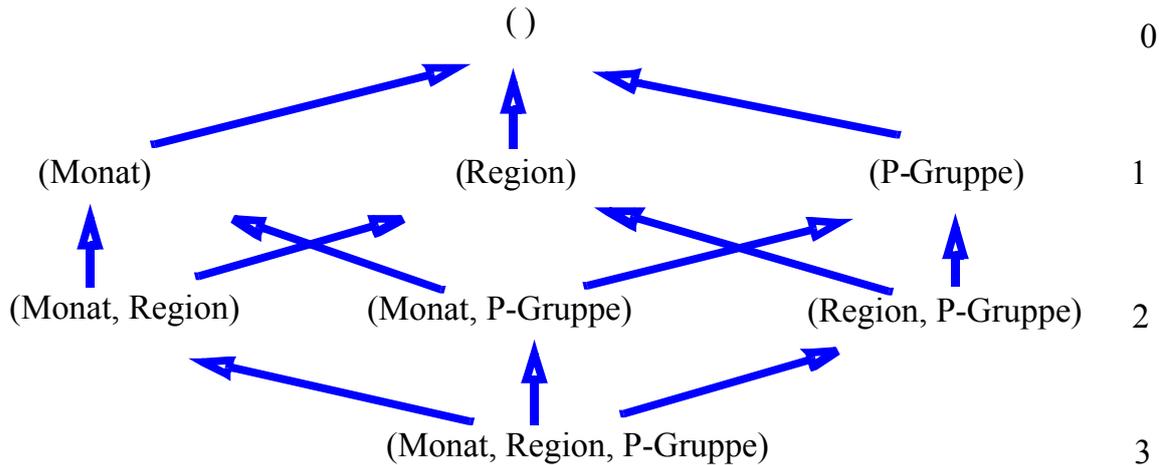
Auswahl von materialisierter Sichten

- Optimierungs-Tradeoff: Nutzen für Anfragen vs. erhöhten Speicherungs- und Aktualisierungskosten
- statische Bestimmung durch DBA oder Tool:
 - keine Berücksichtigung aktueller Anfragen
 - keine Änderung bis zur nächsten Warehouse-Aktualisierung
- dynamische Auswahl: Caching von Anfrageergebnissen (semantisches Caching)
 - Nutzung von Lokalität bei Ad-Hoc-Anfragen
 - günstig bei interaktiven Anfragen, die aufeinander aufbauen (z.B. Rollup)
- Komplexe Verdrängungsentscheidung für variabel große Ergebnismengen unter Berücksichtigung von
 - Zeit des letzten Zugriffs, Referenzierungshäufigkeit
 - Größe der materialisierten Sicht
 - Kosten, die Neuberechnung verursachen würde
 - Anzahl der Anfragen, die mit Sicht bedient wurden



Statische Auswahl materialisierter Sichten

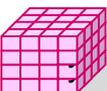
- Auswahl vorzuberechnender Aggregationen des Aggregationsgitters
 - *Aggregationsgitter*: azyklischer Abhängigkeitsgraph, der anzeigt, für welche Kombinationen aus Gruppierungsattributen sich Aggregierungsfunktionen (SUM, COUNT, MAX, ...) direkt oder indirekt aus anderen ableiten lassen
- vollständige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



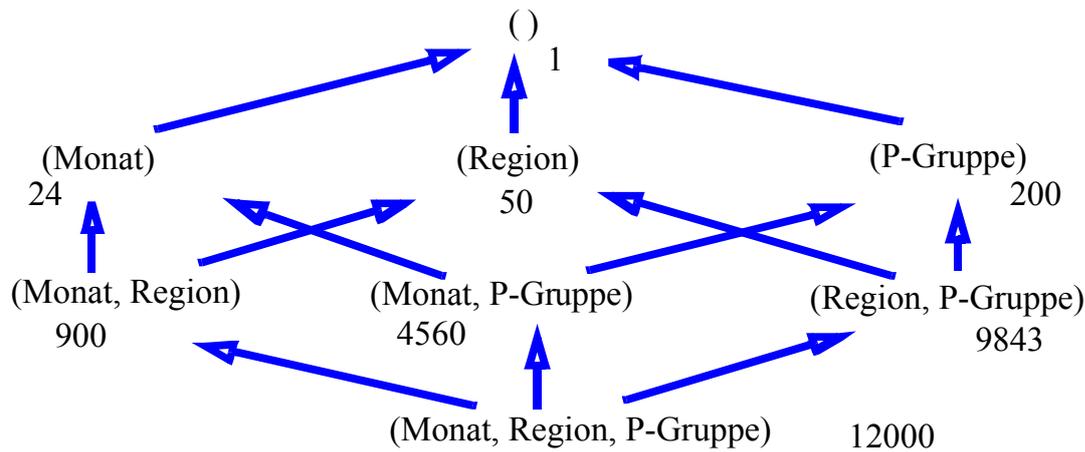
Statische Auswahlheuristik*

- 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

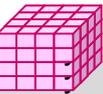
*Harinarayan/Rajaraman/Ullman, *Implementing Data Cubes Efficiently*. Proc. Sigmod 1996



Statische Auswahlheuristik: Beispiel



- Limit sei 75% Zusatzaufwand (bezogen auf Kardinalität der Detaildaten)
 - Vollausswertung erfordert $12.000 + 15.578 = 27.578$ Sätze (+ 130%) -> Beschränkung
- 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:



Zusammenfassung

- mehrdimensionale vs. 1-dimensionale Indexstrukturen
- Bit-Indizes
 - effizient kombinierbar für mehrdimensionale Auswertungen / Star-Joins
 - Bereichs- und Intervallkodierung für Bereichsanfragen
 - Kodierung für höhere Kardinalität: logarithmisch, Mehr-Komponenten-Kodierung, hierarchische Kodierung
- Partitionierung
 - vertikale oder horizontale Zerlegung von Relationen zur Reduzierung des Arbeitsaufwandes und Unterstützung von Parallelverarbeitung
 - mehrdimens. horizontale Fragmentierung: ähnliche Vorteile wie mehrdim. Indexstrukturen
- 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



Übungsfragen

■ Bitlisten-Indizes

- Bestimmen Sie für die Beispiele bereichskodierter bzw. intervallkodierter Bitlisten-Indizes auf Folien 5-10 bzw. 5-12 wie folgende Anfrageprädikate evaluiert werden können:
- Monat="Januar"
- Monat IN („Mai“, „Juni“, „Juli“)

■ Auswahl materialisierter Sichten

- Welche drei Materialisierungen bringen für die im Aggregationsgitter angegebenen Kardinalitäten den größten Einspareffekt gemäß dem Verfahren von Harinarayan et al. ?
- Welcher Zusatzaufwand wird durch sie gegenüber der Speicherung von (A,B,C) eingeführt?

