

# Data-Warehouse-Technologien

Prof. Dr.-Ing. Kai-Uwe Sattler<sup>1</sup>    Prof. Dr. Gunter Saake<sup>2</sup>  
Dr. Veit Köppen<sup>2</sup>

<sup>1</sup>TU Ilmenau  
FG Datenbanken & Informationssysteme

<sup>2</sup>Universität Magdeburg  
Institut für Technische und Betriebliche Informationssysteme

Letzte Änderung: 18.10.2019

## Teil VI

# Speicherstrukturen für Data Warehouse

# Speicherstrukturen für Data Warehouse

- 1 Relationale Speicherung
- 2 Multidimensionale Speicherung
- 3 Speicherungsvarianten
- 4 Spaltenorientierte Speicherung

# Relationale Speicherung – ROLAP

- Umsetzung von Star- bzw. Snowflake-Schema auf Relationen
- Verbreitetste Form der Speicherung von DW-Tabellen (Details: siehe VL „Datenbank-Implementierungstechniken“)
- Besonderheiten
  - ▶ Sehr große Faktentabellen → Beschleunigung der Zugriffe durch Partitionierung
  - ▶ Multidimensionale Zugriffe → spezielle Cluster- und Indexstrukturen
  - ▶ Update-Charakteristik (Anhängen von Daten)

# Partitionierung

- Unabhängig von und ergänzend zu Indexverfahren:  
Aufteilung umfangreicher Relationen in kleinere Teilrelationen (sogenannte **Partitionen** oder **Fragmente**)
- Größe und Inhalt der Partitionen richtet sich nach Anfrage- und Aktualisierungscharakteristik
- Ursprünglich für verteilte Datenbanken um Lastverteilung auf mehreren Knoten zu unterstützen
- Partitionierung umfasst die **logische** Aufteilung von Relationen, die physische Verteilung ist Aufgabe der **Allokation**

# Horizontale Partitionierung

- Masterrelation  $R$  wird vollständig in mehrere paarweise disjunkte Teilrelationen  $R_1, \dots, R_n$  aufgeteilt:

$$R = R_1 \cup \dots \cup R_n; \quad R_i \cap R_j = \emptyset \text{ für } i \neq j$$

- Verschiedene Formen der Aufteilung:

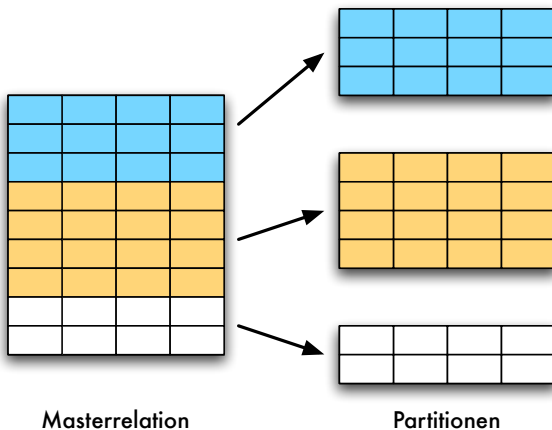
- ▶ **Range-Partitionierung:**

- ★ Jede Partition wird durch ein Selektionskriterium definiert  
 $R_i := \sigma_{\varphi}(R)$  mit  $\varphi$  Selektionsbedingung (range restriction)

- ▶ **Hash-Partitionierung:**

- ★ Hashfunktion (angewendet auf das ganze Tupel oder einzelne Attribute) bestimmt, zu welcher Partition ein Tupel gehört
- ★ Tupel mit gleichem Hashwert (oder Hashwerten in einem vorgegebenem Bereich) befinden sich in einer Partition

## Horizontale Partitionierung (2)



# Vertikale Partitionierung

- Verteilung einzelner Attribute (Spalten) auf Partitionen
- Partition entspricht einer Projektion auf die Masterrelation:

$$R_i := \pi_{attrlist}(R)$$

- Für Rekonstruierbarkeit der Masterrelation muss gemeinsames Attribut in jeweils zwei Partitionen existieren
  - ▶ I.d.R. ist Primärschlüssel in allen Partitionen enthalten



## Vertikale Partitionierung (2)

K	A	B	C	D
001				
002				
003				
004				
005				
006				

*Projektion*

K	A	B
001		
002		
003		
004		
005		
006		

K	C	D
001		
002		
003		
004		
005		
006		

# Partitionierung in Oracle

- Bereichspartitionierung

```
CREATE TABLE Verkauf (  
    Datum DATE NOT NULL,  
    ...)  
PARTITION BY RANGE (Datum) (  
PARTITION Verkauf2009  
    VALUES LESS THAN (to_date('2010-01-01')),  
PARTITION Verkauf2010  
    VALUES LESS THAN (to_date('2011-01-01')),  
PARTITION Verkauf2011  
    VALUES LESS THAN (to_date('2012-01-01')));
```

## Partitionierung in Oracle (2)

- Hash-Partitionierung

```
CREATE TABLE Verkauf (  
    ArtikelID INT NOT NULL,  
    FilialID INT NOT NULL,  
    ...)  
PARTITION BY HASH(ArtikelID, FilialID)  
PARTITIONS 5;
```

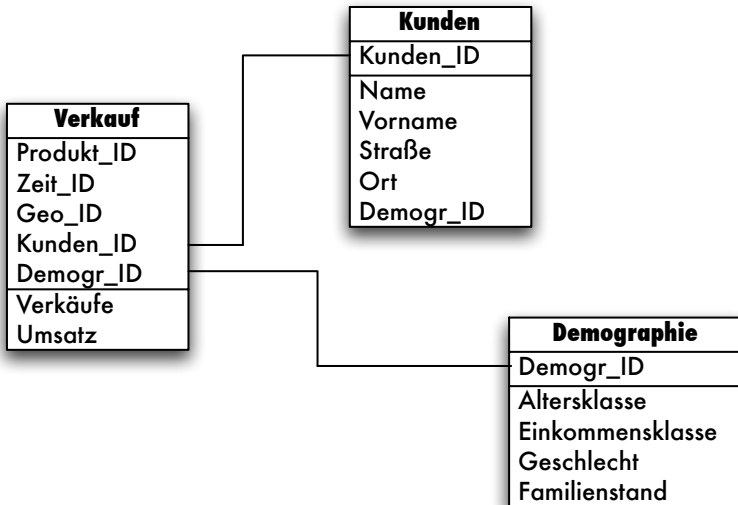
# Partitionierung in Data Warehouses

- Horizontale Partitionierung (insb. Range-Partitionierung) erlaubt z.B. große Faktentabellen in handlichere Teile zu zerlegen
  - ▶ Selektionsbedingungen für einzelne Partitionen sollten die in Anfragen häufig vorkommenden Bereichseinschränkungen berücksichtigen
- Vertikale Partitionierung erfordert i.d.R. teure Join-Operation zum Wiederausammensetzen der Tupel;
  - ▶ Kann zum Abspalten selten angefragter Attribute eingesetzt werden
  - ▶ Verkleinerung von Fakten- oder Dimensionstabellen, auf die häufig zugegriffen wird

## Partitionierung in Data Warehouses (2)

- Spezialfall vertikaler Partitionierung: **Mini-Dimensionen**
- Gelegentlich werden Dimensionstabellen riesig groß:  
z.B. Kundentabelle mit mehreren Millionen Datensätzen
  - ▶ Viele Attribute werden nie oder nur selten angefragt, da sie für Auswertungen uninteressant sind
- oder*
  - ▶ es gibt disjunkte Attributgruppen, die immer nur für verschiedene Anwendungen bzw. verschiedene Arten von Auswertungen benötigt werden
- Abtrennung von Attributen durch vertikale Partitionierung erlaubt dann eine deutliche Verkleinerung der einzelnen Dimensionstabellen

# Mini-Dimensionen



# Spezielle Tabellentypen in DB2

- Append-Mode-Tabellen
  - ▶ optimiert für **insert**-Operationen
  - ▶ Tupel werden am Ende angefügt, ohne Freispeicher auf Seiten zu berücksichtigen
- Bereichsgeclusterte Tabellen (range-clustered tables – RCT)
  - ▶ Für Sequenzdaten
- Multidimensional-geclusterte Tabellen (multidimensional clustering tables – MDC)
  - ▶ Speicherung in mehreren Dimensionen clusterweise

## Append-Mode Tabellen

- Optimierter Modus für Tabellen zum Hinzufügen von Daten
- Hinzufügen erfolgt am Ende → INSERT Optimierung
- Führt to mehreren Page-Loads zur Abfragezeit
- in DB2 per ALTER TABLE darf kein geclusterter Index assoziiert werden
- in Oracle beim Laden, z.B. Bulk-Loader-Option

```
ALTER TABLE Bestellung (  
    BestellNr int primary key, ...  
) APPEND ON
```



# Bereichsgeclusterte Tabellen

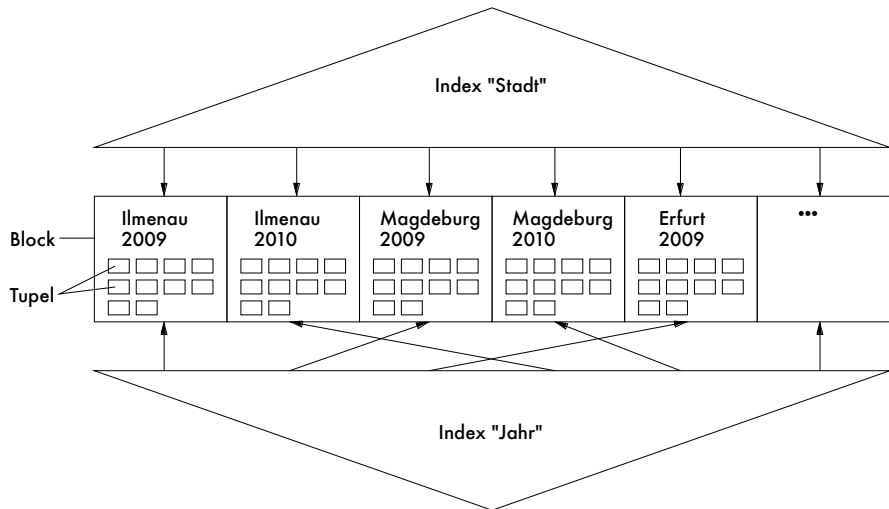
- Nutzung ein Sequenznummer (frei wählbares Attribut) als logisch Rowid zur Ermittlung der physischen Speicheradresse
- Vorab-Allokation des gesamten Speicherplatzes der Tabelle
- Einsortierung des Tupels über Sequenznummer
- Zugriff über Sequenznummer → kein zusätzlicher Index notwendig

```
CREATE TABLE Bestellung (  
    BestellNr int primary key, ...  
) ORGANIZE BY KEY SEQUENCE  
    (BestellNr starting from 1 ending at 10000)
```

# MDC-Tabellen

- Tabellen üblicherweise max. nach einem Index geclustered
- Scan über anderen Index im Worst Case: 1 Seitenzugriff pro Tupel
- MDC:
  - ▶ Tupel mit gleichen Werten bzgl. mehrerer Attribute (Dimensionen) auf gleicher Seite bzw. im gleichen Extent speichern
  - ▶ Indexierung über Block-Indexe (dünn besetzte Indexe)

# MDC-Tabellen und Block-Indexe



## Anlegen einer MDC-Tabelle

```
CREATE TABLE Verkauf (  
    Umsatz number,  
    Jahr int,  
    Stadt varchar(20),  
    ...  
) ORGANIZE BY DIMENSIONS (Stadt, Jahr)
```

# Multidimensionale Speicherung – MOLAP

- Verwendung unterschiedlicher Datenstrukturen für Datenwürfel und Dimension
- Speicherung des Würfels als Array
- Ordnung der Dimension für Adressierung der Würfelzellen notwendig
- Häufig proprietäre Strukturen (und Systeme)

# Datenstrukturen für Dimensionen

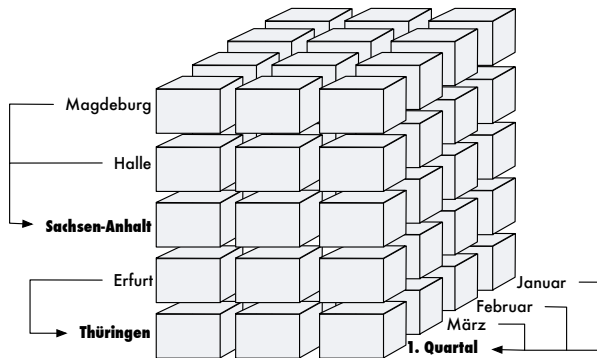
- Endliche, geordnete Liste von Dimensionswerten
- Dimensionswerte: einfache unstrukturierte Datentypen (String, Integer, Date)
- Ordnung der Dimensionswerte (interne ganze Ordnungszahl 2 oder 4 Byte)
  - Endlichkeit der Werteliste

# Datenstruktur für Würfel

- Für  $n$  Dimensionen:  $n$ -dimensionaler Raum
- $m_i$  Dimensionswerte der Dimension  $i$ : Aufteilung des Würfels in  $m$  parallele Ebenen
- Durch Endlichkeit der Dimensionswerteliste: endliche, gleichgroße Liste von Ebenen je Dimension
- Zelle eines  $n$ -dimensionalen Würfels wird eindeutig über  $n$ -Tupel von Dimensionswerten identifiziert
- Zelle kann ein oder mehrere Kennzahlen eines zuvor definierten Datentyps aufnehmen
- Bei mehreren Kennzahlen: Alternative  $\rightarrow$  mehrere Datenwürfel

# Klassifikationshierarchien

- Dimensionswerte umfassen alle Ausprägungen der Dimension: Elemente (Blätter) und Knoten der höheren Klassifikationsstufen
- Knoten der höheren Stufen bilden weitere Ebenen





# Berechnung von Aggregationen

- **Echtzeit:**
  - ▶ Bei Anfrage von Zellen, die Werte einer höheren, aggregierten Klassifikationsstufe repräsentieren → Berechnung aus Detaildaten
  - ▶ Hohe Aktualität, jedoch hoher Aufwand
  - ▶ Eventuell Caching
- **Vorbereitung:**
  - ▶ Nach Übernahme der Detaildaten → Berechnung und Eintragen der Aggregationswerte in entsprechende Zellen
  - ▶ Neuberechnung nach jeder Datenübernahme notwendig
  - ▶ Hohe Anfragegeschwindigkeit, jedoch Zunahme der Würfelgröße und Laufzeitaufwand
- **Ausweg: inkrementelle Vorbereitung**

# Weitere Datenstrukturen

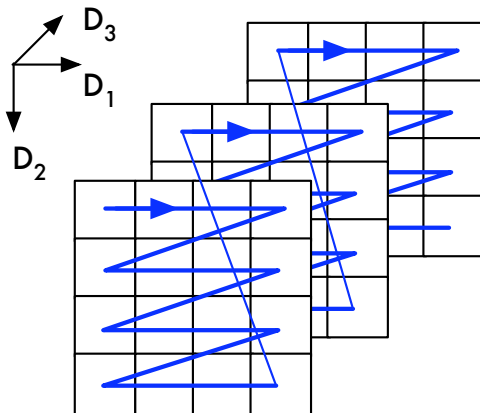
- Attribute
  - ▶ Klassifizierende Merkmale einer Dimension
  - ▶ Identifizierung von Untermengen von Dimensionswerten (z.B. "Produktfarbe")
  - ▶ Nicht zur Vorberechnung vorgesehen
- Virtueller Würfel
  - ▶ Umfasst abgeleitete Daten („Gewinn“, „prozentualer Umsatz“)
  - ▶ Ableitung aus anderen Würfel durch Anwendung von Berechnungsfunktionen  $\approx$  Sichten im relationalen Modell
- Teilwürfel
  - ▶ Kombination mehrerer Ebenen eines Würfels  $\rightarrow$  virtuell

# Array-Speicherung

- Speicherung des Würfels:  $n$ -dimensionales Array  
→ Linearisierung in eine eindimensionale Liste
- Indizes des Arrays  
→ Koordinaten der Würfelzellen (Dimensionen  $D_i$ )
- Indexberechnung für Zelle  $z$  mit Koordinaten  $x_1 \dots x_n$

$$\text{Index}(z) = x_1 + (x_2 - 1)|D_1| + (x_3 - 1)|D_1||D_2| + \dots + (x_n - 1)|D_1| \cdot \dots \cdot |D_{n-1}|$$

# Linearisierungsreihenfolge



# Array-Speicherung: Probleme

- Zahl der Plattenzugriffe bei ungünstigen Linearisierungsreihenfolgen
  - ▶ Reihenfolge der Dimensionen ist bei Definition des Würfels zu beachten
- Caching zur Reduzierung notwendig
- Speicherung dünn besetzter Würfel

# Speicherverbrauch

	<b>Array</b>	<b>Relational</b> (Star-Schema)
Speicherung Koordinaten	Implizit (Linearisierung)	Explizit (redundant)
Leere Zellen	Belegen Platz	Belegen keinen Platz
Neue Klassif. Knoten	Komplette Reorganisation <i>Starkes Wachstum im Speicherverbrauch</i>	Neue Zeile in Dimensionstabelle <i>Kaum Wachstum im Speicherverbrauch</i>
Speicher- verbrauch	$b \cdot \prod_{i=1}^n d_i$	$b \cdot M \cdot (n + 1)$

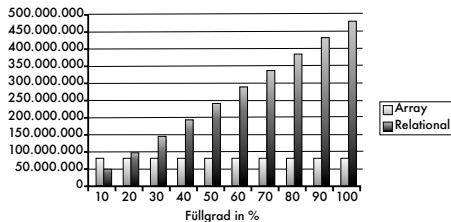
$M$ : Anzahl Fakten, d.h.  $M = \delta \cdot \prod_{i=1}^n d_i$  (Füllgrad  $\delta$ )

# Vergleich Speicherplatz

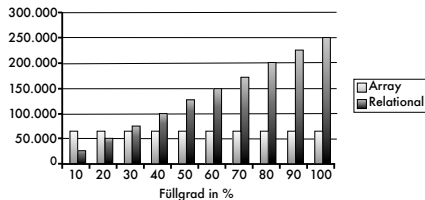
- Faktoren
  - ▶ Füllgrad
  - ▶  $k$ : Anzahl Knoten
  - ▶  $n$ : Anzahl Dimensionen
- Schon bei geringen Füllgraden ist Array Speicherplatz-effizienter
- Performance hängt von vielen Faktoren ab
  - ▶ Indexierung
  - ▶ Sequential reads
  - ▶ ...

# Vergleich Speicherplatz (2)

Speicherplatz nach Füllgrad,  $b=8$ ,  $k=100$ ,  $n=5$



Speicherplatz nach Füllgrad,  $b=8$ ,  $k=20$ ,  $n=3$





# Grenzen der multidimensionalen Speicherung

- Skalierbarkeitsprobleme aufgrund dünn besetzter Datenräume
- Teilweise einseitige Optimierung bezüglich Leseoperationen
- Ordnung der Dimensionswerte notwendig (durch Array-Speicherung)
  - ▶ Erschwert Änderungen an den Dimensionen
- Kein Standard für multidimensionale DBMS
- Spezialwissen notwendig

# Hybride Speicherung – HOLAP

- Verbindung der Vorteile beider Welten
  - ▶ Relational (Skalierbarkeit, Standard)
  - ▶ Multidimensional (analytische Mächtigkeit, direkte OLAP-Unterstützung)
- Speicherung:
  - ▶ Relationale Datenbank: Detaildaten
  - ▶ Multidimensionale Datenbank: aggregierte Daten
  - ▶ Multidimensionale Speicherstrukturen als „intelligenter“ Cache für häufig angeforderte Datenwürfel
- Transparenter Zugriff über multidimensionales Anfragesystem

# Speicherungsvarianten

- Ziel:
  - ▶ Optimierung für Leseoperationen, spez. OLAP-Anfragen (Aggregationen)
  - ▶ Schnelles Laden der tatsächlich benötigten Daten in Hauptspeicher für Berechnung
- Aspekte:
  - ▶ Partitionierung: leere Bereiche entfernen
  - ▶ Komprimierung: Speicherung von Nullwerten und redundanten Daten vermeiden
  - ▶ Indexierung (*nächstes Kapitel*):
    - ★ Von Datenblöcken (Grid-Files, R+-Bäume, Zwei-Ebenen)
    - ★ In einem Datenblock (Array-/relationale Speicherung der Zellen, RLE, Bitmap)
- Insgesamt: Erhaltung der räumlichen Nachbarschaftsbeziehung der Zellen im Sekundärspeicher (multidimensionales Clustering)

# Partitionierung von Datenwürfeln

- Ziel:
  - ▶ Entfernen leerer Bereiche aus Würfel
  - ▶ Optimierte Speicherung für Zugriffsmuster: häufig zugriffene Bereiche in einigen wenigen Blöcken
- Kriterien:
  - ▶ Art der Partitionierung
  - ▶ Steuerung: Optimierung der Partitionierung für Anwendung
  - ▶ Werkzeugunterstützung: für Steuerung der Partitionierung

# Art der Partitionierung

- Aufteilung eines Würfels in nicht überlappende Bereiche (**multidimensionale Intervalle**)
- Allgemeine Form: multidimensionale Kachelung
  - ▶ Geg.:  $n$ -dimensionales Array mit Dimensionswerten

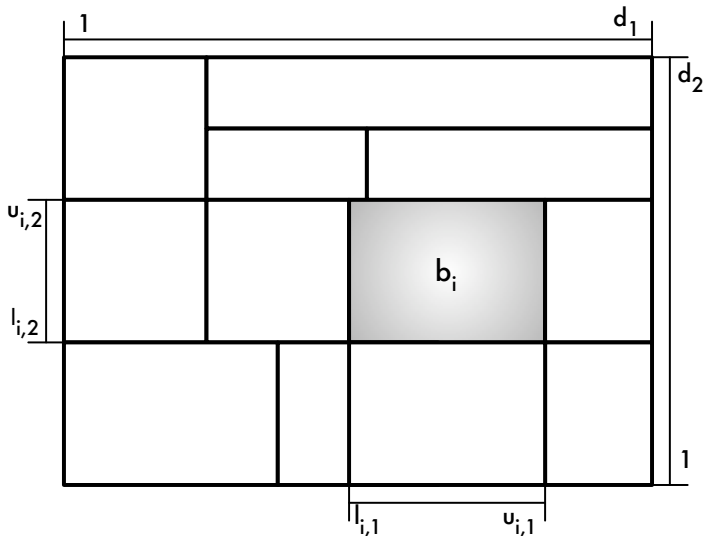
$$D = [1 : d_1, \dots, 1 : d_n]$$

- ▶ Kachelung: Menge von Sub-Arrays, die Bereichen  $b_1, \dots, b_m$  der Dimensionswerte entsprechen

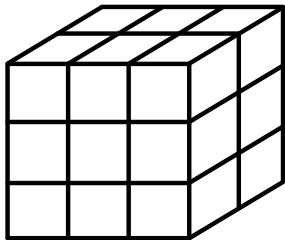
$$b_1 = [l_{1,1} : u_{1,1}, \dots, l_{1,n} : u_{1,n}], \dots, b_m = [l_{m,1} : u_{m,1}, \dots, l_{m,n} : u_{m,n}],$$

so dass  $b_i \cap b_j = \emptyset$  für  $i \neq j$  und  $b_i \subseteq D$ ,  $i, j = 1, \dots, m$  und jede besetzte Zelle einem Sub-Array angehört

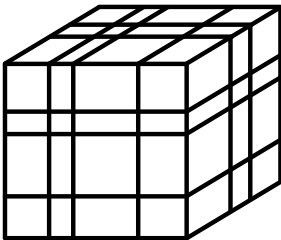
# Multidimensionale Kachelung



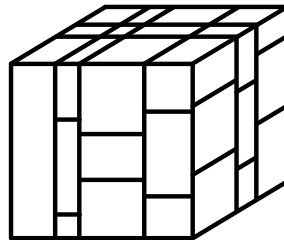
# Kachelung nach Ausrichtung



ausgerichtet,  
regulär

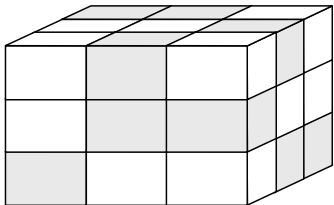


ausgerichtet,  
irregulär

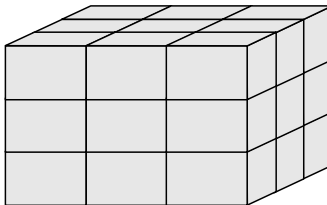


unausgerichtet

# Kachelung nach Besetzung des Raumes



dünn besetzt



dicht besetzt

## Legende



besetzte  
Kachel



unbesetzte  
Kachel



# Steuerung der Partitionierung

- Automatische Partitionierung:
  - ▶ Automatisches Finden der Partitionierung für optimale Ausführung der Operationen
  - ▶ Nutzung des Füllgrades der Bereiche
  - ▶ Nutzung von Zugriffsstatistiken
- Bedeutung bestimmter Dimensionen/ Dimensionskombinationen
  - ▶ Besondere Behandlung der Zeitdimension
  - ▶ Partitionierung nach Zeitreihen (spezielle Formate für Reihen von Werten, z.B. täglich, wöchentlich, etc.)
- Zwei-Ebenen-Speicherung
  - ▶ Nur Speicherung von verwendeten Kombinationen dünn besetzter Dimensionen
- Partitionenspezifikation des Anwenders
  - ▶ Direkte Spezifikation jedes Bereiches
  - ▶ Dimensionspartitionen

# Speicherung der Zellen

- Verwendung eines bestimmten Speicherformats für jeden Datenblock
- Unterstützung verschiedener Speicherformate (in Abhängigkeit vom Füllgrad)
- Ab bestimmten Füllgrad: Array-Speicherung effizienter als relationale Speicherung
  - ▶ Grund: Speicherung der Koordinaten als Schlüssel bei relationaler Speicherung notwendig

## Minimaler Füllgrad für optimale Speicherung

- Ab einem berechenbaren **minimalen Füllgrad** ist Array-Speicherung besser als relationale Speicherung
- Minimaler Füllgrad  $\delta$  ist maximales  $\delta$  so dass gilt:

$$Ix_{rel} + \delta \prod_{i=1}^n L_i \cdot \left( s_c + \sum_{j=1}^n s_j \right) < Ix_{arr} + \prod_{i=1}^n L_i \cdot s_c$$

- $L_i$ : Länge des Sub-Array in Dimension  $i$
- $s_c$ : Speichergröße der Zellen (Platzverbrauch aller Kenngrößen einer Zelle)
- $s_j$ : Speichergröße der Dimensionsattribute  $j$
- $Ix_{rel}$ : Speichergröße der Indizierung (relationale Speicherung)
- $Ix_{arr}$ : Speichergröße der Indizierung (Array-Speicherung)

## Minimaler Füllgrad: Beispiel

- Annahme:  $Ix_{rel}$  und  $Ix_{arr}$  gleich,  $s_j = s_c = 8$

- 2 Dimensionen:

$$\delta \prod_{j=1}^2 L_j \cdot 24 < \prod_{j=1}^2 L_j \cdot 8$$

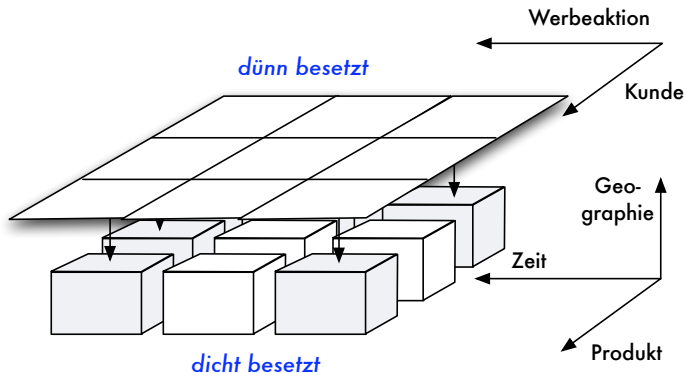
- Array-Speicherung effizienter ab Füllgrad 0.33
- Für drei Dimensionen: 0.25

⇒ Füllgrad sinkt mit steigender Anzahl von Dimensionen

## Zwei-Ebenen-Datenstruktur

- Obere Ebene indiziert Datenblöcke, die auf unterer Ebene gespeichert werden
- Untere Ebene: Array mit allen möglichen Kombinationen von Dimensionswerten
- Zellen des Array:
  - ▶ Zeiger auf Datenblock, der Datenwerte für entsprechenden Dimensionswert der dicht besetzten Dimensionen enthält
  - ▶ **NULL** für leeren Bereich

# Zwei-Ebenen-Datenstruktur



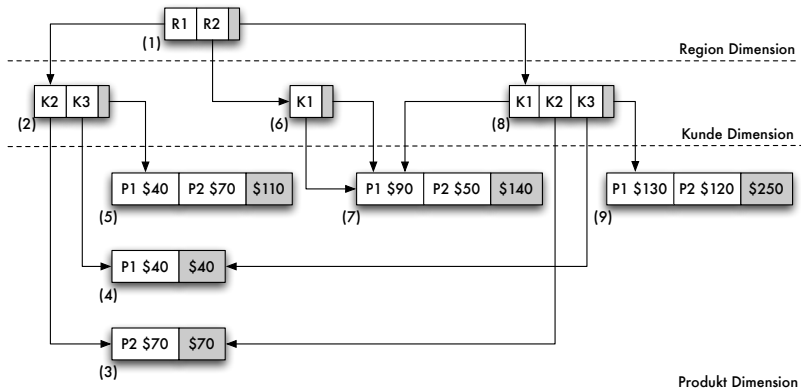
## Dwarf – Schrumpfen des Petacubes

- In hohem Maße komprimierte Datenstruktur
- Prefix und Suffix Redundanzen werden vereinigt
  - ▶ Prefix geeignet für dichte Bereiche
  - ▶ Suffix ist geeignet für dünn besetzte Bereiche
- 1 Petabyte Cube mit 25 Dimensionen → 2.3 GB Dwarf
- geeignet für Verteilung in mobilen Netzen

### Beispiel

<b>Region</b>	<b>Kunde</b>	<b>Produkt</b>	<b>Preis</b>
R1	K2	P2	70
R1	K3	P1	40
R2	K1	P1	90
R2	K1	P2	50

# Dwarf: Beispiel





## Weitere multidimensionale Speicherstrukturen

- Cube Forests und Hierarchically Cube Forests [Johnson & Shasha 1996, 1997]
- CubeTree [Roussopoulos & Kotidis & Roussopoulos 1997]
- CubiST [Fu & Hammer 2000]
- Condensed Cube [Wang & Lu & Feng & Yu 2002]
- Quotient Cube [Lakshmanan & Pei & Han 2002]
- m-Dwarf [Michalarias & Omelchenko & Lenz 2009]

## Weitere multidimensionale Speicherstrukturen (2)

Basierend auf Iceberg Cubes

- Bottom-Up Cube [Beyer & Ramakrishnan 1999]
- H-Cubing [Han & Pei & Dong & Wang 2001]
- Star Cubing [Xin & Han & Li & Wah 2003]

# Zusammenfassung

- Relationale vs. multidimensionale Speicherung
- relationale Erweiterungen
  - ▶ Partitionierung
  - ▶ spezielle Tabellentypen
- Spezialitäten der multidimensionalen Speicherung
  - ▶ Array-Speicherung
  - ▶ Kachelung
  - ▶ Umgang mit dünn besetzten Würfeln
- Hybride Formen

# Motivation für Spaltenorientierte Datenhaltung

- Anfragen dienen Analyse von Daten (lange Transaktionen)
- Datenbestand stabil, d.h. wenige/keine Updates
- Import von Daten (oft) über ETL-Prozess
- Einzelwerte oft uninteressant (vgl. Anwendungsfelder DWH)
- Häufig erstellen und verarbeiten aggregierter Werte
  - ▶ AVG(), SUM(), COUNT()
  - ▶ GROUP BY (CUBE)
  - ▶ CUBE-Operationen
- Für Aggregatfunktionen (z.B. AVG()) einzelne Spalten interessant
- Auch Gruppierungen (und CUBE) intuitive spaltenweise Bearbeitung

# Datenexplosion

- Datenbestände im OLAP-Bereich wachsen ständig  $\leftrightarrow$  Daten in zentralisierten Systemen kaum noch verwaltbar
- Historisierung der Daten erhöht Datenvolumen zusätzlich
- OLAP-Anfragen sehr speicher- und rechenintensiv  $\leftrightarrow$  Verteilung der Last notwendig
- Für Aggregationen (OLAP) ist eine vertikale Partitionierung/ Fragmentierung sinnvoll  $\leftrightarrow$  bereits bestehende Partitionierung von Column Stores ausnutzen
- Aktuelle Systeme setzen Kompressionstechniken zur Datenvolumenreduktion ein

# OLAP: Row Store

- Historisch: Anwendung in On-line Transactional Processing (OLTP) mit kurzen Transaktionen, z.B. Buchungstransaktionen
- Abbildung von Tupeln in DBMS, d.h. Tupel sequentiell gespeichert
- Verarbeitung ganzer Tupel für Aggregatsfunktionen  $\leftrightarrow$  I/O Overhead
- **Insgesamt:** Tupel-orientierte physische Speicherung ungünstig für OLAP

Produkt	Ort	Umsatz	Jahr
Merlot	Magdeburg	4325	2010
Guinness	Magdeburg	2341	2010
Merlot	Ilmenau	5543	2010
Pinot Noir	Ilmenau	4944	2010

# OLAP: Column Store

- Neue Architektur für OLAP notwendig  
↳ Column Stores
- Tupel spaltenweise partitioniert, d.h. Werte einer Column sequentiell gespeichert (und sortiert)
- Aggregatsfunktionen arbeiten direkt auf Spalten ↳ nur benötigte Daten eingelesen
- **Insgesamt:** Column Stores können Aggregationen viel effektiver bearbeiten

Produkt
Merlot
Guinness
Merlot
Pinot Noir

Ort
Magdeburg
Magdeburg
Ilmenau
Ilmenau

Umsatz
4325
2341
5543
4944

Jahr
2010
2010
2010
2010

# Kompression

- Row Stores
  - ▶ Kompression relations- oder partitionsweise
  - ▶ Verschiedene Datentypen  $\leftrightarrow$  Trade off bei Kompressionstechnikauswahl notwendig
  - ▶ Übliche Kompressionsraten 2:1 bis 5:1
- Column Stores
  - ▶ Kompression je Column möglich
  - ▶ Nutzung verschiedenster Techniken, z.B. Run Length Encoding (RLE), Wörterbuchkodierung (Lempel-Ziv)
  - ▶ Auswahl bester Kompressionstechnik je Spalte, d.h. für jeden Datentyp
  - ▶ Übliche Kompressionsraten 10:1 bis 40:1
- Column Stores reduzieren I/O Overhead **und** verringern Datenvolumen z.T. erheblich  $\leftrightarrow$  bessere Ausnutzung des Hauptspeichers



# Anfragebearbeitung

- Column Stores basieren auch auf relationalem Datenmodell ↔ Verwendung der relationalen Algebra und derer Operationen
- Logischer Anfrageplan wie bei Row Stores
- Architekturspezifische Ausführung transparent
- Bitoperationen von Natur aus unterstützt (vgl. Bitmap-Join-Index)
- Spaltenweise Kompression erlaubt Bearbeitung ohne Dekompression
  - ▶ Verlustfreie Kompressionstechniken (Bekannteste: Lempel-Ziv und Derivate) ↔ gleiche (unkomprimierte) Werte haben gleiche komprimierte Darstellung
  - ▶ D.h. Vergleichswert wird falls nötig vor Anfrageausführung in komprimierte Darstellung überführt ↔ gut geeignet für nicht vektorbasierte Joins
  - ▶ Ordnungserhaltende Techniken wie RLE ↔ Werte ggf. direkt vergleichbar oder wie bei verlustfreier Kompression
  - ▶ D.h. Aggregatfunktionen wie MIN/MAX oder SUM können Daten komprimiert verarbeiten

## Anfrageplanausführung Column vs. Row Store

## Row Store

...	Ort	...	Umsatz	...
	MD		1	
	EF		5	
	MD		7	
	EF		4	

**SELECT SUM (Umsatz)**  
**FROM Filiale**  
**WHERE Ort = 'MD'**

$\sigma_{\text{Ort} = \text{'MD'}}$

...	Ort	...	Umsatz	...
	MD		1	
	MD		7	

$\pi_{\text{Umsatz}}$

Umsatz
1
7

**SUM**

SUM(Umsatz)
8

## Column Store

Ort
MD
EF
MD
EF

$\sigma_{\text{Ort} = \text{'MD'}}$

$\begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$

Umsatz
1
5
7
4

Merge

Umsatz
1
7

**SUM**

Sum(Umsatz)
8

Jahr
2010
2009
2011
2010

# Tupelrekonstruktion

- Operator heißt **SPC** (Scan, Predicate, Construct) für volle Tupelrekonstruktion
- Merge** ist ein k-tuplige Rekonstruktion (mit Spalten  $VAL_1 \dots VAL_k$ )

Zeilenorientiert

Seite 1

1	12	15,50	K9	2	
20	25,00	K5	3	10	13,
00	K1	6	13	14,00	
K32	5	14	19,75	K25	

Seite 2

4	33	72,30	K16	...
---	----	-------	-----	-----

ID	Anzahl	Kosten	Kunde
1	12	15,50	K9
2	20	25,00	K5
3	10	13,00	K1
4	33	72,30	K16
5	14	19,75	K25
6	13	14,00	K32
7	30	55,50	K33
8	29	57,00	K19
9	17	34,40	K30

Spaltenorientiert

Seite 1

1	2	3	4	5	6
7	8	9		12	20
10	33	14	13	30	29
17					

Seite 2

15,50	25,00	13,00	72,
30	19,75	14,00	55,50
57,00	34,40		

Seite 3

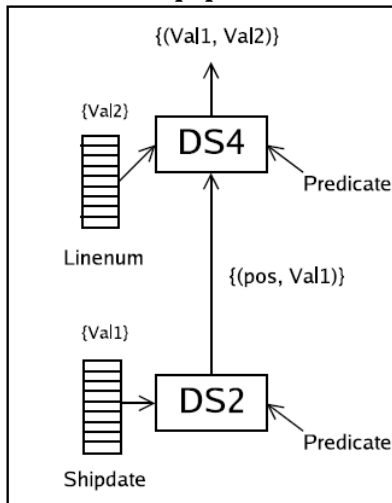
K9	K5	K1	K16	K25
K32	K33	K19	K30	

# Materialisierungszeitpunkt

- Frühe Materialisierung (EM)
  - ▶ Anfragebearbeitung sehr nah an Row Stores
  - ▶ Aggregatfunktionen auf einzelnen Columns
  - ▶ Tupelrekonstruktion sobald Tupel verwendet
  - ▶ Zumeist Verwendung bei tupel-orientierter Anfragebearbeitung
- Späte Materialisierung (LM)
  - ▶ So lang wie möglich auf Columns arbeiten
  - ▶ Mehrfacher Zugriff auf Basistabellen und/oder Zwischenergebnisse
  - ▶ Folge: Anfrageplan kein Baum mehr
  - ▶ Aber: Gleichzeitige Bearbeitung auf komprimierten und unkomprimierten Daten möglich
  - ▶ Notwendig für (effektive) spalten-orientierte Anfragebearbeitung

# Frühe Materialisierung (EM)

## EM-pipelined



## EM-parallel

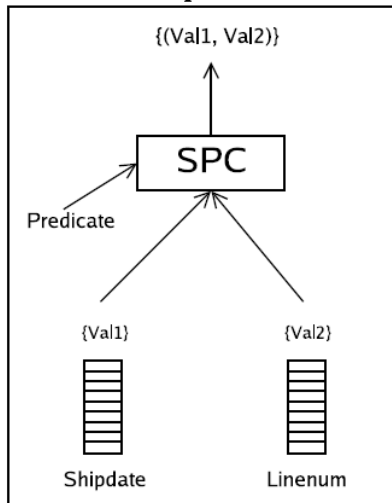


Abb. aus "Query execution in column-oriented database systems", PhD thesis by D. Abadi

# Späte Materialisierung (LM)

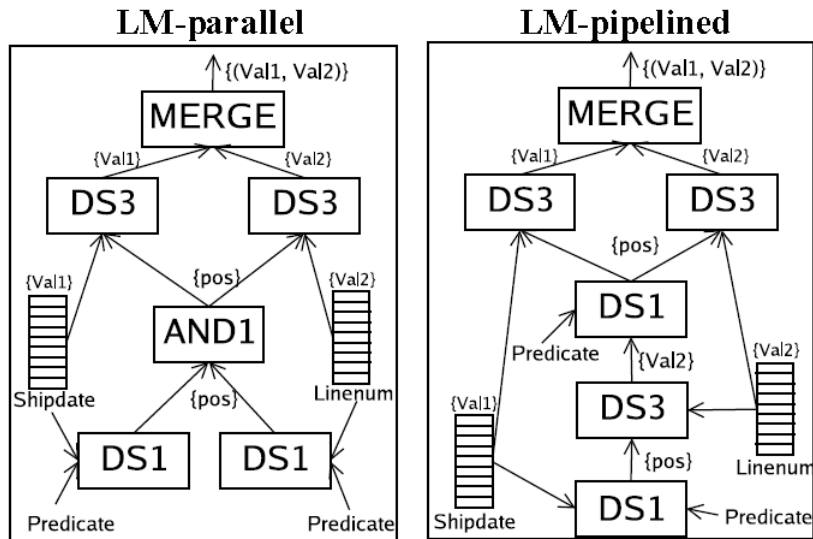


Abb. aus "Query execution in column-oriented database systems", PhD thesis by D. Abadi

# Nachteile

- Tupelrekonstruktion verursacht Kosten
- Kosten für Einfügeoperation durch Partitionierung der Tupel
- Updates benötigen Tupelrekonstruktion
- Folge: Insert- und Update-in-place nicht möglich
- **Aber:** Updates und Inserts in OLAP/DWH Anwendung selten oder nur durch ETL

# Lösungsansätze

- Tupel-orientierte Anfragebearbeitung und frühe Materialisierung
- C-Store/Vertica
  - ▶ Read-optimized (RS) and write-optimized Storage (WS)
  - ▶ Verschiedene sich überlappende Projektionen im RS
  - ▶ Inserts und Updates nur im WS
  - ▶ Tuple mover übermittelt Daten vom WS in RS zu geringer Last (offline)
- SybaseIQ (erster kommerzieller Column Store)
  - ▶ Ähnlich zum C-Store-Ansatz
  - ▶ System unterteilt in Read- und Write- bzw. Read/Write-Knoten
  - ▶ Abgleich im Hintergrund zu Zeiten geringer Last
- Redundanz
  - ▶ Daten column- und row-orientiert im Hauptspeicher
  - ▶ Datenbank redundant als Column und Row Store
  - ▶ Virtualisierung des Datenwürfels
- ...



# Systeme

- Kommerziell
  - ▶ SybaseIQ
  - ▶ Vertica
  - ▶ Infobright ICE
  - ▶ Tenbase (webbasiert)
  - ▶ BigTable (Google, nicht relational)
  - ▶ ...
- Frei
  - ▶ Infobright ICE Community Edition
  - ▶ LucidDB
  - ▶ MonetDBX100 (Ingres/Vectorwise)
  - ▶ C-Store (benötigt alten gcc)
  - ▶ Hbase (Apache), Hypertable, Cassandra (Facebook) alles BigTable-Derivate
  - ▶ ...
- Im Gegensatz zu Row Stores weichen Column Store Implementierungen untereinander sehr stark ab

# Zusammenfassung

- Row Stores nicht optimal für OLAP- und DWH-Anwendungen
- Column Stores besser geeignet für Aggregatsfunktionen
- Column Stores verringern Datenvolumen z.T. drastisch ↔ geringerer I/O, bessere Ausnutzung des Hauptspeichers
- Darstellung von Tupeln erzeugt Kosten in Column Stores (Tupelrekonstruktion)
- Column Stores zeigen Schwächen bei Inserts und Updates
- Sehr viele und stark unterschiedliche Implementierungen für Column Stores