9 Hierarchien von Dimensionen

Wie vorher erwähnt werden Dimensionen so aufgebaut, dass sie erlauben, Messwerte mit verschiedenen „Auflösungen" darzustellen. Diese Auflösungen werden im DWH Jargon als Hierarchie bezeichnet. Die Hierarchie ordnet die verschiedenen Dimensionsauflösungen in einer logischen Reihenfolge.

Figur 3 zeigt drei Beispiele von Dimensionen. Für die Dimension „Ort" ist Stadt in Kanton enthalten, und Kanton ist in Land enthalten. Für die Zeit Dimension (die übrigens meistens die wichtigste Dimension ist) ist Woche aber nicht im Monat enthalten, deshalb gibt es zwei „Branchen".

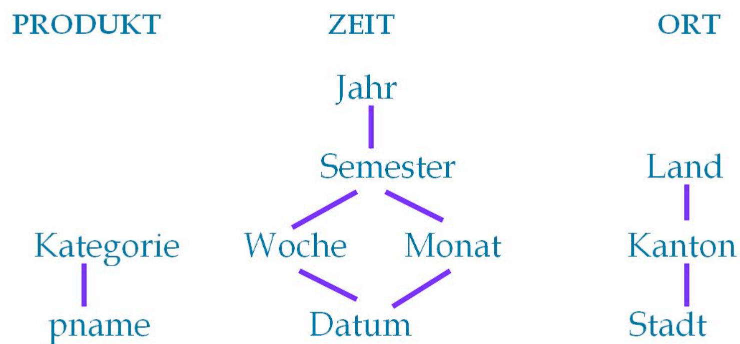

Figure 1: Beispiel von Dimensionshierarchien

10 OLAP Abfragen und typische Operationen auf DWHs

- Beeinflusst von SQL und von Spreadsheets
- Eine gewöhnliche Operation ist, die Messwerte über eine oder mehrere Dimensionen aufzusummieren [Aggregate]:
  - Finde den gesamten Verkauf [verkauf].
  - Finde den gesamten Verkauf von einem bestimmten Produkt [pname] für jede Stadt [stadt], oder für jeden Kanton [kanton]
  - Finde die 5 meist verkauften Produkte (geordnet mit gesamten Verkauf).
- Roll-up: „Wandern“ auf einer bestimmten Dimension von meist detailliert zu meist zusammengefasst.
  - Z.B: finde den gesamten Verkauf von pname per Stadt, dann finde den gesamten Verkauf per Kanton
- Drill-down: das Gegenteil von Roll-up.
  - Z.B., Aus der gesamten Verkauf per Kanton einen Drill-down ausführen, um den gesamten Verkauf per Stadt zu ansehen.
  - Z.B. Man kann auch auf einer anderen Dimension den Drill-down führen, z.B. um den gesamten Verkauf per Produkt zu berechnen.
- Pivoting: Aufsummieren auf ausgewählte Dimensionen.
  - Z.B. Pivoting auf Ort und Zeit ergibt diese Kreuztabulation [cross-tabulation]
- Slicing and Dicing: Gleichheits- bzw. Bereichsselektion auf einer oder mehreren Dimensionen.

Es ist zu merken, dass Roll-up, Drill-down, usw. neue Namen für alte Operationen sind. Die Namen wurde eher aus Marktspezifische Gründe eingeführt, aber repräsentieren Operationen die schon lange in der Datenbankwelt angewendet worden sind.

11 OLAP mit SQL Abfragen

Die Kreuztabulation, die das Pivoting von der vorherige Abbildung entspricht, wird mit folgenden SQL-Abfragen erhalten:

```
SELECT SUM(S.verkauf)
FROM verkauf S, Zeit T, Ort L
WHERE S.zeit_id=T.zeit_id AND S.ort_id=L.ort_id
GROUP BY T.jahr, L.kanton
---

SELECT SUM(S.verkauf)
FROM verkauf S, Zeit T
WHERE S.zeit_id=T.zeit_id
GROUP BY T.jahr
---
SELECT SUM(S.verkauf)
FROM verkauf S, Ort L
WHERE S.ort_id=L.ort_id
GROUP BY L.kanton
---
SELECT SUM(S.verkauf)
FROM verkauf S, Ort L
WHERE S.ort_id=L.ort_id
---
```

12 Der CUBE Operator

Wir können das vorherige Beispiel verallgemeinern: mit k Dimensionen haben wir 2^k mögliche GROUP BY Abfragen, die generiert sein können mit pivotieren auf eine Untermenge von Dokumente.

Ist gleich den Roll-up von Verkauf für alle 8 Untermengen von der Menge {p_id, ort_id, zeit_id}; jede roll-up entspricht eine SQL-Abfrage von der Form:

```
SELECT T.jahr, L.kanton, SUM(S.verkauf)
FROM Verkauf S, Ort L, Zeit T
WHERE S.zeit_id = T.zeit_id and S.loc_id = L.loc_id
GROUP BY CUBE (T.jahr, L.kanton)
```

12.1 CUBE Beispiel mit Oracle SQL

```
SELECT channels.channel_desc, countries.country_iso_code,
     TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM      sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND
     sales.cust_id=customers.cust_id AND
     sales.channel_id= channels.channel_id AND
     channels.channel_desc IN  ('Direct Sales', 'Internet')
AND
     times.calendar_month_desc='2000-09'  AND
     customers.country_id=countries.country_id  AND
     countries.country_iso_code IN ('US','FR')
GROUP BY CUBE(channels.channel_desc,
countries.country_iso_code);
```

Ergibt:

```
CHANNEL_DESC         CO   SALES$
-------------------- --   --------------
                          833,224
                     FR   70,799
                     US   762,425
Internet                  133,821
Internet             FR   9,597
Internet             US   124,224
Direct Sales              699,403
Direct Sales         FR   61,202
Direct Sales         US   638,201
```

Rollup ist eine andere CUBE-verwandte Abfrage auf Oracle. Es erlaubt, nur eine Untermenge von CUBE-Abfragen auszuführen:

```
SELECT channel_desc, calendar_month_desc,
countries.country_iso_code,
     TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM      sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND
sales.cust_id=customers.cust_id AND
     sales.channel_id= channels.channel_id AND
channels.channel_desc IN
     ('Direct Sales', 'Internet') AND
times.calendar_month_desc IN
     ('2000-09', '2000-10')AND countries.country_iso_code IN
('GB','US')
GROUP BY channel_desc, ROLLUP(calendar_month_desc,
     countries.country_iso_code);
```

<u>Ergibt:</u>

```
CHANNEL_DESC         CALENDAR CO SALES$
-------------------- -------- -- --------------
Internet             2000-09  GB        228,241
Internet             2000-09  US        228,241
Internet             2000-09            456,482
Internet             2000-10  GB        239,236
Internet             2000-10  US        239,236
Internet             2000-10            478,473
Internet                                934,955
Direct Sales         2000-09  GB      1,217,808
Direct Sales         2000-09  US      1,217,808
Direct Sales         2000-09          2,435,616
Direct Sales         2000-10  GB      1,225,584
Direct Sales         2000-10  US      1,225,584
Direct Sales         2000-10          2,451,169
Direct Sales                          4,886,784
```

13 Window queries

Die Zeitdimension ist sehr wichtig in Decision Support Systems, da wir z.B. wissen wollen, wie der Verlauf von Verkäufe sich entwickelt über Monate oder Jahre. Deshalb wurde in SQL speziell eine Erweiterung eingeführt, die QUERY WINDOW. Beispiele von zeitliche Abfragen:

(1) Finde den Gesamtverkauf pro Monat;

(2) Finde den Gesamtverkauf pro Monat und Stadt;

(3) Finde die prozentuale Änderung in monatliche Verkäufe für jede Produkte;

(4) Finde die 5 bestverkaufte Produkte;

(5) Finde den mittleren Verkauf über die letzten n Tagen (d.h. für jeden tag müssen wir den Mittelwert der täglichen Verkauf der n vorherigen tagen berechnen);

(6) Finde die top 5 Produkte mit Gesamtverkauf für jeden Monat über das letzte Jahr.

(1) und (2) können normal mit SQL::92 gemacht werden (GROUP BY), (3) und (4) sind schwieriger, (5) und (6) sind mit SQL-92 nicht möglich gewesen. WINDOW ist deshalb eine Erweiterung von SQL-92, und ist ab SQL::99 vorhanden.

Hier ein Beispiel mit Oracle:

```
SELECT L.kanton, T.monat, avg(S.verkauf) OVER w as movavg
FROM Verkauf S, Zeit T, Ort L
WHERE S.timeid=T.timeid AND S.locid = L.locid
WINDOW w AS ( PARTITION BY L.kanton
            ORDER BY T.monat
            RANGE BETWEEN INTERVAL ‚1’ MONTH PRECEDING AND
                        INTERVAL ‚1’ MONTH FOLLOWING)
```

1. Schritt: SELECT / FROM / WHERE kreiert eine Tabelle temp (konzeptuell)

2. Schritt: PARTITION BY: entspricht der GROUP BY, d.h. entscheidet, auf welcher Attribut in der zweiten, nicht-Zeit Dimension die Aufsummierung (also AVG) stattfinden wird, in diesem Fall hier L.kanton. Im Gegensatz zu GROUP BY generiert jede Input-Zeile in der Partition eine Output-Zeile (d.h. hier eine Zeile pro Monat-Attribut) Wir können Partitionen hier betrachten, weil die Abfrage eine Antwort pro Zeile zurückgibt (auch wenn es nur eine Partition geben würde), in GROUP BY könnte T.monat nicht vorkommen.

2. Schritt: ORDER BY: Hier ist es über T.monat geordnet, d.h. die Partition liefert die Resultate pro Monat sortiert.

3. Schritt: Framing: man muss die grenzen der Window angeben: im Beispiel für Juni ist Mai, Juni, Juli.

Die Ordnung muss von Typ numeric, datetime, oder interval sein.

Es gibt eine weitere Form für Framen, die anstatt RANGE BETWEEN... angewendet sein kann: ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING.

Beispiel in Oracle:

```
SELECT t.time_id, TO_CHAR (SUM(amount_sold), '9,999,999,999')
AS SALES, TO_CHAR(AVG(SUM(amount_sold)) OVER
  (ORDER BY t.time_id
   RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND
     INTERVAL '1' DAY FOLLOWING), '9,999,999,999') AS
     CENTERED_3_DAY
FROM sales s, times t
WHERE s.time_id=t.time_id AND t.calendar_week_number IN (51)
     AND calendar_year=1999
GROUP BY t.time_id
ORDER BY t.time_id;

TIME_ID   SALES          CENTERED_3_DAY
--------- -------------- --------------
20-DEC-99        134,337        106,676
21-DEC-99         79,015        102,539
22-DEC-99         94,264         85,342
23-DEC-99         82,746         93,322
24-DEC-99        102,957         82,937
25-DEC-99         63,107         87,062
26-DEC-99         95,123         79,115
```

Man sieht, dass Oracle sehr frei den SQL-Standart interpretiert. Z.B. benutzt er das Keyword OVER anstatt WINDOW. Auch die Position des Partitionings ist anders geregelt. Immerhin ist die dahinterliegende Idee die gleiche.

14 Neue (SQL:2003) Aggregate Funktionen

Die in SQL 2003 neu definierten Funktionen wirken auf Listen von Werte, die mit Windowing- Funktionen erzeugt werden.

RANK(): Gibt die Position einer Zeile innerhalb ihrer Partition; folgt die ORDER BY Klausel in der Window Definition;

DENSE_RANK(): wahrend RANK() eine Liste wie 1,2,2,4,5,.. generiert, wird mit DENSE_RANK keine Zahl überspringen, d.h. die Liste wäre dann 1,2,2,3,4,..

PERCENT_RANK: Relative Position innerhalb einer Partition = (rank-1)/n_rows_partition

Hier ein Beispiel einer Oracle Abfrage mit RANK:

```
SELECT channel_desc, TO_CHAR(SUM(amount_sold), '9,999,999,999')
   SALES$,
   RANK() OVER (ORDER BY SUM(amount_sold)) AS default_rank,
   RANK() OVER (ORDER BY SUM(amount_sold) DESC NULLS LAST) AS
   custom_rank
FROM sales, products, customers, times, channels, countries
WHERE sales.prod_id=products.prod_id AND
      sales.cust_id=customers.cust_id AND
      sales.time_id=times.time_id AND
      sales.channel_id=channels.channel_id AND
      times.calendar_month_desc IN ('2000-09', '2000-10') AND
      country_iso_code='US'
GROUP BY channel_desc;

CHANNEL_DESC          SALES$         DEFAULT_RANK CUSTOM_RANK
-------------------- -------------- ------------ ------------
Direct Sales         2,443,392            3            1
Partners             1,365,963            2            2
Internet             467,478              1            3
```

15 Indizes für OLAP

Indizes und Views können die Geschwindigkeit der CUBE-assoziierten Berechnungen erheblich erhöhen. Entsprechend haben OLAP-Abfragen die Interesse der Forschung und Entwicklung über Indizes und Views erhöht. Die entsprechende Forschung auf dem Thema hat ein neues Leben gegeben.

Neue Indizierungstechniken wurden untersucht, wie zum Beispiel Bitmap Indizes, Join Indizes, Array Darstellungen, Komprimierung, Vorberechnung von Aufsummierungen, usw.

15.1 Bitmap Index

Ein Beispiel von Bitmap Index ist in

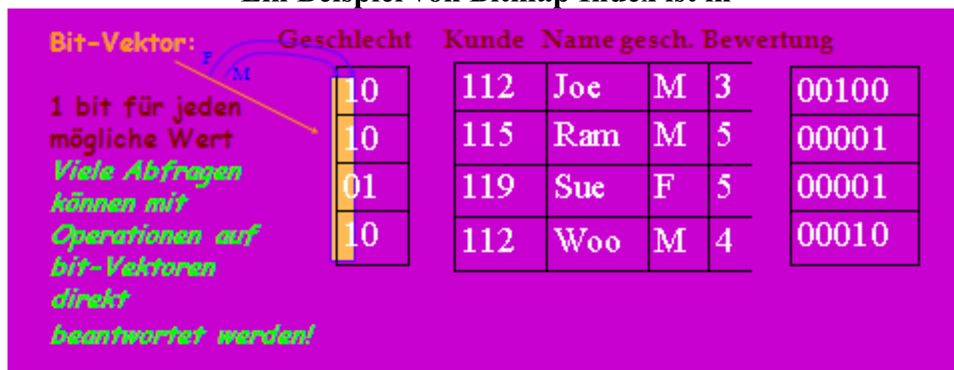

Abbildung 2 angegeben.

Abbildung 2: Konzept des Bitmap Indexes

15.2 Join Index

Joins mit sehr kurzen Antwortszeiten sind schwierig zu realisieren, wenn die Relationen (Tabellen) sehr gross sind. Deshalb möchte man einen Index erstellen, um die spezifische (OLAP) Join Abfragen zu beschleunigen.

Betrachet man z. B. den Schema Verkauf / Ort, kann man ein Index auf ortid erstellen, wobei dieser Index dann Datensätze <ortid, p> enthält, p ist rid in der Fact Table Verkauf.

Eine raffiniert Erweiterung, die von Join Index angegeben ist, ist dass es eine WHERE Klausel akzeptiert wird, was bei Indizes revolutionär ist. So kann der Index vorselektieren. Damit ist ein Join Index etwas zwischen eine View und ein Index.

Wir können der Join index wie folgt beschreiben. Nehmen wir an, wir haben zwei Dimensionen D1 und D2, eine Fact Table F. Wir haben Selektionskonditionen K1, und K2 auf D1, respektiv D2. Wir können damit im Join Index Datensätze aufbauen: <r1, r2, r > wobei r1 erfüllt K1, r2 erfüllt K2 (r1 und r2 sind die ids in die Dimensionen), und das ganze ist ein Join von D1, D2 und F.

Der Nachteil ist, dass der Anzahl Join Indizes sehr schnell wächst wenn wir Indizes für viele Selektionen auf Kolonnen brauchen. Ein Alternative ist deshalb wie folgt:

Sei C1 die Kolonne von D1 auf welcher die Selektion K1 ist angewendet. Sei C2 die Kolonne von D2 auf welcher die Selektion K2 ist angewendet. Wir können F mit D1 joinen und F mit den Feldern von D1 erweitern. Wir bilden dann der Index auf C1: <c1, r>. Wir machen dann das gleiche für C2, D2, K2. Wir bekommen dann eine Zweite relation <c2, r> Dann müssen wir nur nich die rid intersection machen and we're done.

15.3 Bitmap Join Index

Der Bitmap Join Index ist die Erweiterung des Join Index für den Fall wo die Wertebereiche der Indix-Kollonnen beschränkt sind. Z.B. wenn dieser nur 300 Werte enthält kann damit eine Stufe mehr Effizienz gewonnen werden. Hier ist ein bitmap join index in Oracle:

```
create bitmap index
   part_suppliers_state
on
   inventory( parts.part_type, supplier.state)
from
   inventory i,
   parts p,
   supplier s
where
   i.part_id=p.part_id
and
   i.supplier_id=p.supplier_id;
```

16 Views in DWHs

OLAP Abfragen sind üblicherweise Abfragen auf aufsummierte Daten [Aggregate queries]

- Vorberechnungen sind nötig für interaktive Antwortszeiten
- Der CUBE ist tatsächlich eine Sammlung von Integrierte Abfragen. Vorberechnungen sind extrem wichtig. Es wurde viel Arbeit investiert, um zu wissen, was das Beste zu Vorberechnen ist, angenommen wir haben nicht unendlich Platz, um diese Vorberechnungen zu speichern.

Warehouses können als Sammlungen von asynchron replizierten Tabellen und periodisch aktualisierte Views angesehen werden.

Hier ist ein Beispiel von View, das auf unser Star Schema angewendet sein kann:

```
CREATE VIEW LokalVerkauf(kategorie, kanton, verkauf)
     AS SELECT P.kategorie, V.verkauf, O.kanton
          FROM Produkte P, Verkauf V, Orte O
          WHERE P.pid=V.pid AND O.oid=V.oid
```

16.1 View Materialisation (Vorberechnung)

Es gibt es eine Zeitgrenze auf was wir mit OLAP Abfragen erreichen können. Für grosse Datenmengen wird es immer zu langsam sein, und es braucht deshalb eine andere Annäherung an dem Problem. Diese ist die Materialisation der Abfrage, d.h. die Abfrage wird vorgerechnet und physisch in einer neuen Relation gespeichert.

Materialisierte Views sind nicht nur für Data Warehousing interessant. Sie werden in folgenden Bereiche eingesetzt:

- Mobile Computing
- Verbesserung der Abfrageevaluation
- Datenreplizierung für verteilte Systeme

„Lebenslauf“ der Views (spezialisiert auf Oracle)

(1) Der/die DBA erzeugt ein oder mehrere materialisierte Views;

(2) Die Anwendern fragen Tabellen des Star Schemas, nicht die Views ab;

(3) Das Query Rewrite Mechanismus in Oracle schreibt die Abfrage um, um die materialisierte View zu anwenden;

Materialisierte Views sind also transparent zu den Anwendern oder Anwendungen.

Ein Schema für die Query Evaluation ist in

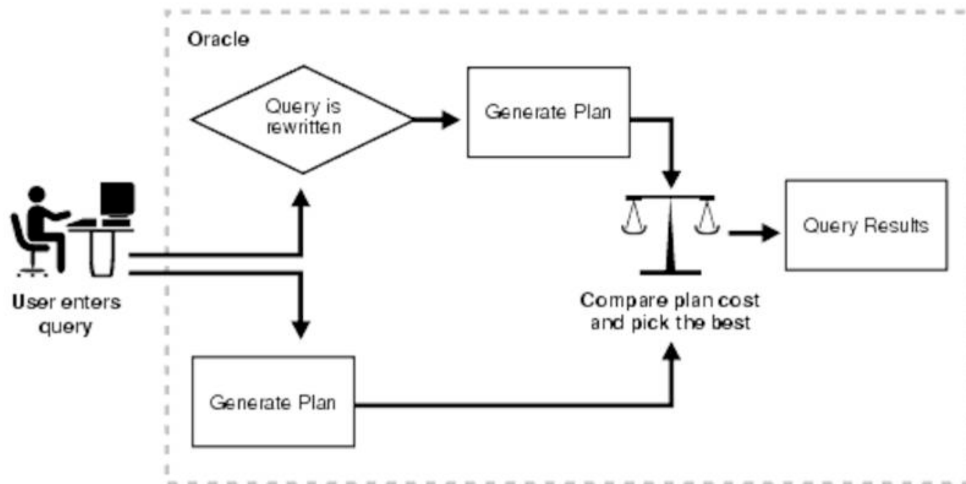

Abbildung 3 angegeben.

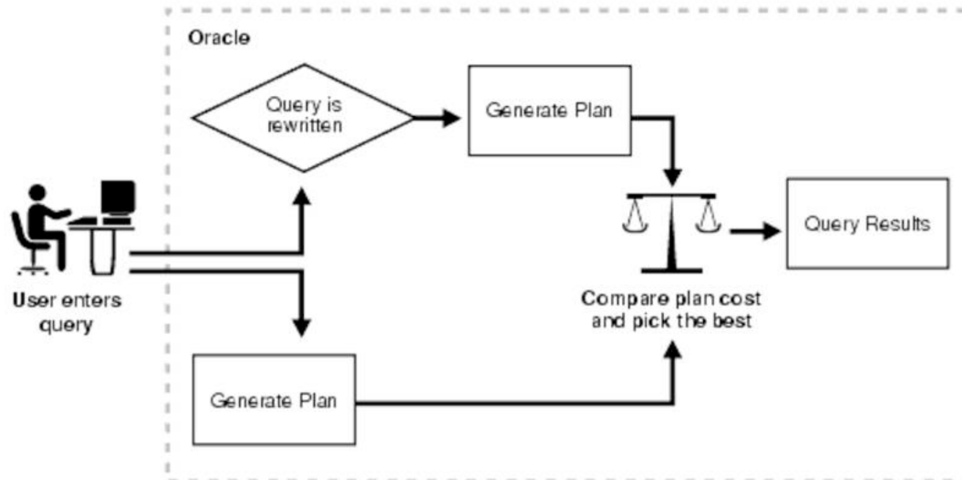

Abbildung 3: Query Rewrite Mechanismus in Oracle

So funktioniert Query Rewrite:
(1) Zuerst wird eine Abfrage erstellt (vom Benutzer/in);
(2) Ein Plan für die Abfrage wird generiert ;
(3) Die Abfrage wird erneut mit der Benutzung der materialisierten View geschrieben;
(4) Ein Plan für diese neue Abfrage wird erstellt;
(5) Die Kosten für beide Pläne werden verglichen;
(6) Die Abfrage mit dem besten Resultat wird gewählt.

16.2 View Materialisation und Indizes

Nehmen wir an, wir rechnen Regionalverkauf vor (also erstellen wir eine materialisierte View), und kreieren einen B-Baum Index auf [kategorie,kanton,verkauf]. Mit dem kann die

Abfrage, die bei der Query Modification angegeben ist, durch ein index-only Scan evaluiert werden.

Man betrachte folgende Abfrage:

```
SELECT R.kanton, SUM(R.verkauf)
FROM Regionalverkauf R
WHERE R.kategoie="Laptop"
GROUP BY R.kanton
```

Diese Abfrage kann sehr effizient mit der vorgerechneten View und dem oben erwähnten Index evaluiert werden: man sucht mit dem Baum die Blätter, die der Kategorie „Laptop" entsprechen, dann scannen die Blätter bis man auf ein Element kommt, der nicht die Kategorie „Laptop" gehört.

```
SELECT R.kategorie, SUM(R.verkauf)
FROM Regionalverkauf R
WHERE R.kanton="Aargau"
GROUP BY R.kategorie
```

Auf dieser Abfrage ist der Index weniger nützlich, da alle Blätter des Baums trotzdem durchgesucht sein müssen.

Das zeigt ein Problem mit Materialisation und Indizes: sie sind gut für gewisse Abfragen, aber für andere nicht! Das heisst also, der oder die DBA muss immer sich fragen, welche Abfragen einen Sinn machen, und welche angefragt werden, um zu entscheiden, welche Indizes/Views zu kreieren.

Hier ein weiteres Beispiel, das die Nützlichkeit von View Materilaisierung illustrieren sollte.

SELECT P.kategorie, SUM(V.verkauf)

FROM Produkte P, Verkauf V

WHERE P.pid = V.pid

GROUP BY P.kategorie

Das änliche kann man mit Orte machen:

SELECT O.Kanton, SUM(V.verkauf)

FROM Orte O, Verkauf V

WHERE P.oid = O.oid

GROUP BY O. Kanton.

Was wir hier brauchen ist die Verkauf Tabelle (sehr gross!) mit der Produkte Tabelle zu joinen, dann das Resultat aggregieren. Wie kann da die Materialisieren helfen?

....

16.3 Probleme mit View Materialisation