Data Warehousing und Data Mining

Sprachen für OLAP Operationen

Ulf Leser

Wissensmanagement in der Bioinformatik

Übersicht

- Konzeptionell: Modellierung und Sprachen
  - Architektur & Prozesse
  - Multidimensionale Modellierung
    - MDDM, MER
    - ROLAP und MOLAP
    - OLAP Operationen und Sprachen
  - Extraction, Transformation, Load (ETL)
    - Differential Snapshots
    - Transformations and types of heterogeneity
    - Bulk loading
- Umsetzung: Logische und physische Ebene
  - Indexstrukturen für DWH: Bitmap, Join-Indexe
  - Multidimensionale Indexstrukturen: Grid-File, kd-Tree
  - Optimierung: Star-Join, Partitionierung
  - Implementierung von OLAP Operationen
  - Column Stores, Main Memory, Map-Reduce
- Materialisierte Sichten
  - Auswahl
  - Query Rewriting
  - Aktualisierung
- Data Mining
  - Datenaufbereitung
  - Clustering: k Means, DBScan, hierarchisch
  - Klassifikation: kNN, Naive Bayes, Decision Trees
  - Assoziationsregeln

Inhalt dieser Vorlesung

- OLAP Operationen
- MDX: Multidimensional Expressions
- SQL Erweiterungen

OLAP Operationen

- OLAP Kern-Operation: Aggregation auf dem Cube
  - Roll-up, Drill-Down
  - Hierarchische Roll-Ups mit Aggregation auf allen Ebenen (Summe pro Tag, pro Monat, pro Jahr)
  - Aggregation über mehrere Dimensionen (Cross-Tabs): Verkäufe pro Marke und Jahr und Shop, Summen in allen Kombinationen
- Weitere OLAP Operationen
  - Gleitende Durchschnitte, attributlokale Vergleiche, Arbeiten auf Zeitreihen – Sequenzbasierte Operationen
  - Auch: „analytische Anfragen“

Sprachen für OLAP Operationen

- Ziel: Ökonomisches, aufgabengerechtes Design
  - Keine tief geschachtelten SQL Operationen
  - Verwendung von MDDM Konzepten und Begriffen
- Hauptsächlich zwei Ansätze
  - Multidimensional Expressions (MDX)
    - Basiert direkt auf MDDM Elementen: Cube, Dimension, Fakt, ...
    - Mapping auf ROLAP und MOLAP möglich (und vorhanden)
    - Von vielen Tools zur Kommunikation mit OLAP-Server benutzt
    - Unterstützt z.B. von Microsoft, Microstrategy, Cognos (IBM), BusinessObjects (SAP), Teradata, Pocahontas, Oracle, ...
    - Auf dem Vormarsch als de-facto Standard
  - Erweiterungen von SQL
    - Einführung spezieller Operationen (**ROLLUP**, **CUBE**, ...)
    - Ausführung auf Star-/ Snowflake-Schema

MDX

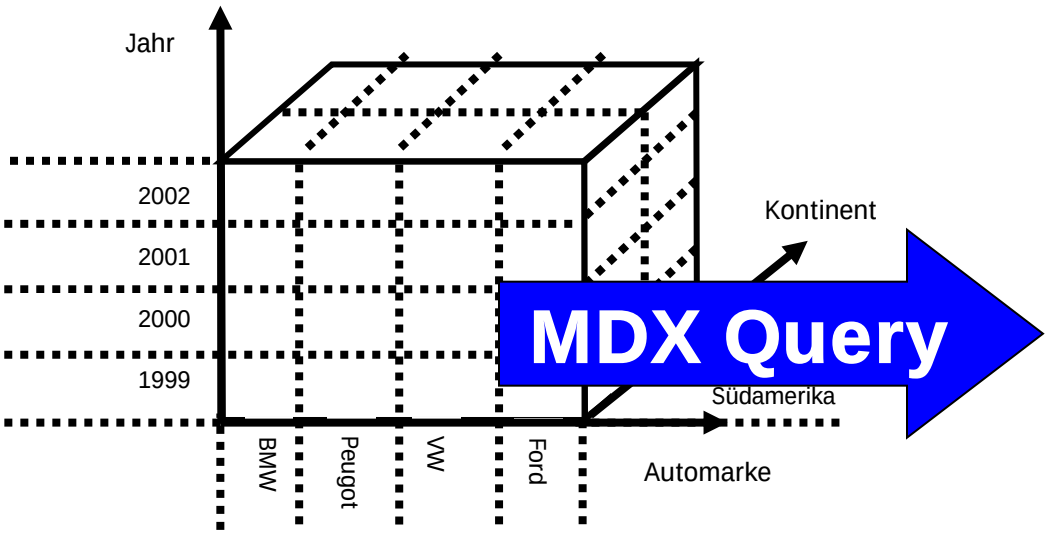
- MDX: Multidimensional Expressions
  - Ursprung: Microsoft‘s „OLE DB for OLAP“
- Eigene Anfragesprache
  - Definition ohne feste Semantik („by example“)
  - MDDM Konzepte als First-Class Elemente
  - SQL-artige Syntax
  - Sehr mächtig und komplex
- Erzeugung der Objekte (DDL) muss anderweitig erfolgen
  - Zugriff über Metadatenkataloge

MDX Elemente

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

Ausgabe

- MDX Anfragen erzeugen mehrdimensionale (gestapelte) Reports

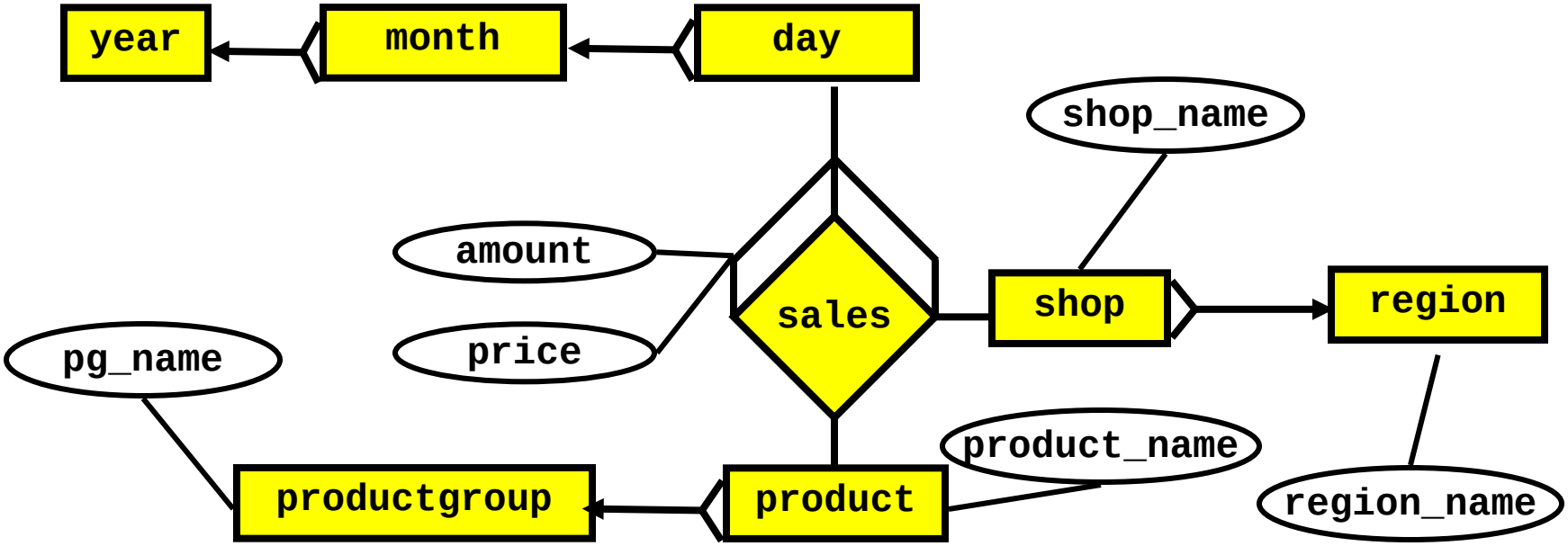


1997					

1998					

1999					

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

Struktur einer MDX Query

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

- Dimensions (**SELECT**)
  - Angabe der darzustellenden Achsen der Ausgabetabelle(n)
    - **ON COLUMNS, ROWS, PAGES, CHAPTER, ...**
  - Achsenspezifikation muss eine Menge von Members definieren
    - Achsenbeschriftungen (geben auch Aggregationslevel vor)
    - Explizit als Menge oder implizit durch Berechnungsfunktionen
- Cube (**FROM**): Basis-Cube für die Anfrage
- Slicer (**WHERE**)
  - Auswahl des Fakts
  - Einschränkungen auf Members (Knoten)

Navigation in den Hierarchien

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

Navigationsfunktionen

- Unterstützung eines navigierenden Zugriffs (GUI)
- **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				
...				

- Schachtelung zweier Dimensionen
- Kartesisches Produkt der Mengen von Klassifikationsknoten

Weitere Achsenauswahl-Features

- TOP-N 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
- Aggregationsfunktion AVG, MAX, ...: „AS“ Klausel
- ...

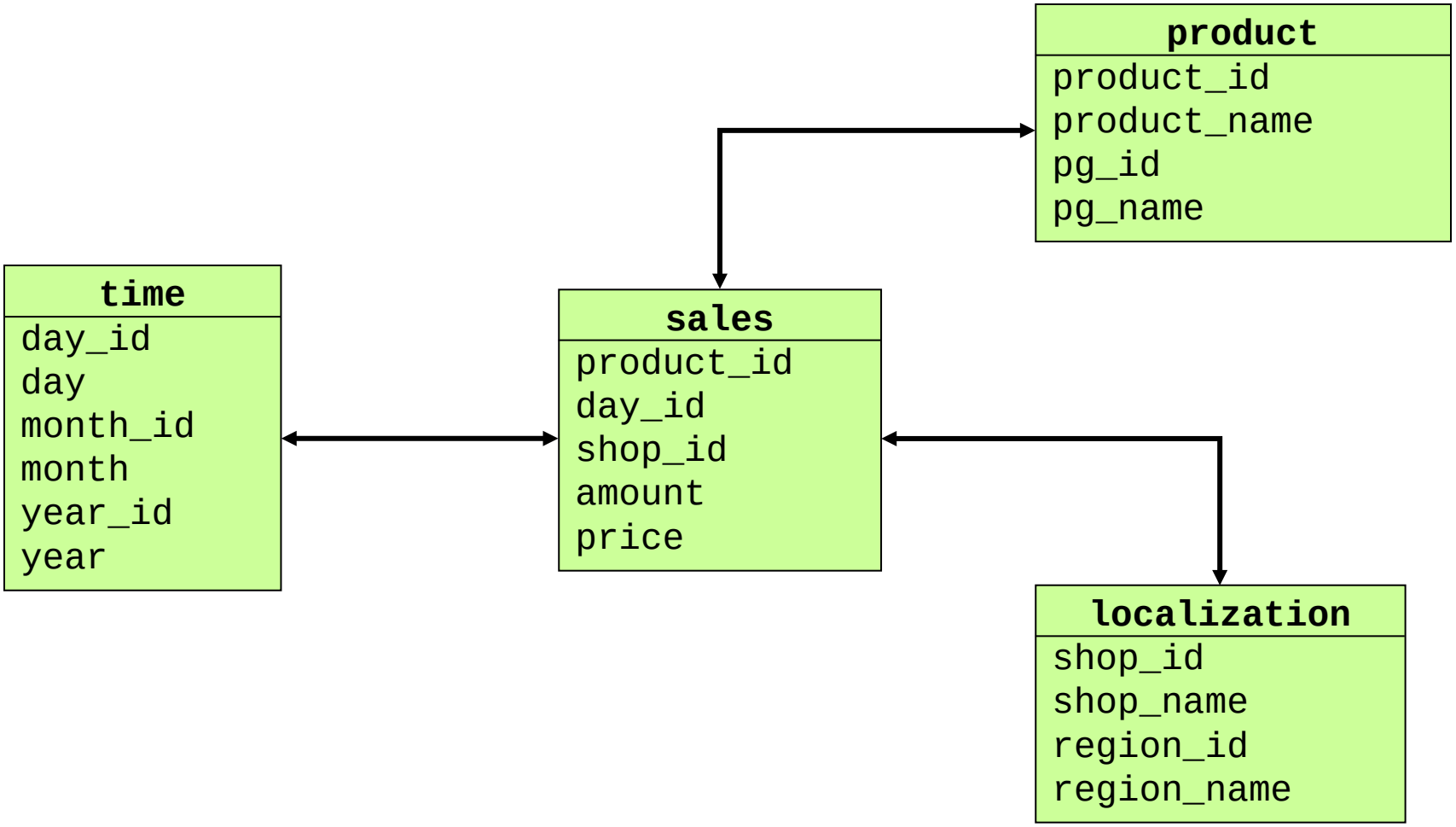
Inhalt dieser Vorlesung

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

SQL und OLAP

- Annahme: Star- oder Snowflake Schema
- Einfache Operationen
  - Auswahl (slice, dice): Joins und Selects
  - Verfeinerung (drill-down): Joins und Selects
  - Aggregation auf eine feste Granularität: Group-by und Agg-Fkt
- Schwieriger
  - Gleichzeitige hierarchische Aggregationen
  - Gleichzeitige multidimensionale Aggregationen
  - Analytische Funktionen (gleitende Durchschnitte etc.)
  - Alles auch 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
```

- SELECT Klausel darf nur GROUP_BY Ausdrücke, Konstante und Aggregatsfunktionen enthalten
- Semantik
  - Partitionierung der Ergebnistupel nach GROUP-BY Attribut
  - 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
```

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

- Summen nur für Tage
- Keine Summen pro Monat / pro Jahr
- SQL92: nicht mit einer SFW-Query möglich

Hierarchische Aggregation –2-

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

Benötigt UNION und eine Query pro Klassifikationsstufe

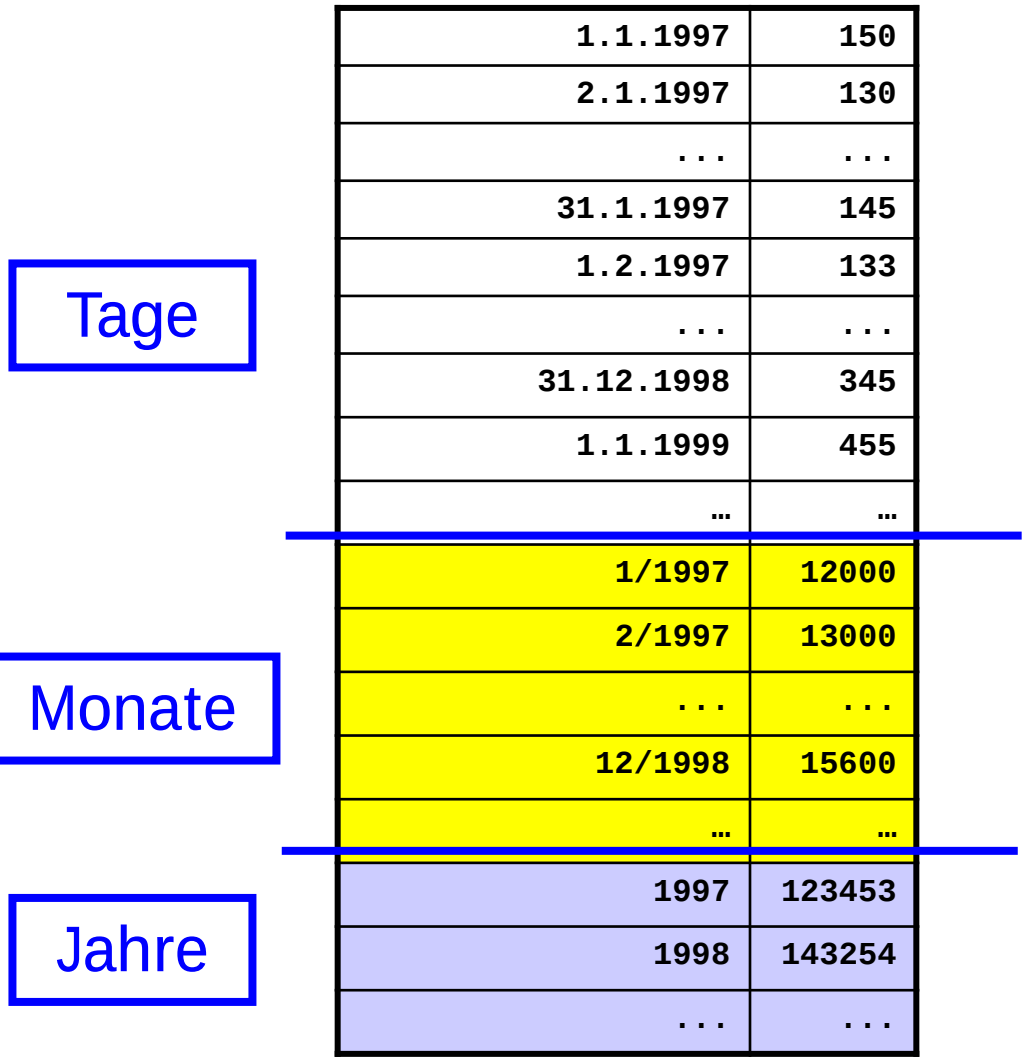
```
SELECT   T.day_id, sum(amount*price)
FROM     sales S, product P
WHERE    P.pg_name=„Wein“ AND
         P.product_id = S.product_id
GROUP BY T.
```

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

GROUP BY T
```

```
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
Monate
Jahre
1.1.1997 150
2.1.1997 130
... ...
31.1.1997 145
1.2.1997 133
... ...
31.12.1998 345
1.1.1999 455
… …
1/1997 12000
2/1997 13000
... ...
12/1998 15600
… …
1997 123453
1998 143254
... ...

OLAP-Operator: ROLLUP

- Herkömmliches SQL für hierarchische ROLLUP
  - Dimension mit k Stufen – Union von k Queries
  - k Scans der Faktentabelle
    - Typischerweise keine Optimierung wg. fehlender Multiple-Query Optimierung in den meisten RDBMS
- OLAP/SQL Erweiterung: ROLLUP Operator
  - Hierarchische Aggregation mit Summen auf allen Stufen
  - Summen werden durch „ALL“ als Wert repräsentiert
    - In Oracle ist es NULL statt ALL
  - Identifizierung über GROUPING-Funktion

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

- Man will nicht immer alle Klassifikationsknoten sehen
  - Z.B.: Hierarchische Aggregation über Shop und Region, aber explizite Ausweisung nur für die Shops „Wedding“ und „Mitte“; Gesamtsumme pro Region soll erhalten bleiben
  - Selektion in der WHERE Klausel geht nicht, weil sonst die Gesamtsummen pro Region verfälscht werden
- Zwei Möglichkeiten
  - HAVING, um unerwünschte Tupel zu filtern
    - Unterdrückt Tupel in der Ausgabe
  - Verwendung einer CASE Anweisung im ROLLUP Operator
    - Ermöglicht Zusammenfassen von Gruppen nach komplexen Bedingungen
    - Erzeugt neue Tupel, die andere zusammenfassen können

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	…
Bayern	All	…
Berlin	Wedding	…
Berlin	Mitte	…
Berlin	Others	…
Berlin	ALL	…
...	...	

Könnte man noch mit HAVING unterdrücken

Multidimensionale Aggregation

Verkäufe nach Produktgruppen und Jahren

	1998	1999	2000	Gesamt
Limonaden	15	17	13	45
Mineralwasser	10	15	11	36
Gesamt	25	32	24	81

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

OLAP-Operator: Cube

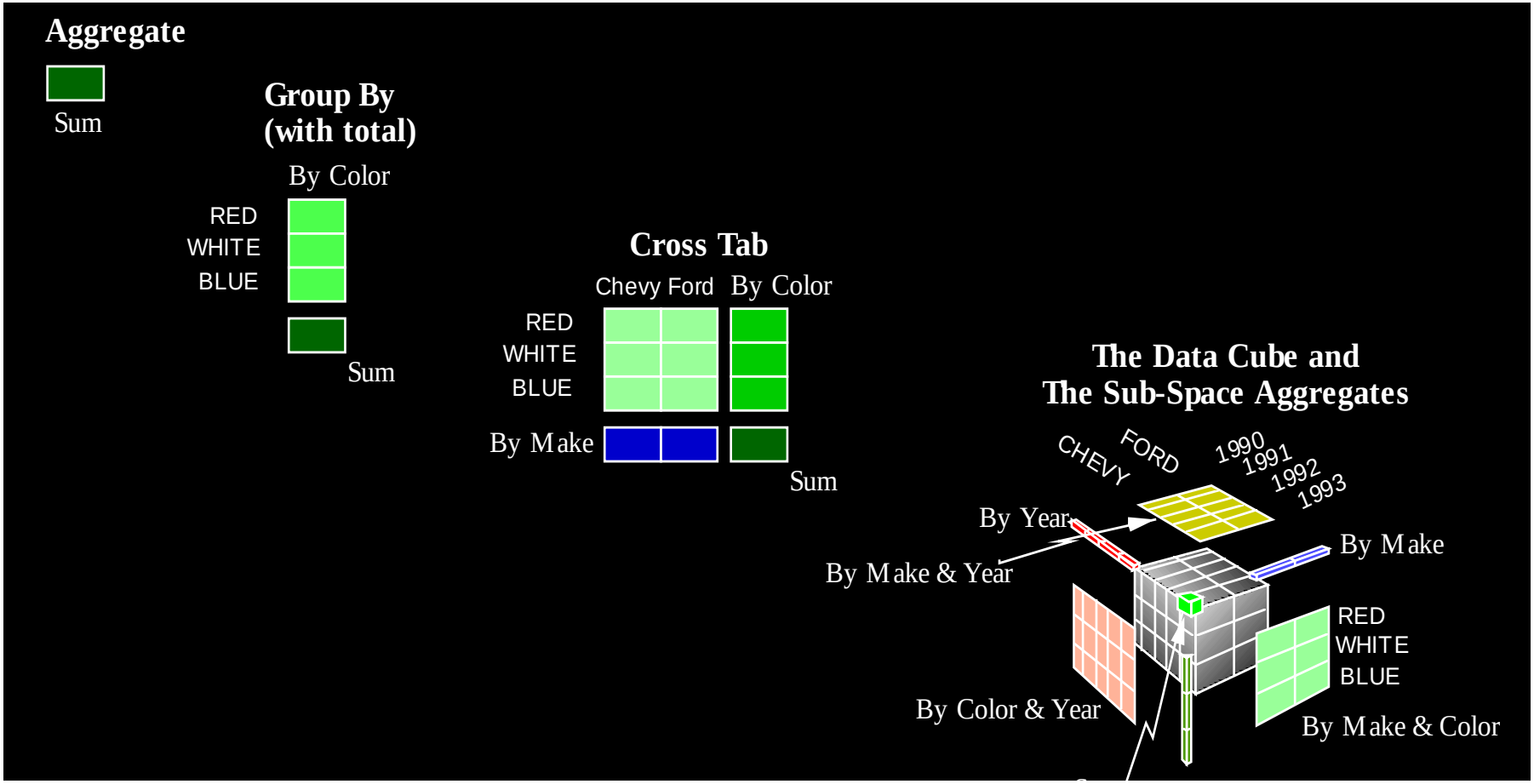
- d Dimensionen, jeweils eine Klassifikationsstufe
  - Jede Dimension kann in Gruppierung enthalten sein oder nicht
  - 2^d Gruppierungsmöglichkeiten
- Herkömmliches SQL
  - Viel Schreibarbeit
  - Wahrscheinlich 2^d Scans der Faktentabelle
- OLAP/SQL Erweiterung: CUBE Operator
  - Berechnung der Summen aller Kombinationen der Gruppierungsattribute (Klassifikationsstufen)

CUBE – Beispiel

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

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

OLAP-Operator: GROUPING SETS

- CUBE
  - Alle Gruppierungskombinationen
  - Das können sehr viele sein
- GROUPING SETS
  - Explizite Angabe der gewünschten Gruppierungsattribut-Mengen
  - Gruppierung wird für jede Attributmenge einzeln ausgeführt aber kann i.d.R. mit einem SCAN berechnet werden
    - Wenn genügend Hauptspeicher vorhanden ist (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 SETS 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 SETS 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 SETS((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

- Ziel: Flexiblere Aggregate und Rankings
  - Summe Verkäufe eines Tages zusammen mit der Summe Verkäufe des Monats in einer Zeile („ % von „)
  - Rang der Summe Verkäufe eines Monats im Jahr über alle Jahre
  - Vergleich der Summe Verkäufe eines Monats zum gleitenden Dreimonatsmittel
- Analytical Functions (auch „Window functions“)
  - Erscheinen in der SELECT Klausel
  - Berechnen Werte, die von neuen, in der SELECT Klausel angegeben Partitionierungen / Sortierungen abhängen können
  - Damit kann man unabhängige Gruppierungen in einer Zeile des Ergebnisses verwenden
    - Keine neuen Tupel, sondern neue Attribute
    - ANSI-SQL (92) kann das nicht

Einschub: Spaltenaliase

- Folgendes ist in SQL nicht erlaubt

```
SELECT foo as A, bar as B, A/B
FROM table;
```

  - Verwendung von Spaltenaliasen in derselben Query
  - Wir werden das aber machen, um Platz zu sparen
    - Wir werden auch immer schön sortiere Ergebnisse ausgeben, uns aber die dazu notwendige ORDER BY Klausel sparen
  - Wie würde es korrekterweise gehen?

Wie kann das in korrektem SQL aussehen?

- 1. Ausformulieren (anstrengend, ggf. doppelte Berechnung)

```
SELECT foo as A, bar as B, foo/bar
FROM table;
```

- 2. View anlegen (anstrengend)

```
CREATE VIEW myview AS
SELECT foo as A, bar as B
FROM table;
SELECT A, B, A/B
FROM myview;
```

- 3. Geschachtelte Anfrage (Umständlich, Optimierer?)

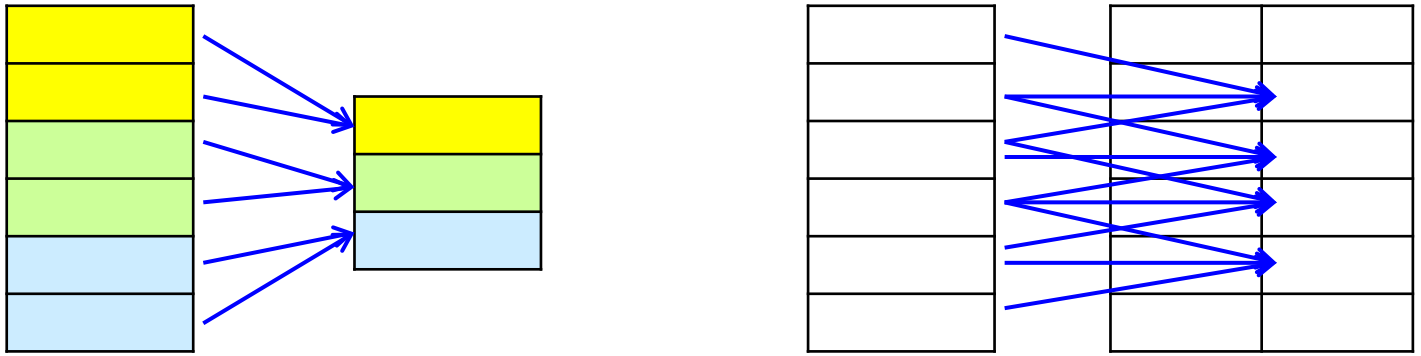
```
SELECT A, B, A/B
FROM (SELECT foo as A, bar as B
      FROM table) T;
```

- 4. Tabellenfunktion (Overkill, Optimierung)

```
SELECT A, B, A/B
FROM myTableFunction(table);
```

Grundprinzip Analytical Functions

- SQL erzeugt einen Tupelstrom
  - SELECT kann nur auf Werte in aktuellem Tupel zugreifen
  - GROUP-BY verkleinert den Tupelstrom (ein Tupel pro Partition)
  - Es gibt nur eine disjunkte Partitionierung pro Tupelstrom
- Eine Window Function berechnet einen zusätzlichen Wert pro Tupel und hat dazu Zugriff auf ein zu definierendes „Window“ um das aktuelle Tupel



Syntax

- **SELECT …, function(arg1, …, argn) OVER (**
  [PARTITION BY <…>]
  [ORDER BY <…>]
  [window_clause])
  - OVER: Indikator für Window Function
  - PARTITION BY: Partitioniert den gesamten Ergebnisstrom nach einem Attribut und selektiert implizit alle Tupel der Partition, in der das aktuelle Tupel ist, als Window
  - ORDER BY: Sortierung der Werte im Window
  - WINDOW_CLAUSE: Explizite Angabe der Fenstergrenzen
- Verschiedene Windowfunktionen in einer Query können unterschiedliche Partitionen / Sortierungen verwenden

Beispiel

- Summe Verkäufe eines Tages im Verhältnis zu den Gesamtverkäufen – in SQL

```
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, potentiell ineffizient (viele Scans von sales)
- Besser: over() Klausel
  - OVER() ohne Partitionierung selektiert alle Tupel als Window

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

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 Partitionierung

- 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;
```

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	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

Illustration

1. Basistabelle: SALES

day_id	Prod_id	Shop_id	T_id	li_sale
1.1.2017	5	3	1	20
1.1.2017	6	3	1	10
2.1.2018	5	2	2	15
3.3.2019	5	2	2	10
4.4.2019	6	7	1	13

day_id	day_sum
1.1.2017	30
2.1.2018	15
3.3.2019	10
4.4.2019	13

2. Primärergebnis: GROUP-BY

day_id	day_sum	Window Func
1.1.2017	30	30/30
2.1.2018	15	15/15
3.3.2019	10	10/23
4.4.2019	13	13/23

4. Erweitertes Ergebnis: Window Function

day_id	day_sum
1.1.2017	30
2.1.2018	15
3.3.2019	10
4.4.2019	13

3. Unabhängiger Scan+Partition

Unabhängige Partitionierung

- Verkäufe eines Tages eines Shops im Vergleich zu den Verkäufen im Jahr und zu 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?
- Latenz – wann kann man erste Tupel ausgeben?
- Diverse Optimierungsmöglichkeiten

Selbstversuch

day_id	Prod_id	Shop_id	T_id	li_sale
1.1.2017	5	3	1	20
1.1.2017	6	3	1	10
1.1.2017	5	2	2	15
1.1.2017	5	2	2	10

- Verkäufe nach Produkt_ID und Transaction_ID (Bon-genau), wobei pro T_ID ausgegeben wird, wie viel Prozent des Gesamtkaufs (des Bon) auf diese Produktgruppe entfallen ist

Prod_id	T_ID	Sum	Percent
5	1	20	66%
6	1	10	33%
5	2	25	100%

Mit Analytical Functions

day_id	Prod_id	Shop_id	T_id	li_sale
1.1.2017	5	3	1	20
1.1.2017	6	3	1	10
1.1.2017	5	2	2	15
1.1.2017	5	2	2	10

Prod_id	T_ID	Sum	Percent
5	1	20	66%
6	1	10	33%
5	2	25	100%

Erst mal nur die Summen

```
SELECT    S.prod_ID, S.T_ID, sum(li_sale)
FROM      sales S
GROUP BY  S.prod_ID, S.T_ID;
```

Dann die %-Angabe dazu

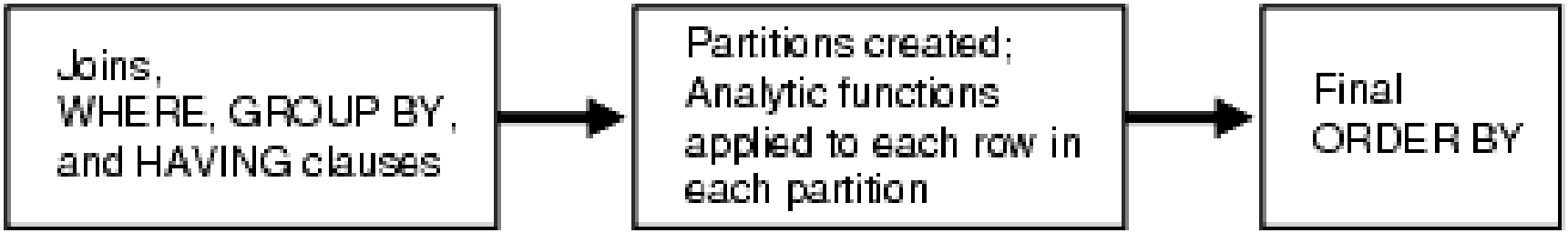
- Pro t_id aufsummieren
- %-bilden

```
SELECT    S.prod_ID, S.T_ID, sum(li_sale) su,
          sum(li_sale) OVER (PARTITION BY T_ID) sut,
          su/sut*100
FROM      sales S
GROUP BY  S.prod_ID, S.T_ID;
```

Selektiert implizit die „richtige“ (aktuelle) Partition

Genaue Reihenfolge

- Window Functions werden nach der Core-Query (Joins, GROUP-BY, HAVING), aber vor einem finalen ORDER BY ausgeführt
  - Können aber ihre eigene Sortierung vornehmen
- Dürfen nur in SELECT und ORDER BY vorkommen
- Können in geschachtelten Anfragen verwendet werden



Sortierung und Ranking

- ORDER BY nach OVER: Bestimmung der Reihenfolge der Tupel im Window
- Anwendung mit RANK(): Rang des aktuellen Tuples
  - RANK: Gleiche Werte erhalten gleichen Rang, dann Lücke
  - Weitere Funktionen: DENSERANK(), NTILE (Percentile), ...
- Beispiel: Aller Verkäufe mit dem Rang des Verkaufs am aktuellen Tag

```
SELECT    S.sale,
          rank() OVER (PARTITION BY day_id ORDER BY sale)
FROM      sales S;
```

Fenster: Nur Tupel des aktuellen Tags

Diese sortieren

Sortierung und Ranking

- Alle Tagesverkäufe mit dem Rang des Tagesverkaufs im Verhältnis zu allen anderen Tagesverkäufen
  - Wir müssen erst über die Tage summieren
  - Variante 1: Wir bauen 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,
           rank() OVER(ORDER BY dsum)
FROM       daysum D;
```

Sortierung und Ranking

- Alle Tagesverkäufe mit dem Rang des Tagesverkaufs im Verhältnis zu allen anderen Tagesverkäufen
  - Wir müssen erst über die Tage summieren
  - Variante 2: Inline-Views (geschachtelte Queries)

```
SELECT    D.day_id, D.dsum,
          denserank() OVER(ORDER BY dsum)
FROM      (SELECT   S.day_id, sum(amount) AS dsum,
           FROM     sales S,
           GROUP BY S.day_id);
```

Sortierung und Ranking

- Alle Tagesverkäufe mit dem Rang des Tagesverkaufs im Verhältnis zu allen anderen Tagesverkäufen
  - Wir müssen erst über die Tage summieren
  - Variante 3: WITH-Klausel
    - „Query subfactoring“
    - Weniger Geschachtel, übersichtlicher

```
WITH
  daysum as (
    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)
FROM   daysum D;
```

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

Kumulierte Summen

- Eine Aggregatfunktion vor OVER aggregiert bei einem ORDER BY nur über die Tupel zwischen dem ersten und dem aktuellen Tupel im Window
- 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
ORDER BY  S.day_id;
```

Aggfkt ohne OVER – auf dem gruppierten inneren Tupelstrom

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

Selbstversuch

day_id	Prod_id	Shop_id	T_id	li_sale
1.1.2017	5	3	1	20
1.1.2017	6	3	1	10
1.1.2017	5	2	2	15
1.1.2017	5	2	2	10
1.1.2017	5	2	3	8
1.1.2017	5	2	3	10
1.1.2017	5	2	3	15

- Alle verschiedenen Einzelverkaufswerte sowie deren Häufigkeit und die akkumulierte Häufigkeit
  - „70% unserer Verkäufe sind Produkte, die unter 1,99 kosten“

Li_sale	Count	Acc_count
8	1	1
10	3	4
15	2	6
20	1	7

Mit Analytical Functions

Day_id	pid	S_ip	T_id	Li_sale
1.1.2017	5	3	1	20
1.1.2017	6	3	1	10
1.1.2017	5	2	2	15
1.1.2017	5	2	2	10
…				

Li_sale	Count	Acc_count
8	1	1
10	3	4
15	2	6
20	1	7

Alle verschiedenen Verkaufswerte

```
SELECT    S.li_sale
FROM      sales S
GROUP BY  S.li_sale;
```

Häufigkeit zählen

```
SELECT    S.li_sale, count(*)
FROM      sales S
GROUP BY  S.li_sale;
```

Richtig sortieren und akkumulieren

```
SELECT    S.li_sale, count(*),
          count(*) OVER(ORDER BY S.li_sale ASC)
FROM      sales S
GROUP BY  S.li_sale;
```

Explizite Fenster

- Implizite Window Definitionen
  - OVER() ohne Argument: Alle Tupel
  - OVER(PARTITION BY): Alle Tupel der gleichen Partition
  - OVER(ORDER BY): Alle Tupel zw. erstem und aktuellem Tupel
- Windowsgrenzen 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 z.B. gleitende Durchschnitte bilden

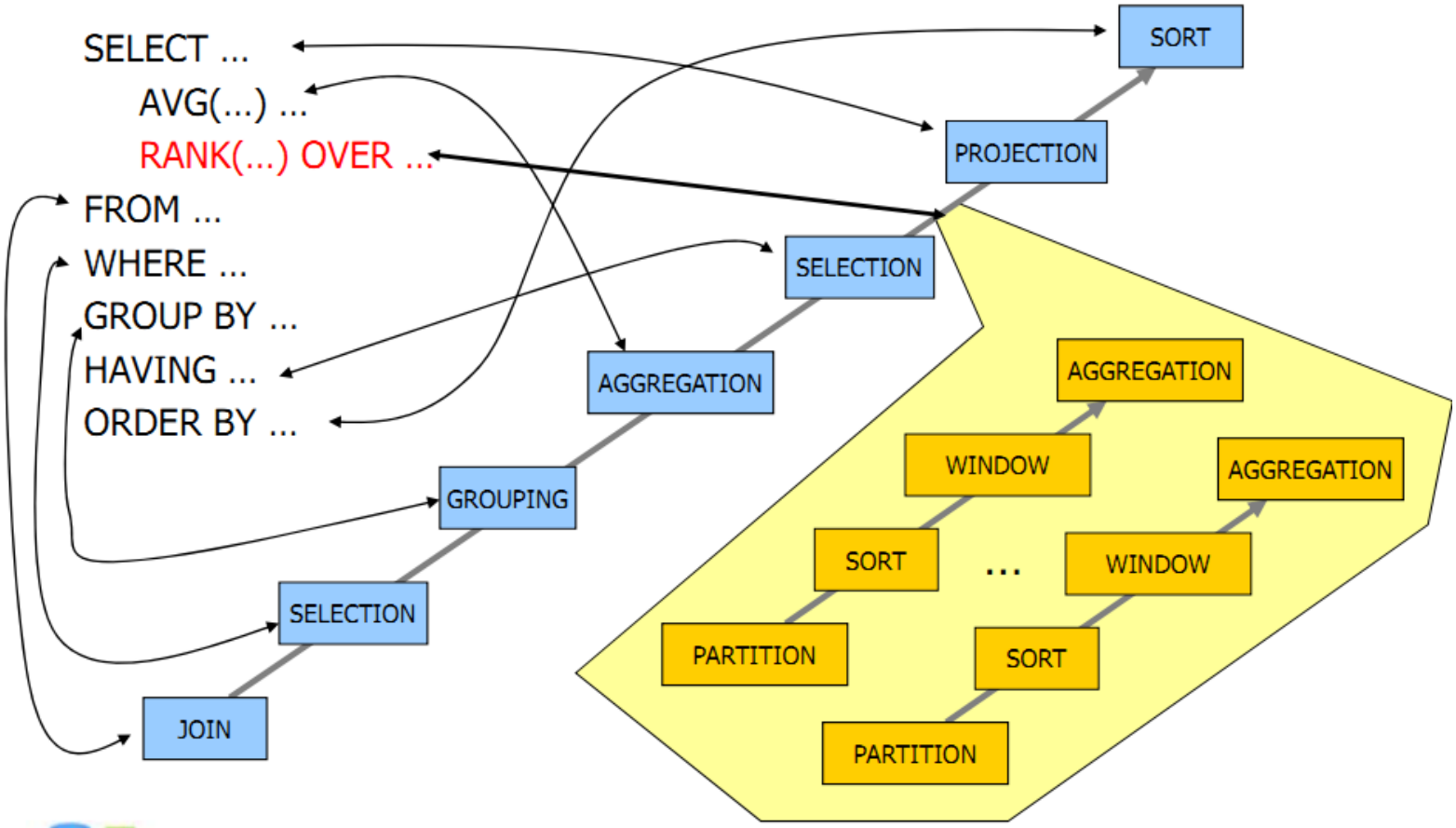
Beispiel

- 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: Variable Fenstergrößen (z.B.: Wochenende ignorieren)
  - FIRST_VALUE, LAST_VALUE: Feste Fenstergrenzen

Es wird komplizierter



SQL-Erweiterungen: Fazit

- Erweiterungen für
  - Hierarchische, multidimensionale Aggregate
  - Komplexe Vergleichsoperationen über unabhängigen Aggregaten
- Verbesserung
  - Kompaktere Queries
  - Gibt dem Optimierer mehr Wissen
    - Und idR deutliche Beschleunigung durch weniger Scans
- Geschickte Implementierung nicht trivial
  - CUBE erzeugt massenweise Summen, die größtenteils voneinander abhängen
  - Wann kann man OVER() mit einem Scan implementieren?

Literatur

- Lehner: „Datenbanktechnologie für Data Warehouse Systeme“, dpunkt.Verlag, 2003
- 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
- Whitehorn, M., Zare, R. and Pasumansky, M. (2003). "Fast Track to MDX", Springer.

Selbsttest

- Was sind typische OLAP Operationen?
- Erklären Sie den Grundaufbau einer MDX Query
- Was war der Unterschied zwischen Stapeln und Schachteln bei der 2D-Projektion hochdimensionaler Daten? Welchen Weg geht MDX?
- Formulieren Sie eine SQL Query mit einer Window-Function, die ...
- Welche Arten gibt es, Windows bei Window-Functions zu spezifizieren?
- Wann wird eine Window-Function im Lebenszyklus einer Query ausgewertet? Welche Konsequenzen hat das?