

# Data Warehousing und Data Mining

Sprachen für OLAP Operationen

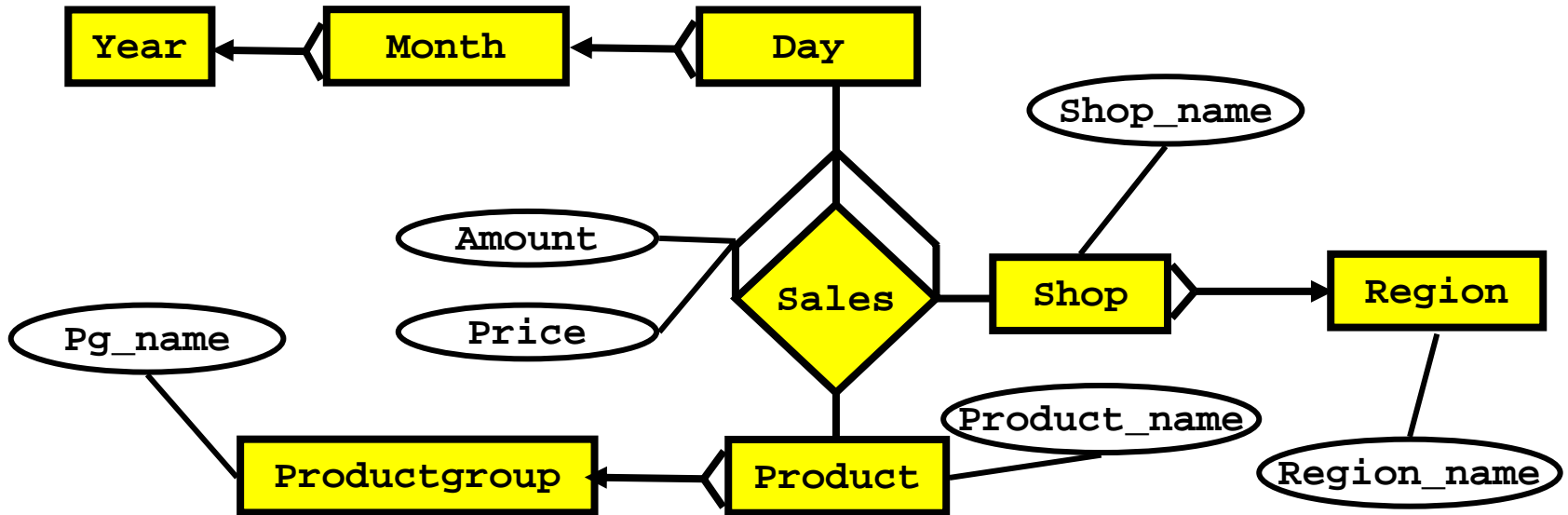


Ulf Leser

Wissensmanagement in der  
Bioinformatik

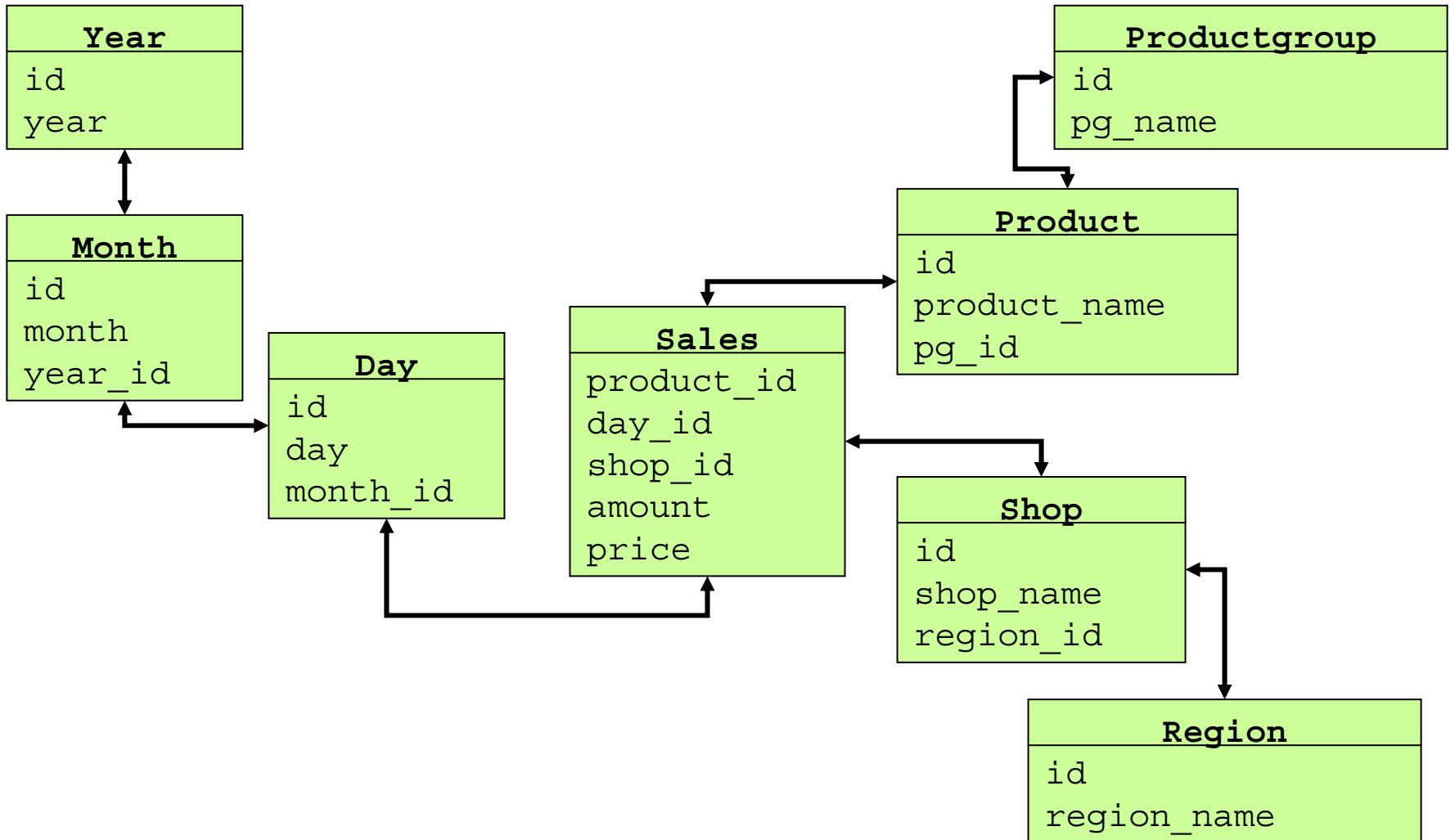


# Beispielquery

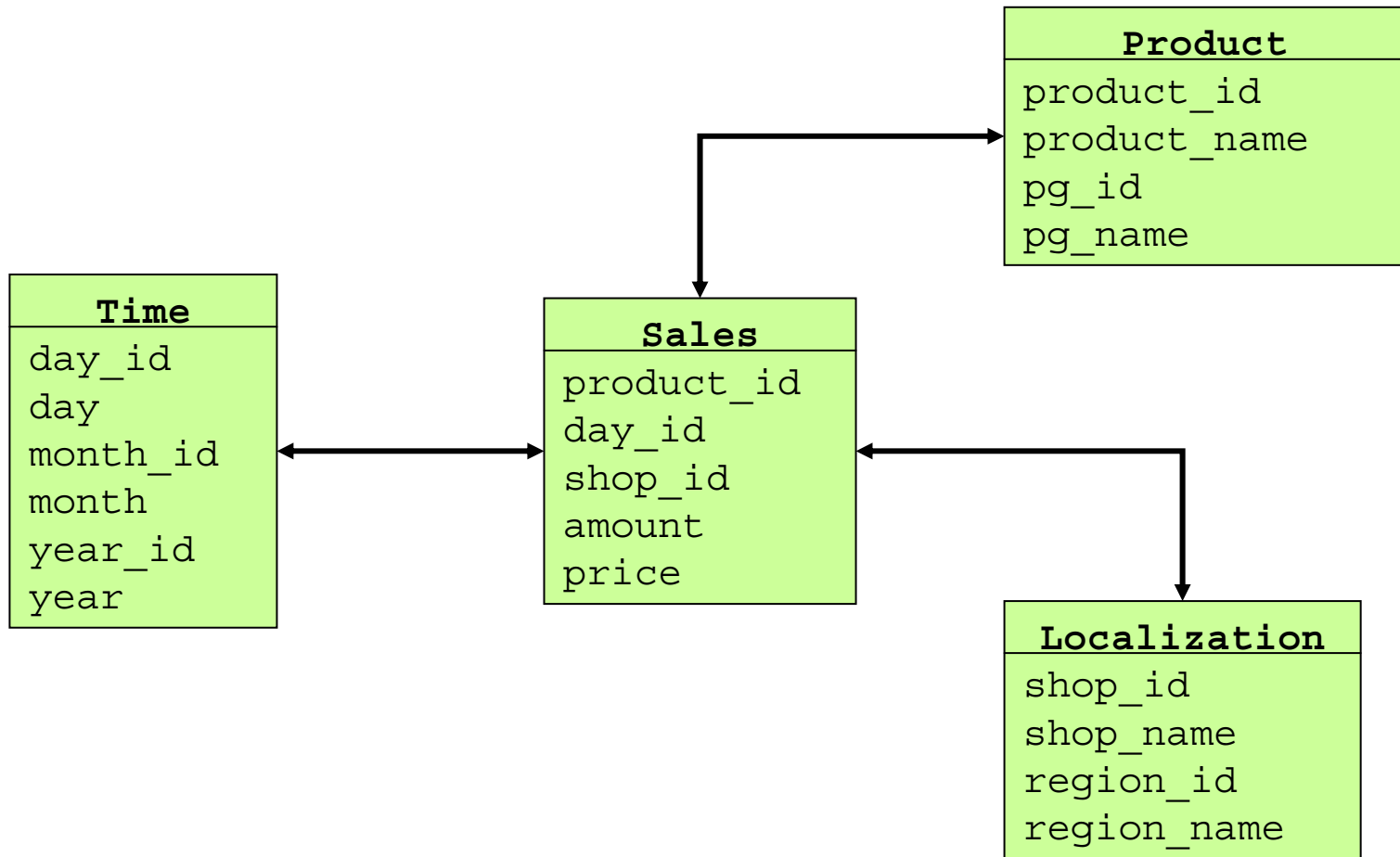


Alle Verkäufe der Produktgruppe „Bier“ nach Shop und Jahr

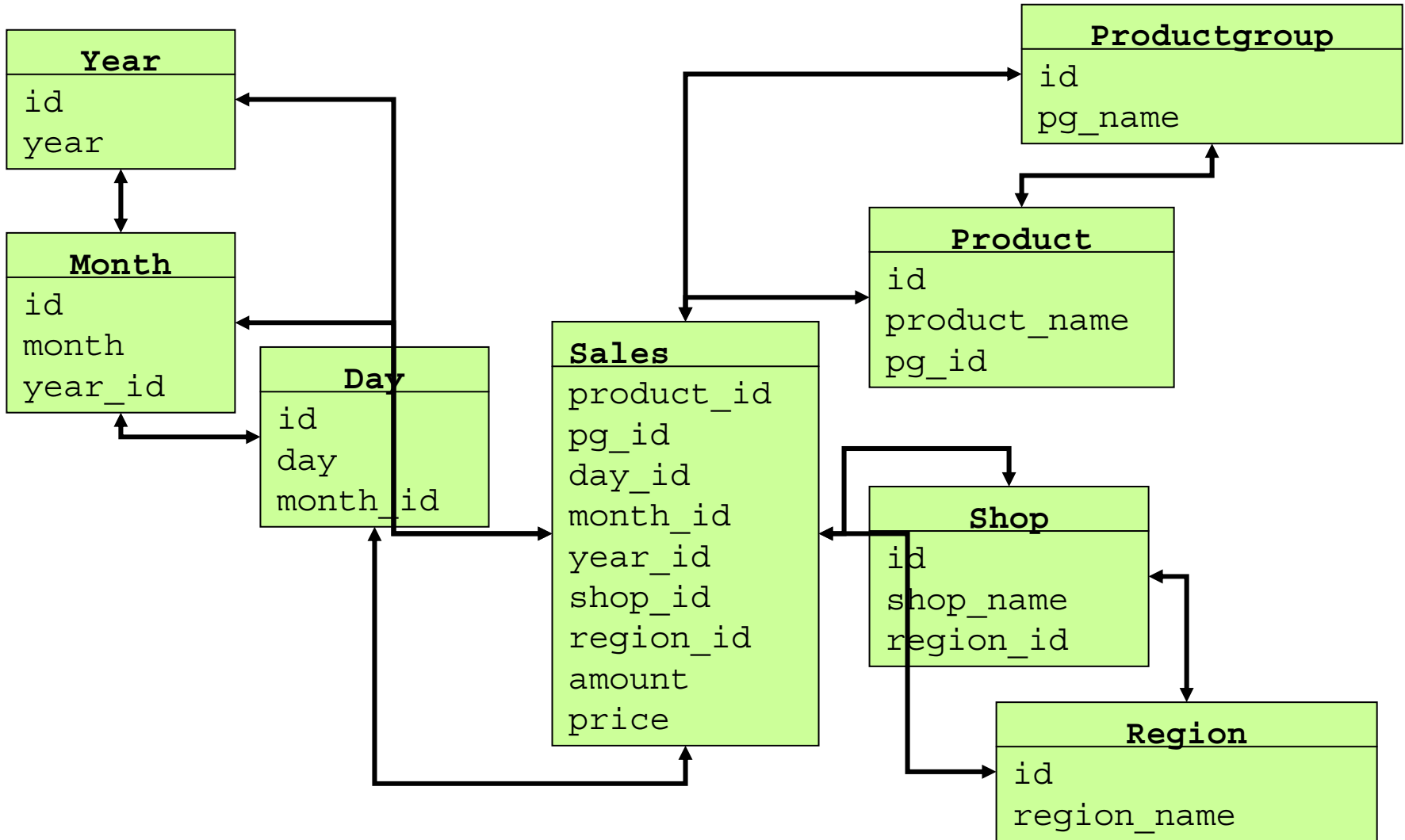
# Variante 1 - Snowflake



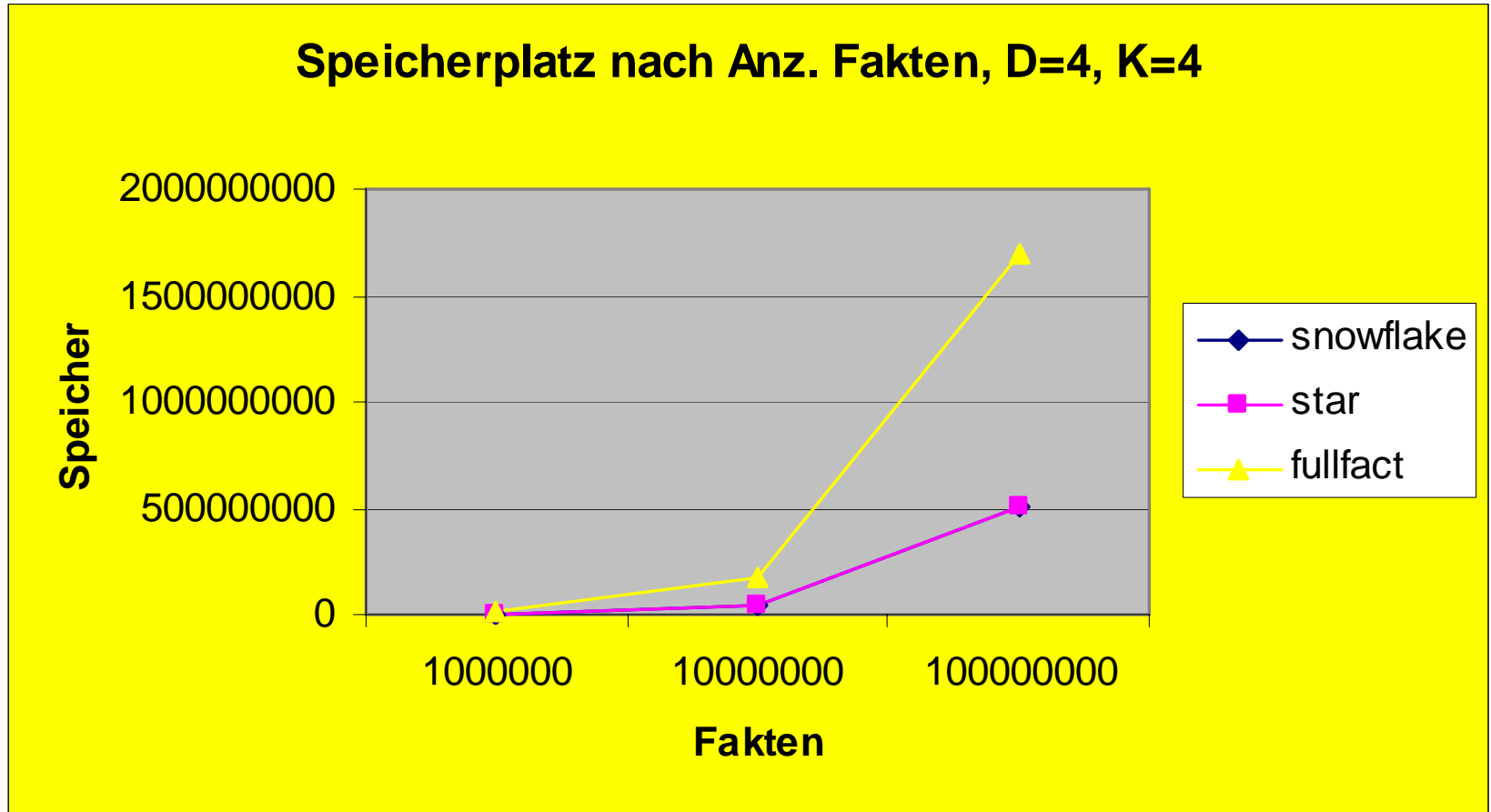
# Variante 2: Star Schema



# Variante 3: Fullfact



# Speicherverbrauch 2



# Fazit – Speicher und Query

---

- Speicherverbrauch Snowflake / Star praktisch identisch
  - Wenn Bedarf für Dimensionen vernachlässigbar
- Fullfact mit deutlich höherem Speicherverbrauch
  - Dafür minimale Anzahl Joins (im Idealfall 0)
- Laufzeitverhalten hängt von mehr Faktoren als dem Schema ab
  - Bereichs- oder Punktanfrage
  - Indexierung
  - Selektivität der Bedingungen
  - Gruppierung und Aggregation
  - ...
- ... aber **Joins sind tendenziell teuer**

# Änderungskosten

Semantisch fragwürdig: NULL in Knoten mit Level  $j < i$

	Snowflake	Star	Fullfact
<b>InsN</b>	1 Insert	(1 Insert)	1 Insert
<b>InsS</b>	1 neue Tabelle, $3^{(k+1)-i}$ Updates (Umhängung)	1 neues Attribut, $3^k$ Updates (Wert des Attrib.)	1 neue Tabelle, 1 neues Attribut, m Updates (in Faktentabelle)
<b>DelN</b>	1 Delete, 3 Update	0 Delete, 3 Update	1 Delete, $m/3^{k-i}$ Updates (Level i: $3^{k-i}$ Knoten)
<b>DelN<sub>0</sub></b>	1 Delete, $m/3^k$ Update	1 Delete, $m/3^k$ Update	1 Delete, $m/3^k$ Updates

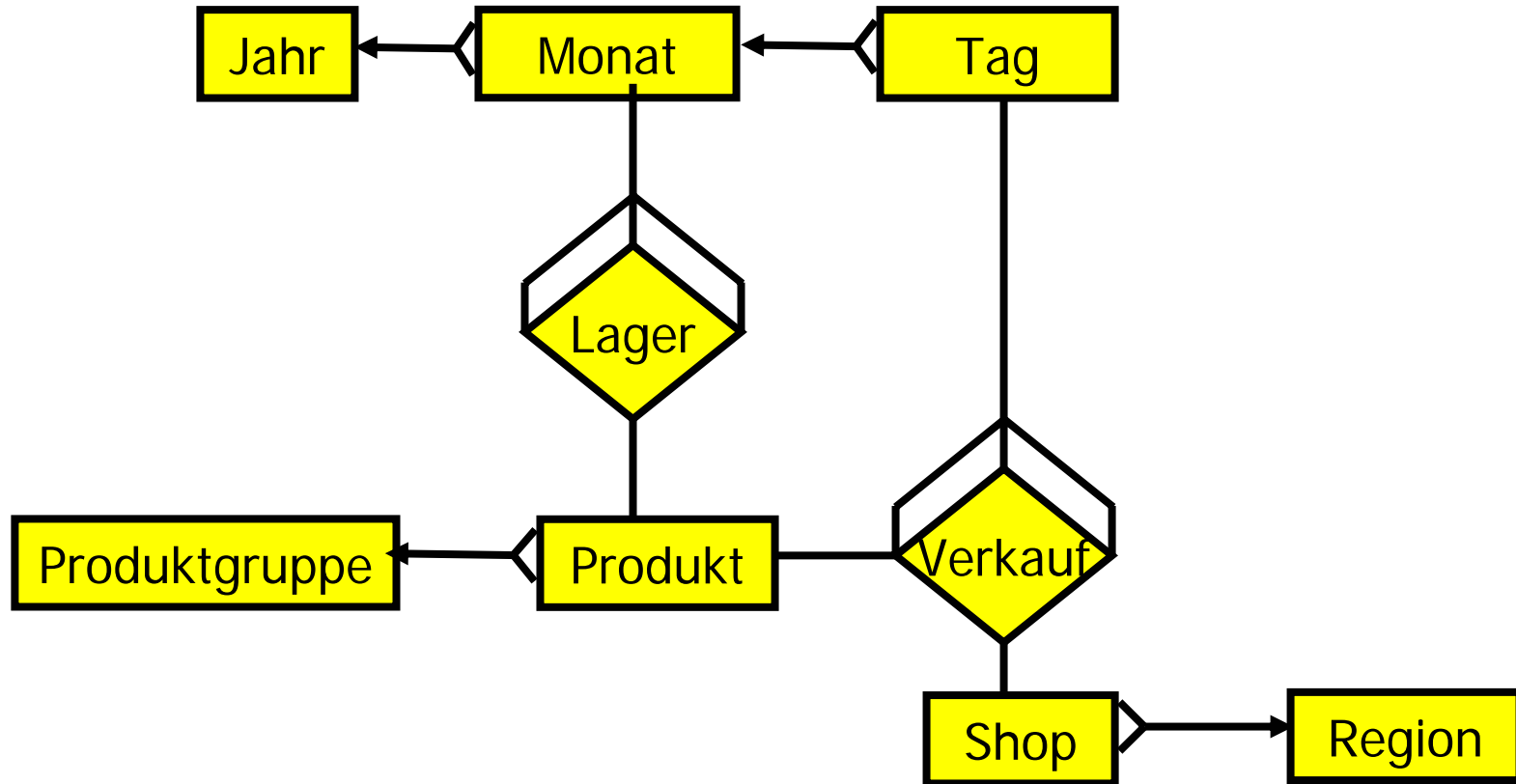
# Fazit

---

- Insert/Delete in Dimensionen
  - Teuer bei Fullfakt
  - Moderat bei Star / Snowflake
- Insert/Delete von Blatt – Klassifikationsknoten
  - Werden durch FK aus Faktentabelle adressiert
  - Immer teuer (je nach Größe und Werteverteilung innerhalb der Dimension)
- Star Schema „verlangt“ **balancierte Hierarchien**
  - Sonst NULL Werte in Knoten-IDs und -attributen
  - Probleme beim Aggregieren/Gruppieren über NULLs

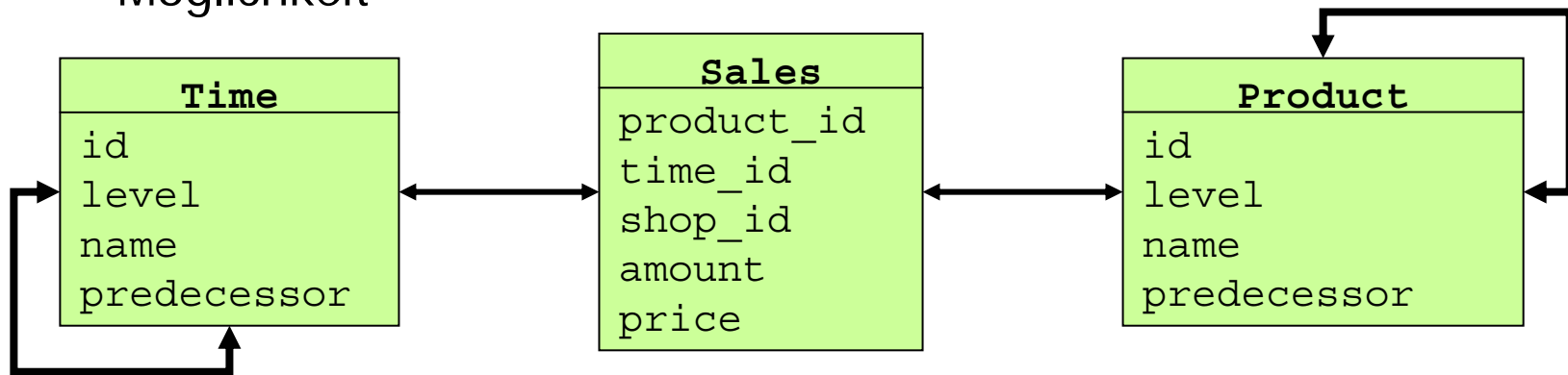
# Galaxy Schema

Mehrere Faktentabellen



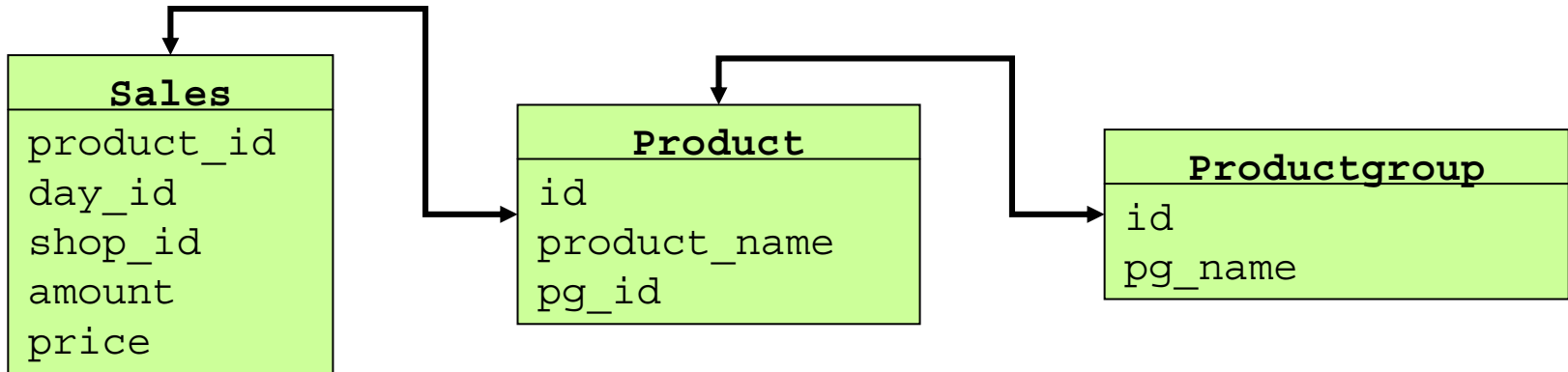
# Rekursive Modellierung

- **Unbeschränkt lange, unbalancierter Klassenhierarchien**
  - Beispiel: Stücklisten (Teil von ...)
  - Weder in Star noch Snowflake Schema möglich
  - Erweiterungen immer DDL Operationen
- Möglichkeit



- Probleme
  - **Rekursive Anfragen** oder viele Self-Joins (wie viele ?)
  - Keine individuellen Attribute pro Stufe
  - **Eingeschränkte referenzielle Integrität**
    - `Sales.time_ID` könnte auf Knoten in `time` mit `Level≠0` zeigen

# Beispiel: Snowflake Schema



```
CREATE DIMENSION s_pg
  LEVEL product IS (product.id)
  LEVEL productgroup IS (productgroup.id)
HIERARCHY pg_rollup (
  product CHILD OF productgroup
  JOIN KEY (pg_id)
  REFERENCES productgroup
);
```

# Multiple Hierarchien in einer Dimension

Dimension

Klassifikationsstufen

```
CREATE DIMENSION times_dim
  LEVEL day IS TIMES.TIME_ID
  LEVEL month IS TIMES.CALENDAR_MONTH_DESC
  LEVEL quarter IS TIMES.CALENDAR_QUARTER_DESC
  LEVEL year IS TIMES.CALENDAR_YEAR
  LEVEL fis_week IS TIMES.WEEK_ENDING_DAY
  LEVEL fis_month IS TIMES.FISCAL_MONTH_DESC
  LEVEL fis_quarter IS TIMES.FISCAL_QUARTER_DESC
  LEVEL fis_year IS TIMES.FISCAL_YEAR
HIERARCHY cal_rollup (
  day CHILD OF
  month CHILD OF
  quarter CHILD OF
  year )
HIERARCHY fis_rollup (
  day CHILD OF
  fis_week CHILD OF
  fis_month CHILD OF
  fis_quarter CHILD OF
  fis_year )
<attribute determination clauses>...
```

Klassifikationspfad

Klassifikationspfad

# Multidimensionales OLAP

---

- Grundidee
  - Speicherung multidimensionaler Daten ...
  - ... in [multidimensionalen Arrays](#)
- Kommerzielle Produkte verwenden proprietäre (und geheime) Techniken
  - Siehe auch [multidimensionale Indexstrukturen](#) (später)
  - OLAP will vor allem aggregieren
  - [Range-Queries und Speicherung von prä-aggregierten Daten](#)
- Im folgenden nur
  - 1 Würfel, 1 Fakt
  - Keine Betrachtung von Knotenattributen
  - Klassifikationsknoten haben nur diskrete Werte
    - Bei numerischen Knoten: Diskretisierung

# Array Adressierung

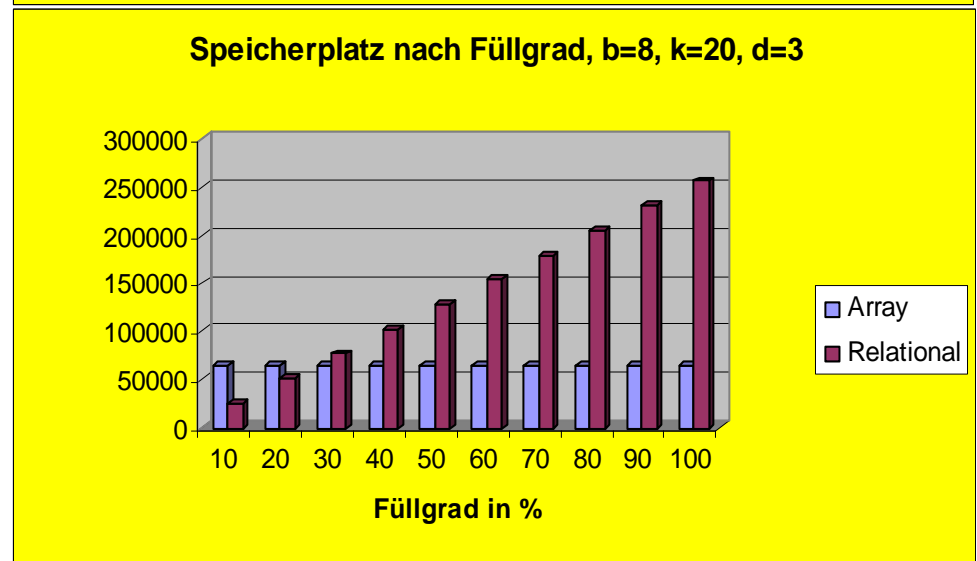
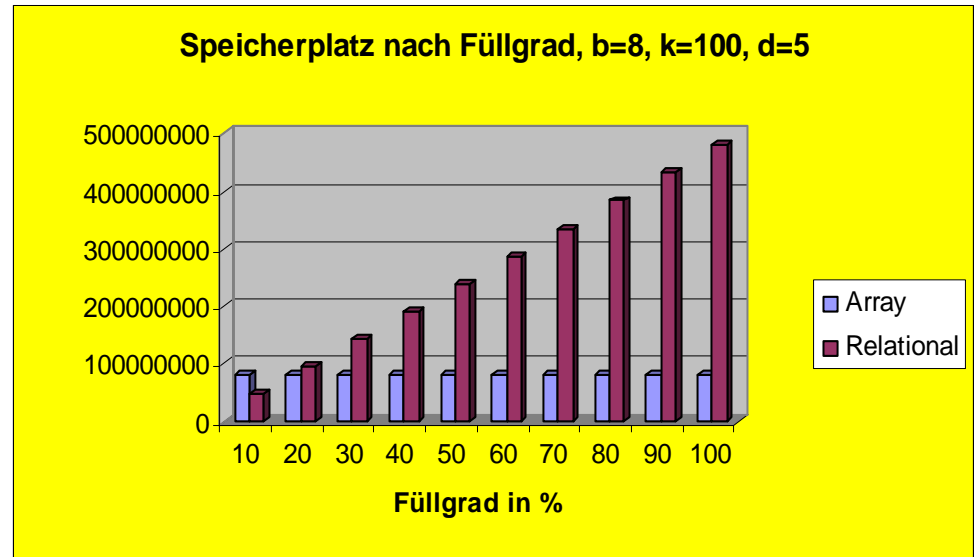
---

- Sei  $d_i = |D_i|$ ,  $d$  Dimensionen,
- $b$ : Speicherplatz pro Attribut (Key oder Wert)
- Linearisierte Darstellung
  - $\langle 1,1,1 \rangle, \langle 1,1,2 \rangle, \dots, \langle 1,1,d_1 \rangle, \langle 1,2,1 \rangle, \dots$
- Offset der Zelle  $\langle a_1, \dots, a_n \rangle$

$$b * \sum_{i=1}^d \left( (a_i - 1) * \prod_{j=1}^{i-1} d_j \right)$$

# Vergleich Speicherplatz

- Faktoren
  - Füllgrad
  - k: Anz. Knoten
  - d: Anz. Dimensionen
- Schon bei **geringen Füllgraden** ist **Array platzeffizienter**
- Aber: Tatsächlicher Füllgrad ist bei vielen Dimensionen bzw. Klassifikationsknoten oft erstaunlich gering
  - Bei weitem **nicht alle Kombinationen vorhanden**



# Inhalt dieser Vorlesung

---

- OLAP Operationen
- MDX: Multidimensional Expressions
- SQL Erweiterungen

# OLAP Operationen

---

- OLAP Operationen arbeiten auf einem Würfel
  - Aggregation: Roll-Up
  - Verfeinerung: Drill-Down
  - Selektion: Slice, Dice
- Kern-Operation: **Hierarchische Aggregation**
  - Roll-Ups mit Aggregation auf allen Ebenen (Summe pro Tag, pro Monat, pro Jahr)
  - Roll-Ups über mehrere Dimensionen (Cross-Tabs): Verkäufe pro Marke und Jahr und Shop, Summen in allen Kombinationen
- Im weiteren Sinne zählt man auch „analytische Anfragen“ zu den OLAP Operationen
  - Gleitende Durchschnitte, attributlokale Vergleiche, Arbeiten auf Zeitreihen – **Sequenzbasierte Operationen**

# Sprachen für OLAP Operationen

---

- Sprachen für OLAP Operationen
  - Bereitstellung der notwendigen Operationen
  - **Ökonomisches Design**
    - Keine vielfach geschachtelten SQL Operationen
    - Keine „Programmierung“
- Hauptsächlich zwei Ansätze
  - **Multidimensional Expressions (MDX)**
    - Basiert direkt auf MDDM Elementen: Cube, Dimension, Fakt, ...
    - Auf dem Vormarsch als Standard
  - **Erweiterungen von SQL**
    - Erweiterungen um spezielle Operationen (**ROLLUP**, **CUBE**, ...)
    - Daten müssen relational vorliegen
    - „MDDM-unaware“ – Anfragen müssen auf Star-/ Snowflake-Schema formuliert werden

# MDX

---

- MDX: **Multidimensional Expressions**
  - Microsoft's Vorschlag, „OLE DB for OLAP“
- Eigene Anfragesprache
  - Standard **ohne feste Semantik** (by example)
  - MDDM Konzepte als **First-Class Elemente**
  - Dadurch kompaktere Anfragen als mit SQL
  - SQL-artige Syntax
  - Sehr mächtig und komplex
- Erzeugung der Objekte (DDL) erfolgt anderweitig
  - DSO Interface (Decision Support Objects) von SQL Server
- Wird von vielen kommerziellen Tools zur Kommunikation mit OLAP Datenbank benutzt

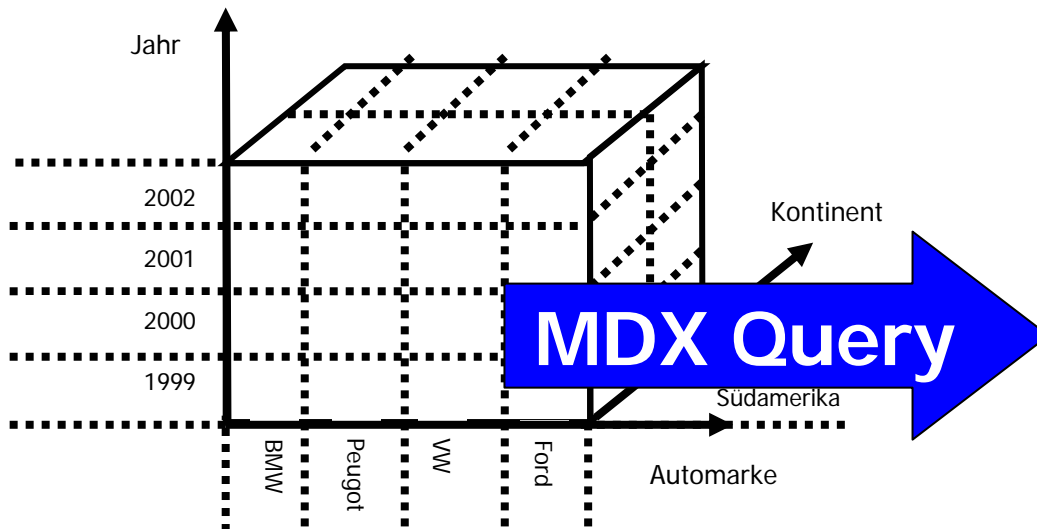
# MDX Elemente

---

- **Measures** = Fakten
  - Modelliert als eigene Dimension mit einer Stufe
  - Faktname ist der einzige Klassifikationsknoten
- **Dimensions** = Dimensionen
  - **Level** = Klassifikationsstufe
  - **Multiple hierarchies** = Verschiedene Pfade
  - **Member** = Klassifikationsknoten

# Grundprinzip

- MDX Anfragen erzeugen mehrdimensionale Reports



1997				

1998				

1999				

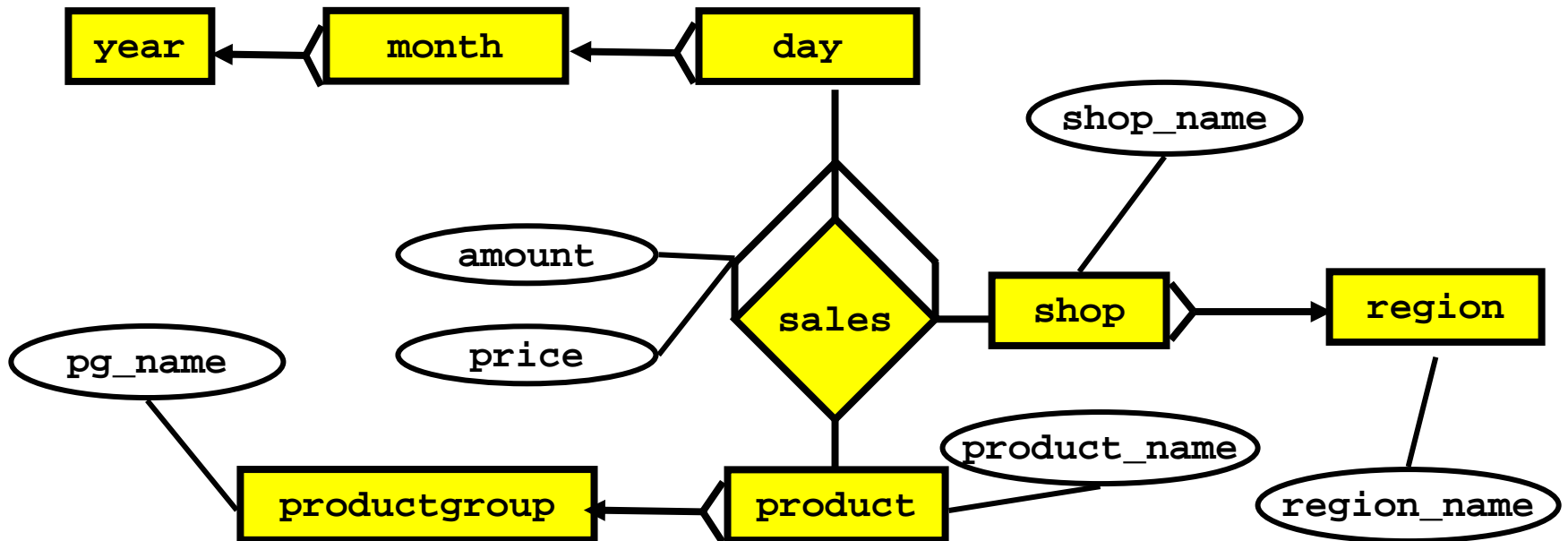
# Struktur einer MDX Query

---

```
SELECT <axis-spec1>, <axis-spec2>, ..  
FROM <cube-spec1>, <cube-spec2>, ...  
WHERE <slice-specification>
```

- **Dimensions** (**SELECT**)
  - Angabe der **darzustellenden Achsen**
  - Abbildung in „mehrdimensionale“ Tabellen
    - **ON COLUMNS, ROWS, PAGES, CHAPTER, ...**
  - Jede Achsenspezifikation muss eine Menge von Members beschreiben
    - Als explizit angegebene Menge (in {})
    - Oder durch Funktionen
  - Namen von Members werden in [] verpackt
    - Wenn nötig zur syntaktischen Abgrenzung
- **Cube** (**FROM**)
  - Auswahl des Basis-Cubes für die Anfrage
- **Slicer** (**WHERE**)
  - Auswahl des betreffenden Fakten
  - Intuitiv zu lesen als „etwas zu tun haben mit“

# Beispielschema



## Member: Unique Names oder Navigation

Verschiedene Klassifikationsstufen in einer Dim möglich

```
SELECT {Wein, Bier, Limo, Saft} ON COLUMNS  
      {[2000], [1999], [1998].[1], [1998].[2]} on ROWS  
FROM Sales  
WHERE (Measures.Menge, Region.Germany)
```

Auswahl des Fakt „Menge“  
Beschränkung auf Werte in BRD

	Bier	Limo	Saft
2000			
1999			
1998.1			
1998.2			

Implizite Summierung

# Navigation in den Hierarchien

---

```
SELECT {Prodgroup.MEMBERS, [Becks Bier].PARENT} ON COLUMNS
       {Year.MEMBERS} on ROWS
FROM Sales
WHERE (Measures.Menge)
```

## Navigationfunktionen

- **MEMBERS:** Knoten einer Klassifikationsstufe
- **CHILDREN:** Kinderknoten eines Klassifikationsknoten
- **PARENT:** Vaterknoten eines Klassifikationsknoten
- **LASTCHILD, FIRSTCHILD, NEXTMEMBER, LAG, LEAD...**
- **DESCENDENTS:** Rekursives Absteigen

# Crossjoin

```
SELECT CROSSJOIN( {Germany, France}
                  {Wein, Bier}) ON COLUMNS
      {Year.MEMBERS} on ROWS
FROM Sales
WHERE (Measures.Menge)
```

	Germany		France	
	Wein	Bier	Wein	Bier
1997				
1998				
...				

- Projektion zweier Dimensionen in eine
- Semantik: Alle möglichen Kombinationen
- Keine hierarchische Summierung

# Weitere Feature

---

- TOPN Queries

```
SELECT {Year.MEMBERS} ON COLUMNS  
       {TOPCOUNT(Country.MEMBERS, 5, Measures.Menge)} ON ROWS
```

- Auswahl von Members über Bedingungen

```
SELECT FILTER(Germany.CHILDREN,  
              ([2002], M.Menge) > ([2001], M.Menge)) ON COLUMNS  
          Month.MEMBER ON ROWS
```

- Named Sets und Calculated Members
- Zeitreihenoperationen, gleitende Durchschnitte
- ...

# MDX - Fazit

---

- Hohe Komplexität
- Mächtige Sprache
- Direkte Anlehnung an MDDM
- Könnte sich als Standard durchsetzen
  - Schnittstelle zwischen OLAP GUI und DB-Server
  - Unterstützt von MS, Microstrategy, Cognos, BusinessObjects, ...

# Inhalt dieser Vorlesung

---

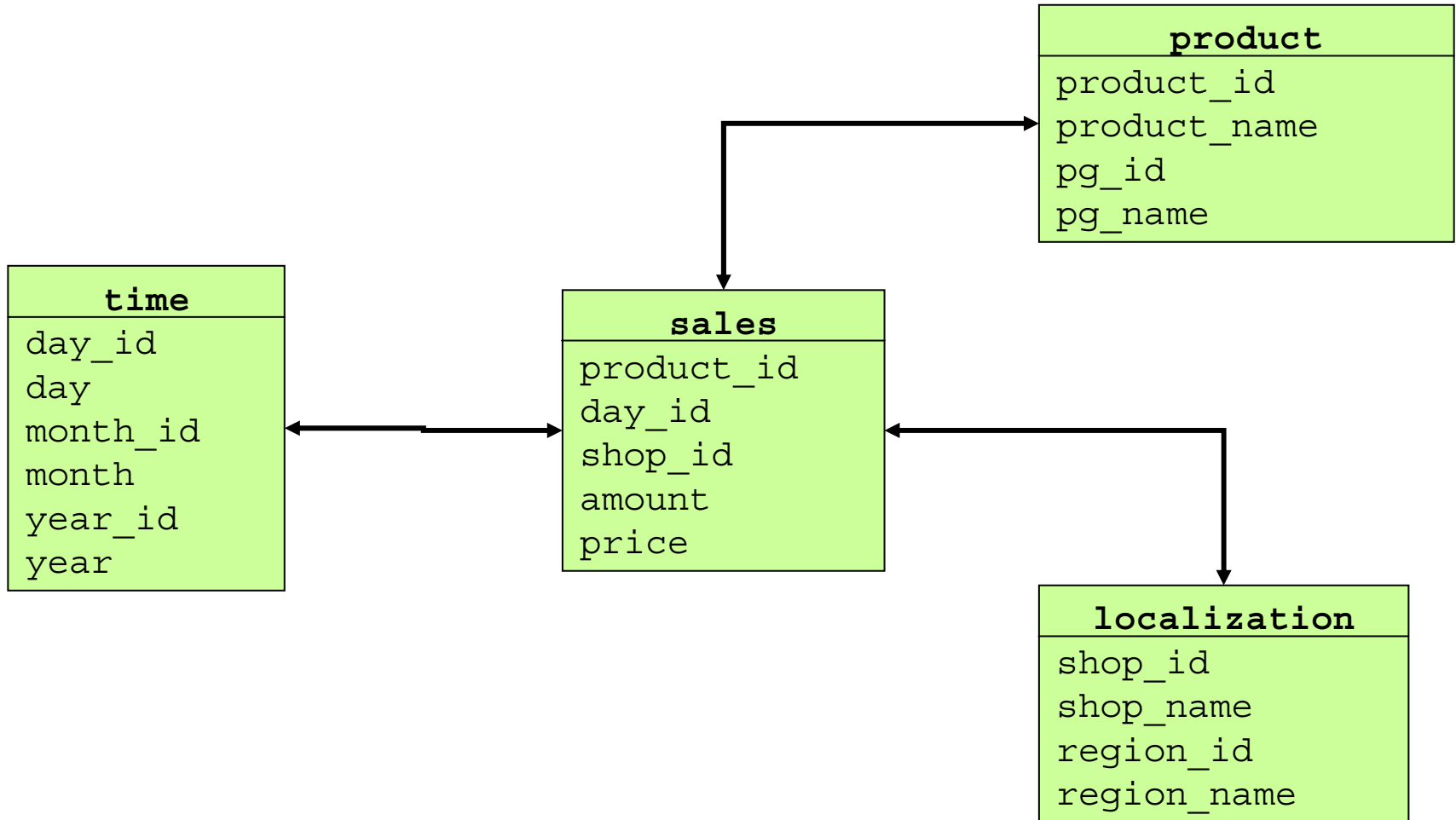
- OLAP Operationen
- MDX: Multidimensional Expressions
- SQL Erweiterungen
  - Rückblick: Gruppierung
  - OLAP: Hierarchische Aggregation
  - OLAP: SQL Analytical Functions

# SQL und OLAP

---

- Übersetzung MDDM in Star- oder Snowflake Schema
- Triviale Operationen
  - Auswahl (slice, dice): Joins und Selects
  - Verfeinerung (drill-down): Joins und Selects
- Einfache Operation
  - Aggregation um eine Stufe: Group-by
- Interessant
  - Hierarchische Aggregationen
  - Multidimensionale Aggregationen
  - Analytische Funktionen (gleitende Durchschnitte etc.)
- Ist alles in SQL-92 möglich, aber nur **kompliziert auszudrücken und ineffizient** in der Bearbeitung

# Beispiel Star-Schema



# Erinnerung: Semantik von GROUP-BY

```
SELECT    T.day_id, sum(amount*price) SU
FROM      sales S
WHERE     price>100
GROUP BY  T.day_id
HAVING    SU>0
ORDER BY  day_id
```

- Syntax
  - SELECT Klausel darf nur GROUP\_BY Ausdrücke, Konstante und Aggregatfunktionen enthalten
- Semantik
  - **Partitionierung** der Ergebnistupel nach unterschiedlichen Werten der GROUP-BY Attribute
  - Aggregation (der Measures) pro Partition



# Beispiel

Alle Verkäufe der Produktgruppe „Wein“ nach Monaten  
(was passiert?)

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

Scheitert: „... T.month is not a GROUP-BY expression ...“

- Funktionale Abhängigkeit T.month\_id->T.Month nicht bekannt
- (Erinnerung: „ATTRIBUTE ... DETERMINES“ in Oracle)

# Hierarchische Aggregation

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

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

- Summe nur für Tage (unterteilt nach Monaten/Jahren)
- Keine Summen pro Monat / pro Jahr
- Nicht in einer Query formulierbar
- Also?

1997	1	1	150
1997	1	2	130
1997	1	3	145
1997	1	4	122
...	...	...	...
1997	1	31	145
1997	2	1	133
1997	2	2	122
...	...	...	...
1997	3	10	180
1997	12	31	480
1998	1	1	240
...	...	...	...
2003	6	18	345

# Hierarchische Aggregation –2-

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

Benötigt UNION und eine Query pro Klassifikationsstufe

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

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

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

# Ergebnis

Tage

1.1.1997	150
2.1.1997	130
...	...
31.1.1997	145
1.2.1997	133
...	...
28.2.1997	480
1.3.1997	240
...	...
31.12.1998	345

Monate

1/1997	12000
2/1997	13000
...	...
12/1998	15600

Jahre

1997	123453
1998	143254

Schöner wäre

1997	1	1	150
...	...	...	...
1997	1	31	145
1997	1	-	12000
1997	2	1	480
...	...	...	...
1997	12	-	14000
1997	-	-	123453
1998	1	1	345
...	...	...	...
1998	-	-	143254
...	...	...	...

# ROLLUP Operator

---

- Herkömmliches SQL
  - Dimension mit  $k$  Stufen – Union von  $k$  Queries
  - $k$  Scans der Faktentabelle
    - Keine Optimierung wg. fehlender Multiple-Query Optimierung in kommerziellen RDBMS
  - Für die meisten Reports ungünstige Ergebnisreihenfolge
    - Und schwierig zu sortieren!
- SQL Erweiterung: **ROLLUP** Operator
  - Hierarchische Aggregation mit Zwischensummen
  - Summen werden durch „ALL“ als Wert repräsentiert

# ROLLUP Beispiel

```
SELECT    T.year_id, T.month_id, T.day_id, sum(...)
FROM      sales S, time T
WHERE     T.day_id = S.day_id
GROUP BY  ROLLUP(T.year_id, T.month_id, T.day_id)
```

1997	Jan	1	200
1997	Jan	...	
1997	Jan	31	300
1997	Jan	ALL	31.000
1997	Feb	...	
1997	March	ALL	450
1997	...	...	
1997	ALL	ALL	1.456.400
1998	Jan	1	100
1998	...	...	
1998	ALL	ALL	45.000
...	...	...	
ALL	ALL	ALL	12.445.750

# Bedingtes ROLLUP

---

- Bei großen Dimensionen will man **nicht in allen Klassifikationsknoten** aggregieren
  - Z.B.: Hierarchische Aggregation über Shop und Region, aber explizite Ausweisung nur für die Shops „Wedding“, „Mitte“, „Pankow“; Gesamtsumme pro Region soll erhalten bleiben
  - Selektion in der WHERE Klausel geht nicht, weil sonst die Gesamtsummen pro Region verfälscht werden
- Zwei Möglichkeiten
  - Verwendung von **HAVING und GROUPING**, um die überflüssigen Tupel aus dem Ergebnis zu filtern
    - Grouping: Identifikation der Summenzeilen
    - Erzeugt viele Tupel und wirft sie dann weg
  - Verwendung einer **CASE Anweisung** im ROLLUP Operator
    - Erzeugt nur die gewünschten Tupel
    - Benötigt prinzipiell gleich viele Scans, aber weniger Speicherplatz

# Bedingtes ROLLUP

Hierarchische Aggregation über Shop und Region, aber Werte nur für die Shops „Wedding“ und „Mitte“

```
SELECT      L.shop_id, L.region_id, sum(amount)
FROM        sales S, localization L
WHERE       S.shop_id = L.shop_id
GROUP BY   ROLLUP(L.region_id,
                  CASE WHEN L.shop_id IN (,Wedding`, ,Mitte`
                    THEN L.shop_ID
                    ELSE ,Others`
                  END))
```

Region_id	Shop_id	sum
Bayern	Others	...
Hessen	Others	...
Berlin	Wedding	...
Berlin	Mitte	...
Berlin	Others	...
Hamburg	Others	...
...	...	...
ALL	ALL	...

# Multidimensionale Aggregation

Verkäufe nach Produktgruppen und Jahren

	1998	1999	2000	Gesamt
Weine	15	17	13	45
Biere	10	15	11	36
Gesamt	25	32	24	81

- `sum() ... GROUP BY pg_id, year_id`
- `sum() ... GROUP BY pg_id`
- `sum() ... GROUP BY year_id`
- `sum()`

# Cube Operator

---

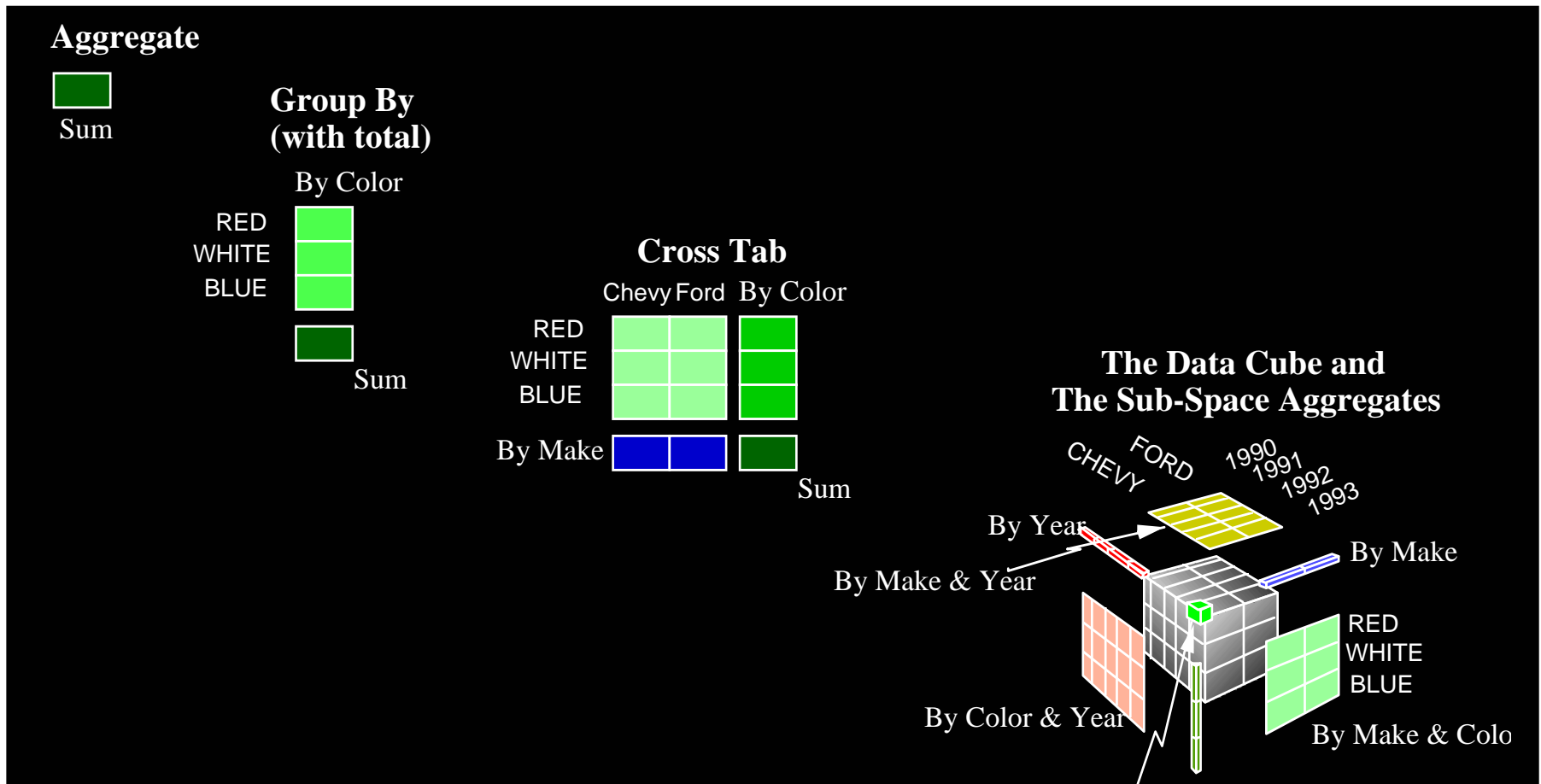
- d Dimensionen, jeweils eine Klassifikationsstufe
  - Jede Dimension kann in Gruppierung enthalten sein oder nicht
  - 2<sup>d</sup> Gruppierungsmöglichkeiten
- Herkömmliches SQL
  - Viel Schreibarbeit
  - Wahrscheinlich 2<sup>d</sup> Scans der Faktentabelle
- CUBE Operator
  - Berechnung der Summen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

# CUBE - Beispiel

```
SELECT  pg_id, shop_id, sum(amount*price)
FROM    sales S ...
GROUP BY CUBE (S.pg_id, S.shop_id, T.year_id)
```

Bier	Kreuzberg	ALL	...
Bier	Charlottenburg	ALL	...
Bier	ALL	1997	...
Wein	ALL	1998	...
ALL	Kreuzberg	1997	...
ALL	Charlottenburg	1998	...
Bier	ALL	ALL	...
Wein	ALL	ALL	...
ALL	ALL	1997	...
ALL	ALL	1998	...
ALL	Kreuzberg	ALL	...
ALL	Charlottenburg	ALL	...
ALL	ALL	ALL	...

# Cube-Operator: Veranschaulichung



Source: Gray et al., „Datacube“, Microsoft & IBM

# GROUPING SETS Operator

---

- CUBE
  - Alle Gruppierungskombinationen
  - Das sind sehr viele ...
- GROUPING SETS
  - Explizite Angabe der gewünschten Gruppierungen
  - Gruppierung wird für jeden SET einzeln ausgeführt aber kann mit einem SCAN der Faktentabelle berechnet werden
    - Wenn genügend Hauptspeicher vorhanden (später)
  - Äquivalent zu UNION einzelner GROUP BY

# GROUPING SETS – Beispiel

---

```
SELECT  pg_id, shop_id, sum(amount*price)
FROM    sales S
GROUP BY GROUPING SETS((S.pg_id), (S.shop_id))
```

Bier	ALL	300
Wein	ALL	450
ALL	Kreuzberg	350
ALL	Charlottenburg	400

# GROUPING SET und ROLLUP

---

- Wie kann man

```
SELECT    T.year_id, T.month_id, T.day_id, sum(...)
FROM      sales S, time T
WHERE     T.day_id = S.day_id
GROUP BY  ROLLUP(T.year_id, T.month_id, T.day_id)
```

- Mit GROUPING SET ausdrücken?

```
SELECT    T.year_id, T.month_id, T.day_id, sum(...)
FROM      sales S, time T
WHERE     T.day_id = S.day_id
GROUP BY  GROUPING SET((T.year_id),
                        (T.year_id, T.month_id),
                        (T.year_id, T.month_id, T.day_id))
```

# Inhalt dieser Vorlesung

---

- OLAP Operationen
- MDX: Multidimensional Expressions
- SQL Erweiterungen
  - Rückblick: Gruppierung
  - OLAP: Hierarchische Aggregation
  - OLAP: SQL Analytical Functions

# SQL Analytical Functions

---

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

# OVER() Klausel

---

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

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

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

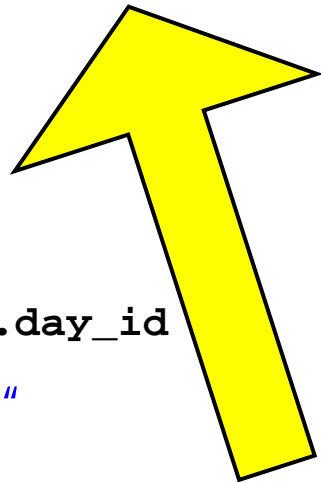
```
SELECT      S.day_id, sum(amount) AS day_sum,
            day_sum/ (sum(amount) OVER())
FROM        sales S
GROUP BY    S.day_id;
```

- OVER() ohne Parameter iteriert über alle Tupel

# Veranschaulichung

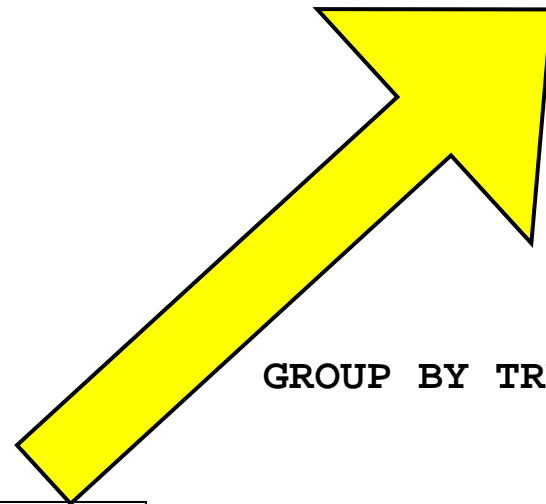
---

```
SELECT S.day_id, sum(amount) AS day_sum, day_sum/ (sum(amount) OVER())
```



GROUP BY s.day\_id

„Group-By Query“



GROUP BY TRUE

„Innere Query“

```
(S.day_id, sum(amount))  
FROM sales S
```

# Ergebnis

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

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

day_id	day_sum	all_sum	Ratio
1.1.1999	500	10.000	0.05
2.1.1999	500	10.000	0.05
3.1.1999	1.000	10.000	0.10
1.2.1999	1.500	10.000	0.15
2.2.199	1.500	10.000	0.15
1.1.2000	600	10.000	0.06
5.5.2000	400	10.000	0.04
1.10.2001	4.000	10.000	0.40

# OVER() mit lokaler Partitionierung

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

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

SQL Funktion  
YEAR()

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

Sommers

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

# Veranschaulichung

---

```
S.day_id, sum(amount), sum(amount) OVER(PARTITION BY YEAR(S.day_id))
```

GROUP BY S.day\_id

GROUP BY YEAR(day\_id)

```
(S.day_id, sum(amount))  
FROM sales S
```

# Unabhängige Partitionierung

---

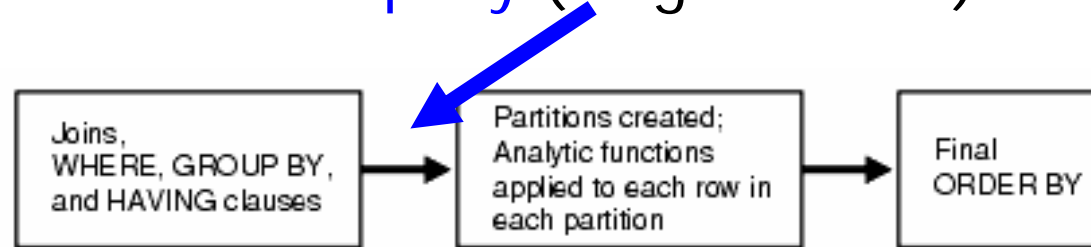
- Summe der Verkäufe eines Tages eines Shops im Vergleich zu den Verkäufen im Jahr und den Verkäufen im Shop

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

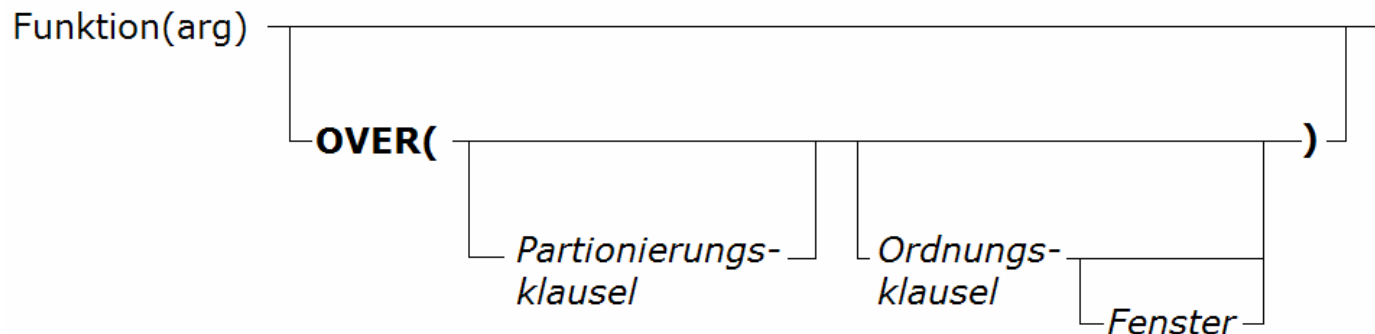
- Wie viele Scans braucht man?

# Ablauf der Auswertung

- OVER() berechnet **einen Wert pro Tupel des Tupelstroms der Restquery** (Eingabestrom)



- Wie – später
- Komplette Syntax



# Lokale Sortierung

---

- Durch **lokale Sortierung** mit ORDER BY kann die Reihenfolge der Tupel im inneren Strom pro OVER() Klausel bestimmt werden
- Die Aggregatfunktion iteriert bei einem ORDER BY nur über die Tupel zwischen dem ersten bis zum aktuellen Tupel (bzgl. der angegebenen Ordnung)
- Dadurch kann man **kumulierte Summen** erreichen
- Beispiel: Summe der Verkäufe pro Tag sowie die kumulierten Gesamtverkäufe nach Tagen, und die kumulierten Verkäufe im jeweiligen Monat nach Tagen

```
SELECT      S.day_id, sum(amount) AS day_sum,
            sum(amount) OVER(ORDER BY S.day_id) AS cum_sum,
            sum(amount) OVER(PARTITION BY S.month_id
                               ORDER BY S.day_id) AS cumm_sum
FROM        sales S,
GROUP BY   S.day_id;
```

# Ergebnis

```
SELECT      S.day_id, sum(amount) AS day_sum,  
            sum(amount) OVER(ORDER BY S.day_id) AS cum_sum,  
            sum(amount) OVER(PARTITION BY S.month_id  
                               ORDER BY S.day_id) AS cumm_sum  
FROM        sales S,  
GROUP BY   S.day_id;
```

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

day_id	Day_sum	Cum_sum	Cumm_sum
1.1.1999	500	500	500
2.1.1999	500	1.000	1.000
3.1.1999	1.000	2.000	2.000
1.2.1999	1.500	3.500	1.500
2.2.1999	1.500	5.000	3.000
1.1.2000	600	5.600	600
5.5.2000	400	6.000	400
1.10.2001	4.000	10.000	4.000

# Lokale Sortierung und Ranking

---

- RANK() gibt den **Rang des aktuellen Tuples** innerhalb einer Sortierung
  - Gleiche Werte ergeben den gleichen Rang und eine Lücke
  - DENSERANK() lässt keine Lücke
  - Weitere Funktionen: NTILE (Percentile), Behandlung von NULLs, ...
- Beispiel: Ausgabe aller Tagesverkäufe mit dem Rang des Tagesverkaufs im Verhältnis zu allen anderen Tagesverkäufen
  - Wir bauen erst einen View

```
CREATE VIEW daysum
SELECT      S.day_id, sum(amount) AS dsum,
FROM        sales S,
GROUP BY   S.day_id;
```

```
SELECT      D.day_id, D.dsum,
            denserank() OVER(ORDER BY dsum) AS rank
FROM        daysum D,
ORDER BY   S.day_id;
```

# Ergebnis

```
SELECT      D.day_id, D.dsum,  
            denserank() OVER(ORDER BY dsum) AS rank  
FROM        daysum D,  
ORDER BY    S.day_id;
```

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

day_id	Dsum	rank
1.1.1999	500	5
2.1.1999	500	5
3.1.1999	1.000	3
1.2.1999	1.500	2
2.2.199	1.500	2
1.1.2000	600	4
5.5.2000	400	6
1.10.2001	4.000	1

# Dynamische Fenster

---

- Der Vergleich der Werte des aktuellen Tuples mit den **lokalen Aggregaten** zieht immer folgende Tupel in die Aggregation mit ein
  - OVER() ohne Argument: Alle Tupel
  - OVER(PARTITION BY): Alle Tupel der gleichen Partition
  - OVER(ORDER BY): Alle Tupel zwischen Ordnungsbeginn und aktuellem Tupel
- Das **Fenster des Vergleichs** kann man auch explizit angeben
  - ROWS BETWEEN ... AND ...
  - Möglich jeweils
    - UNBOUND PRECEDING / FOLLOWING: Vom Anfang / bis zum Ende
    - K PRECEDING / FOLLOWING: Die k Tupel vorher / nachher
    - CURRENT ROW: aktuelles Tupel
- Damit kann man gleitende Durchschnitte bilden

# Beispiel

---

- Durchschnitt der Tagesverkäufe im gleitenden 3-Tagesfenster, innerhalb des Monats, und über die letzten 30 Tage hinweg

```
SELECT      D.day_id, D.dsum,
            AVG(amount) OVER(ORDER BY D.day_ID
                               ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS threedays,
            AVG(amount) OVER(PARTITION BY month(D.day_id)),
            AVG(amount) OVER(ORDER BY D.day_ID
                               ROWS BETWEEN 30 PRECEDING AND CURRENT ROW) AS prev_30,
FROM        daysum D,
ORDER BY   S.day_id;
```

- Weitere Fensterfunktionen
  - RANGE: Tupel-variable Fenstergrößen (z.B: Wochenende überspringen)
  - FIRST\_VALUE, LAST\_VALUE: Datenabhängige, feste Fenstergrenzen
- Und vieles, vieles mehr ...

# Fazit

---

- Operationen schachtelbar
  - GROUP BY ROLLUP( year, month, day, CUBE(pg\_id, shop\_id))
- SQL-Standard
  - Implementiert (mindestens) von Oracle, DB2, SQLServer
- Nicht unkomplizierte, aber sehr mächtige Erweiterungen
- Wesentliche Verbesserung
  - Kompaktere Queries
  - Theoretisch **deutliche Beschleunigung** durch weniger Scans
- **Geschickte Implementierung** nicht trivial
  - CUBE erzeugt massenweise Summen, die größtenteils voneinander abhängen
  - Kann man OVER() mit nur einem Scan implementieren?

# Literatur

---

- [Leh03] Lehner: „Datenbanktechnologie für Data Warehouse Systeme“, dpunkt.Verlag, 2003
- [GCB+97] Gray et al. „Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals“, Journal on Data Mining and Knowledge Discovery, 1997
- [Nol99] Nolan: „Introduction to Multidimensional Expressions (MDX)“, Microsoft Corp.
- [WZP03] Whitehorn, M., Zare, R. and Pasumansky, M. (2003). "Fast Track to MDX", Springer.