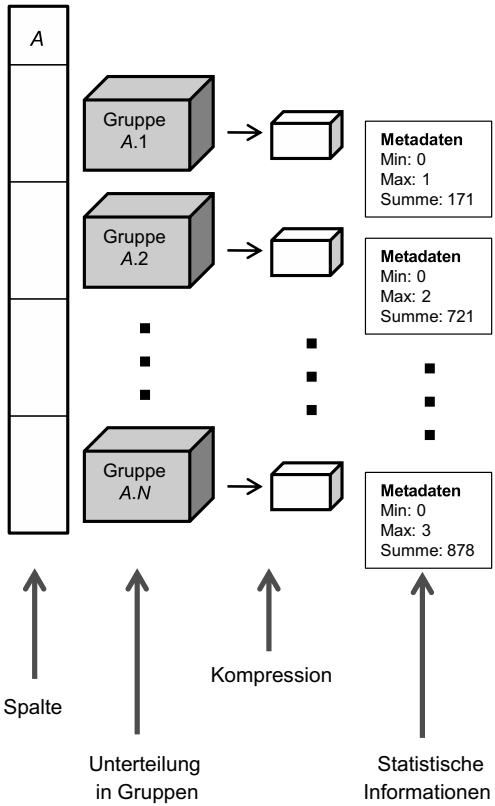

Bild 14.15 Statistische Informationen und Metadaten

14.4 Erweiterung relationaler Datenbanken

Aufgrund der besonderen Eigenschaften von DWH (z. B. großes Datenvolumen, besondere Ladeanforderungen, komplexe Anfragetypen) kann es bei Verwendung konventioneller relationaler Datenbanken zu Performance-Engpässen kommen. Begegnet worden ist diesem Problem von den führenden DBMS-Anbietern (u. a. Oracle, IBM und Microsoft) durch Erweiterungen zur Beschleunigung von Anfragen bzw. Einfügeoperationen. Die wichtigsten Erweiterungen werden in diesem Abschnitt vorgestellt.

14.4.1 Materialisierte Sichten

Nur in wenigen Fällen beziehen sich Anfragen der Datenanalyse auf den im DWH gespeicherten Detaillierungsgrad, vielmehr werden die Daten häufig in verdichteter Form benötigt, z. B. bei einer Auswertung nach Verkaufsregionen und Produktgruppen pro Quartal statt nach Filialen und Produkten pro Tag.

Diese Art von Anfragen kann durch **materialisierte Sichten** gut unterstützt werden, indem das vorberechnete Resultat redundant gespeichert wird. Die SQL-Anweisung `CREATE VIEW` (→ 4.7.6), die ein rein virtuelles Konzept realisiert, wurde zur `CREATE-MATERIALIZED-VIEW`-Anweisung erweitert. Hierbei können zusätzliche Optionen, wie z. B. die Aktualisierungsstrategie, festgelegt werden.

Bild 14.16 zeigt Beispieldaten, aus denen eine materialisierte Sicht angelegt wird.

Ort

Filiale	Stadt	Region	Land
Hamburg	Hamburg	Nord	D
Leipzig	Leipzig	Ost	D
Stuttgart	Stuttgart	Süd	D
Bremen-Nord	Bremen	Nord	D
Bremen-Süd	Bremen	Nord	D
München	München	Süd	D

Zeit

Tag	Woche	Monat	Quartal	Jahr
5.1.2006	2006 - 1	2006 - 1	2006 - Q1	2006
12.1.2006	2006 - 2	2006 - 1	2006 - Q1	2006
13.2.2006	2006 - 7	2006 - 2	2006 - Q1	2006
23.2.2006	2006 - 8	2006 - 2	2006 - Q1	2006
4.3.2006	2006 - 9	2006 - 3	2006 - Q1	2006
7.4.2006	2006 - 14	2006 - 4	2006 - Q2	2006
25.4.2006	2006 - 17	2006 - 4	2006 - Q2	2006

Verkaufszahl

Filiale	Produkt	Tag	Anzahl
Hamburg	Pizza Funghi	5.1.2006	78
Hamburg	Pizza Funghi	12.1.2006	67
Leipzig	Pizza Hawaii	12.1.2006	42
München	Pizza Calzione	13.2.2006	53
Stuttgart	Pizza Napoli	23.2.2006	23
Bremen-Nord	Pizza Funghi	4.3.2006	69
Bremen-Süd	Pizza Vegetale	7.4.2006	45
Stuttgart	Pizza Hawaii	25.4.2006	92

Produkt

Produkt	Marke	Hersteller	Produktgruppe
Pizza Funghi	Gourmet-Pizza	Frost GmbH	Tiefkühlkost
Pizza Hawaii	Gourmet-Pizza	Frost GmbH	Tiefkühlkost
Pizza Napoli	Pizza	TK-Pizza AG	Tiefkühlkost
Pizza Vegetale	Good&Cheap	Frost GmbH	Tiefkühlkost
Pizza Calzione	Pizza	TK-Pizza AG	Tiefkühlkost

Bild 14.16 Beispieldaten für das Sternschema aus Bild 14.11

◊ *Programm:* Aus den Daten in Bild 14.16 wird mit folgender Anweisung eine materialisierte Sicht angelegt (→ Bild 14.17):

```
CREATE MATERIALIZED VIEW Region_Marke_Jahr AS
SELECT O.Region, P.Marke, Z.Jahr, SUM(V.Anzahl) AS Anzahl
FROM   (((Verkaufszahl V JOIN Ort O ON(V.Filiale=O.Filiale))
            JOIN Zeit Z ON(V.Tag=Z.Tag))
            JOIN Produkt P ON(V.Produkt=P.Produkt))
GROUP BY O.Region, P.Marke, Z.Jahr;
```

Mat. Sicht: Region_Marke_Jahr			
Region	Marke	Jahr	Anzahl
Nord	Gourmet-Pizza	2006	214
Nord	Good&Cheap	2006	45
Ost	Gourmet-Pizza	2006	42
Süd	Pizza	2006	76
Süd	Gourmet-Pizza	2006	92

Bild 14.17 Anlegen Materialisierter Sichten

Wird nun eine Anfrage an das DWH gestellt, so erkennt der Optimierer die Existenz dieser materialisierten Sicht und kann sie durch Umschreiben der Anfrage (*query rewrite*) nutzen. Dabei kann die materialisierte Sicht auch für Anfragen dienen, die höher verdichtet sind als die materialisierte Sicht selbst.

◊ *Programm:*

```
SELECT O.Jahr, Z.Jahr, P.Hersteller
FROM   (((Verkaufszahl V JOIN Ort O ON(V.Filiale=O.Filiale))
            JOIN Zeit Z ON(V.Tag=Z.Tag))
            JOIN Produkt P ON(V.Produkt=P.Produkt))
WHERE  O.Region='Nord' AND Z.Jahr='2006'
GROUP BY O.Jahr, Z.Jahr, P.Hersteller;
```

wird vom DBMS umgeschrieben zu

```
SELECT O.Jahr, Z.Jahr, P.Hersteller
FROM   (((Region_Marke_Jahr RMJ
            JOIN Ort O ON(RMJ.Region=O.Region))
            JOIN Zeit Z ON(RMJ.Jahr=Z.Jahr))
            JOIN Produkt P ON(RMJ.Marke=P.Marke))
WHERE  O.Region='Nord' AND Z.Jahr='2006'
GROUP BY O.Jahr, Z.Jahr, P.Hersteller;
```


14.4.2 Partitionierung

Partitionierung hat ihren Ursprung im Bereich verteilter und paralleler Datenbanksysteme, wobei die Aufteilung einer Tabelle auf einzelne Rechnerknoten mit dem Ziel der Lastverteilung im Vordergrund steht. Dabei

werden die zwei Phasen der *Fragmentierung* (Bestimmung der Verteilungseinheiten) und der *Allokation* (Zuordnung der Fragmente zu physischen Einheiten wie Plattenspeichern oder Rechnerknoten) unterschieden (→ 13.2). Aber auch in nicht verteilten Datenbanken können Performance-Steigerungen durch Partitionierung erreicht werden, indem eine Tabelle mit umfangreicher Extension auf mehrere kleinere, als **Partitionen** bezeichnete Tabellen aufgeteilt wird. Aufgrund ihrer Größe bietet sich in einem DWH insbesondere die Faktentabelle zur Partitionierung an. Im Wesentlichen wird zwischen den in Bild 14.18 dargestellten Varianten *horizontaler* und *vertikaler* Partitionierung unterschieden.

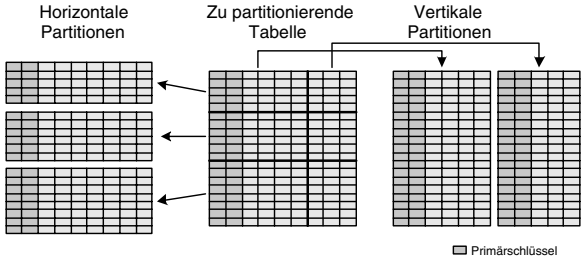

Bild 14.18 Partitionierung

Horizontale Partitionierung kann *zufällig*, z. B. nach dem *Round-Robin-Verfahren*, oder *wertebasiert* erfolgen /14.7/, /14.17/. Während wertebasierte Partitionierung bei lesenden DB-Operationen nur auf bestimmte Partitionen zuzugreifen braucht, kann zufällige Partitionierung zur Erhöhung des Parallelitätsgrades von Einfügeoperationen genutzt werden. Bei der wertebasierten Variante unterscheidet man wiederum zwischen *Bereichs-* und *Hash-Partitionierung*. Während bei der Hash-Partitionierung eine Funktion über die Fragmentierung der Tupel entscheidet, geschieht die Bereichspartitionierung aufgrund semantischer Kriterien. Im Kontext von DWH bieten sich häufig Ort und Zeit als Kriterien zur Fragmentierung an. Werden beispielsweise in einem DWH die Zahlen der letzten vier Jahre gespeichert, könnte die gesamte Faktentabelle durch Bereichsfragmentierung in vier Partitionen aufgesplittet werden.

Vertikale Partitionierung bietet sich vor allem für besonders „breite“ Tabellen, d. h. solche mit vielen Attributen, an. Im DWH kann dies für einige Dimensionstabellen zutreffen. Weil für das Wiederzusammensetzen der Partitionen jedoch eine relativ teure 1:1-Verbundanfrage nötig ist, besitzt die vertikale Partitionierung im DWH-Umfeld nur eine untergeordnete Bedeutung.

Zur Unterstützung der Partitionierung von Tabellen ist die CREATE-TABLE-Anweisung um eine Klausel erweitert worden, in der die Partitionierungskriterien festgelegt werden.

❑ *Beispiel:* Bereichspartitionierung

```
CREATE TABLE Verkaufszahl(...) AS
PARTITION BY RANGE (Tag)(
PARTITION VerkaufVor2005 VALUES LESS THAN('2005-01-01'),
PARTITION Verkauf2005 VALUES LESS THAN('2006-01-01'),
PARTITION Verkauf2006 VALUES LESS THAN('2007-01-01'),
PARTITION VerkaufNach2006 VALUES LESS THAN(MAXVALUE));
```

14.4.3 Bitmap-Index

Herkömmliche Indexstrukturen basieren auf B- bzw. B*-Bäumen (→ 8.3.2) und sind für das effiziente Suchen und Finden einzelner Datensätze ausgelegt. Die meisten analytisch geprägten Anfragen an ein DWH umfassen jedoch immer einen Bereich von Daten, und herkömmliche Indexstrukturen können nicht unterstützend wirken. *Bitmap-Indexe* jedoch unterstützen diese Art von Anfragen.

> Bei einem **Bitmap-Index** wird für jede Ausprägung des indexierten Attributs ein Bitvektor von der Länge der Anzahl der Datensätze angelegt und an jeder Stelle der Wert 0 oder 1 eingetragen, je nachdem, ob das Attribut in diesem Datensatz einen Wert hat oder nicht.

Bei Anfragen kann durch Anwendung von booleschen Operatoren auf die Bitvektoren die Resultatsmenge effizient bestimmt werden, was unter anderem bei sog. Star Queries (→ 14.4.6) zur Anwendung kommt.

Eine wichtige Variante sind *bereichscodierte Bitmap-Indexe*, in denen in einem Bit nicht ein einzelner Wert, sondern ein Bereich codiert wird.

❑ *Beispiel:* Bild 14.19 zeigt ein Beispiel der Codierung des Attributs Monat: Der i-te Vektor repräsentiert den i-ten Monat und besitzt an einer Position eine 1, wenn der Wert des entsprechenden Tupels im Bereich bis einschließlich dieses Wertes liegt.

Für jede Bereichsanfrage müssen höchstens zwei Bitvektoren gelesen werden, z. B.:

- Bereich von Februar bis einschließlich Mai („Januar“ $< x \leq$ „Mai“): NOT B_1 AND B_5
- Bereich bis einschließlich Juni ($x \leq$ „Juni“): B_6
- Bereich ab August ($x >$ „Juli“): NOT B_7

- Punktanfragen (d. h. im Beispiel das Lesen eines einzelnen Monats) erfordern jedoch das Lesen von zwei Bitvektoren, z. B. für Mai: B_5 AND NOT B_4

Zeit				
Tag	Woche	Monat	Quartal	Jahr
5.12.2006	2006 - 49	2006 - 12	2006 - Q4	2006
12.1.2006	2006 - 02	2006 - 1	2006 - Q1	2006
23.3.2006	2006 - 12	2006 - 3	2006 - Q1	2006
4.5.2006	2006 - 18	2006 - 5	2006 - Q2	2006
7.1.2006	2006 - 01	2006 - 1	2006 - Q1	2006
13.7.2006	2006 - 28	2006 - 7	2006 - Q3	2006
7.8.2006	2006 - 32	2006 - 8	2006 - Q3	2006
13.5.2006	2006 - 19	2006 - 5	2006 - Q2	2006

Bereichscodierter Bitmap-Index über der Spalte Monat

B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}	B_{11}	B_{12}
0	0	0	0	0	0	0	0	0	0	0	1
1	1	1	1	1	1	1	1	1	1	1	1
0	0	1	1	1	1	1	1	1	1	1	1
0	0	0	0	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	1	1	1	1	1	1
0	0	0	0	0	0	0	1	1	1	1	1
0	0	0	0	1	1	1	1	1	1	1	1

Bild 14.19 Bereichscodierter Bitmap-Index

14.4.4 SQL-Erweiterungen zum Einfügen

Im SQL-Standard kann nur eine Aktualisierungs- oder eine Einfügeoperation ausgeführt werden. Die `MERGE-INTO`-Anweisung ermöglicht eine kombinierte Anwendung dieser beiden Operationen (→ 4.6.4). Ist ein Datensatz bereits in der Tabelle, so soll dieser aktualisiert werden. Ist der Datensatz jedoch noch nicht in der Tabelle, so wird er eingefügt.

Produkt	
ID	Name
4711	Pizza Funghi
4712	Pizza Quattro Stagione
4713	Pizza Vegetale

Produkt_Neu	
ID	Name
4711	Pilz-Pizza
4712	Pizza Quattro Stagione
4713	Pizza Vegetale
4714	Pizza Hawaii

```
MERGE INTO Produkt P1
USING (SELECT ID, Name
       FROM Produkt_Neu) P2
ON    (P1.ID = P2.ID)
WHEN MATCHED THEN
  UPDATE SET P1.Name = P2.Name
WHEN NOT MATCHED THEN
  INSERT (P1.ID, P1.Name) VALUES (P2.ID, P2.Name);
```

Produkt	
ID	Name
4711	Pilz-Pizza
4712	Pizza Quattro Stagione
4713	Pizza Vegetale
4714	Pizza Hawaii

Bild 14.20 Beispiel `MERGE-INTO`-Anweisung

❑ *Beispiel:* In Bild 14.20 wird eine neue Liste mit Produkten zur Verfügung gestellt, die mit der Tabelle Produkt gemischt werden soll. Datensatz 4711 wird aktualisiert, die unveränderten Datensätze 4712 und 4713 bleiben erhalten (bzw. werden mit den gleichen Werten überschrieben) und der neue Datensatz 4714 wird hinzugefügt.

14.4.5 Komplexes Gruppieren

Für die beschleunigte bzw. leichtere Berechnung von (Zwischen-)Summen ist die GROUP-BY-Klausel erweitert worden. Zur Demonstration dieser erweiterten Gruppierungsmöglichkeiten soll der Datenbestand in Bild 14.21a dienen.

Ort_Produkt_Monat_Verkauf

Ort	Produkt	Monat	Anzahl
Stuttgart	Pizza Funghi	2006 - 1	155
Stuttgart	Pizza Vegetale	2006 - 1	133
Stuttgart	Pizza Hawaii	2006 - 1	89
Stuttgart	Pizza Funghi	2006 - 2	141
Stuttgart	Pizza Vegetale	2006 - 2	112
Stuttgart	Pizza Hawaii	2006 - 2	95
Frankfurt	Pizza Funghi	2006 - 1	77
Frankfurt	Pizza Vegetale	2006 - 1	93
Frankfurt	Pizza Hawaii	2006 - 1	102
Frankfurt	Pizza Funghi	2006 - 2	144
Frankfurt	Pizza Vegetale	2006 - 2	178
Frankfurt	Pizza Hawaii	2006 - 2	177

a) Beispieldaten für Gruppierungsanfragen

```
SELECT    Monat, Produkt,
          SUM(Anzahl) AS Anzahl
FROM      Ort_Produkt_Monat_Verkauf
GROUP BY Monat, Produkt;
```

Monat	Produkt	Anzahl
2006 - 1	Pizza Funghi	232
2006 - 1	Pizza Vegetale	226
2006 - 1	Pizza Hawaii	191
2006 - 2	Pizza Funghi	285
2006 - 2	Pizza Vegetale	290
2006 - 2	Pizza Hawaii	272

b) Beispiel GROUP-BY-Anfrage

```
SELECT
  DECODE(GROUPING(Monat),1,
  'Alle Monate',Monat)  AS Monat,
  DECODE(GROUPING(Produkt),1,'Alle
  Produkte',Produkt)  AS Produkt,
  SUM(Anzahl) AS Anzahl
FROM    Ort_Produkt_Monat_Verkauf
GROUP BY ROLLUP(Monat, Produkt);
```

Monat	Produkt	Anzahl
2006 - 1	Pizza Funghi	232
2006 - 1	Pizza Vegetale	226
2006 - 1	Pizza Hawaii	191
2006 - 1	Alle Produkte	649
2006 - 2	Pizza Funghi	285
2006 - 2	Pizza Vegetale	290
2006 - 2	Pizza Hawaii	272
2006 - 2	Alle Produkte	847
Alle Monate	Alle Produkte	1496

c) Beispiel GROUP-BY-Anfrage mit ROLLUP- und GROUPING-Funktion

```
SELECT
  DECODE(GROUPING(Monat),1,
  'Alle Monate',Monat)  AS Monat,
  DECODE(GROUPING(Produkt),1,'Alle
  Produkte',Produkt)  AS Produkt,
  SUM(Anzahl) AS Anzahl
FROM    Ort_Produkt_Monat_Verkauf
GROUP BY CUBE(Monat, Produkt);
```

Monat	Produkt	Anzahl
2006 - 1	Pizza Funghi	232
2006 - 1	Pizza Vegetale	226
2006 - 1	Pizza Hawaii	191
2006 - 1	Alle Produkte	649
2006 - 2	Pizza Funghi	285
2006 - 2	Pizza Vegetale	290
2006 - 2	Pizza Hawaii	272
2006 - 2	Alle Produkte	847
Alle Monate	Pizza Funghi	517
Alle Monate	Pizza Vegetale	516
Alle Monate	Pizza Hawaii	463
Alle Monate	Alle Produkte	1496

d) Beispiel GROUP-BY-Anfrage mit CUBE-Funktion

Bild 14.21 Komplexe Gruppierungen

Anwendung und Resultat der SQL-Standard-Gruppierung am Beispiel des Gruppierens nach Monaten und Produkten zeigt Bild 14.21b. Neben dieser Gruppierung berechnet die **ROLLUP-Funktion** auch Zwischen- und Gesamtsummen. Bild 14.21c zeigt die um `ROLLUP` erweiterte `GROUP-BY`-Klausel und das Resultat, das auch Zwischensummen von jedem Monat und die Endsumme umfasst. Die Anwendung der Funktion `GROUPING` auf ein gruppiertes Attribut liefert den Wert 1, wenn der Wert eine Zwischen- oder Gesamtsumme ist. Dieses kann in Kombination mit der `DECODE`-Funktion dazu genutzt werden, die leeren Attribute im Resultat mit Text zu füllen, wie das Beispiel in Bild 14.21c zeigt.

- *Hinweis:* Die `DECODE`-Funktion realisiert eine bedingte Anweisung in SQL. Die Syntax ist `DECODE(value, if1, then1, if2, then2, if3, then3, ..., else)`.

- *Hinweis:* Die Verwendung von `SELECT Monat, Produkt, SUM(Anzahl) AS Anzahl` als Projektion (→ Bild 14.21c), d. h. Verzicht auf `GROUPING` und `DECODE`, hätte zur Folge, dass die Attributwerte statt „Alle ...“ den Wert `NULL` hätten.

Zur Berechnung von Zwischensummen für alle Kombinationen von gruppierten Werten steht die **CUBE-Funktion** zur Verfügung, deren Anwendung in Bild 14.21d demonstriert wird.

- *Hinweis:* Ohne diese Erweiterungen müssten die Zwischen- und Gesamtsummen auf Seite des Clients berechnet werden oder es wären mehrere SQL-Anfragen notwendig.

14.4.6 Star Query

Anfragen an ein DWH betreffen typischerweise die Faktentabelle und eine Menge von Dimensionstabellen. In Anlehnung an die Schemabezeichnung wird dieser Anfragetyp als **Star Query** bezeichnet. Werden diese Anfragen ohne weitere Maßnahmen formuliert, so erfolgt die Berechnung durch einen Standard-Verbundalgorithmus (z. B. Nested-Loops- oder Hash-Join, → 7.4.1) und kann bzgl. der Anzahl durchzuführender Leseoperationen relativ teuer sein, da den speziellen Gegebenheiten des zugrunde liegenden Schematyps nicht Rechnung getragen wird.

Als Optimierung für eine Star Query ist folgende Vorgehensweise möglich: Die Fremdschlüssel in der Faktentabelle werden mit einem Bitmap-Join-Index versehen und dem DBMS wird das Vorliegen einer Star Query angezeigt, was durch einen Hinweis in der SQL-Anweisung oder in Konfigurationseinstellungen erfolgt. Ist dies geschehen, wird keiner der Standard-Verbundalgorithmen verwendet, sondern es erfolgt eine Star Query.

❑ *Beispiel:* Für die Beispieldaten aus Bild 14.16 soll die Anzahl abgesetzter Produkte der Marken Gourmet-Pizza und Good&Cheap in den ersten beiden Quartalen in der Region Nord ermittelt werden:

```
SELECT O.Region, Z.Quartal, P.Marke, SUM(V.Anzahl) AS Anzahl
FROM   (((Verkaufszahl V JOIN Ort O ON(V.Filiale=O.Filiale))
            JOIN Zeit Z ON(V.Tag=Z.Tag))
            JOIN Produkt P ON(V.Produkt=P.Produkt))
WHERE O.Region='Nord'
       AND Z.Quartal IN ('2006-Q1','2006-Q2')
       AND P.Marke IN ('Gourmet-Pizza','Good&Cheap')
GROUP BY O.Region, Z.Quartal, P.Marke;
```

Der optimierte Ablauf der Star Query erfolgt in nachstehenden Schritten (→ Bild 14.22).

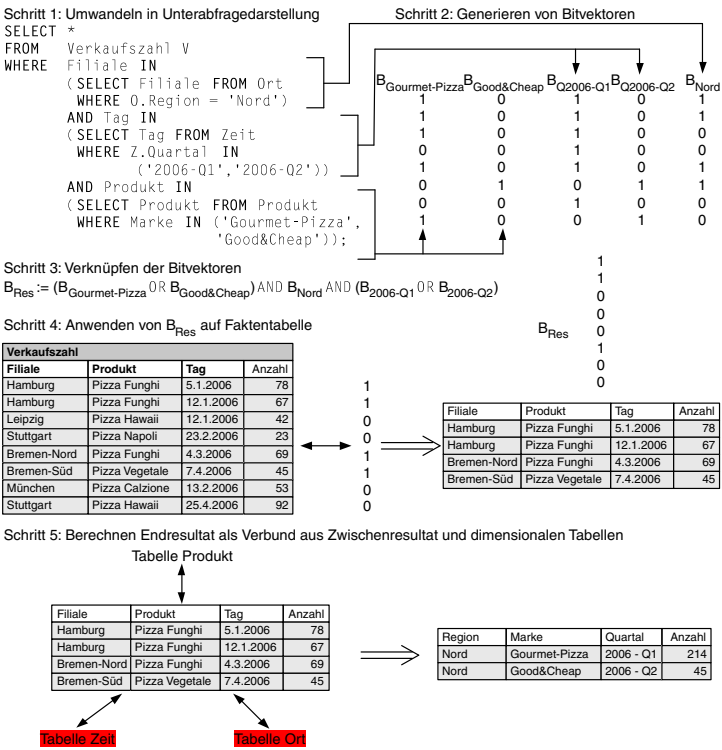

Bild 14.22 Optimierter Ablauf einer Star Query

14.4.7 Bulk Loader

Das Nachladen von Daten in das DWH als letzter Teilschritt des ETL-Prozesses kann eine zeitkritische Aufgabe sein. Aus diesem Grunde wurde mit sog. **Bulk Loadern** (Massenladern) ein Typ Ladewerkzeug entwickelt, der eine große Datenmenge effizient in Datenbanken einfügen kann. Hierzu nehmen Bulk Loader syntaktisch korrekte Daten an und vernachlässigen die Mehrbenutzerkoordination und die Prüfung von Konsistenzbedingungen. Außerdem schreiben Bulk Loader direkt in Datendateien und nicht erst in den DB-Puffer.

► *Hinweis:* Im DWH-Kontext kann das korrekte Datenformat durch entsprechende Konfiguration der am ETL-Prozess beteiligten Werkzeuge erreicht werden. Die Nichtbeachtung des Mehrbenutzerbetriebs ist gerechtfertigt, weil der Ladeprozess als Einziger schreibend auf das DWH zugreift.