Kapitel 10: Performance−Tuning

10.1 Physischer Datenbankentwurf

10.1.1 Indexauswahl

Der wichtigste Faktor beim physischen Datenbankentwurf ist das Anlegen von Indizes. Dabei ist der Nutzen von Indizes zur Beschleunigung von Queries gegenüber den Kosten für Änderungsoperationen und den zusätzlichen Speicherplatzkosten abzuwägen.

Gegeben sei
- ein konzeptionelles Datenbankschema,
- ein Mengengerüst der Datenbank (Anzahl der Tupel in den verschiedenen Relationen, Attributwerteverteilungen, usw.) und
- eine Lastbeschreibung mit Häufigkeiten verschiedener Queries und Änderungsoperationen.

Das Problem der optimalen Auswahl von unclustered, mittelbaren (B*−Baum−) Indizes auf einzelnen Attributen der Relationen kann in simplifizierter Weise wie folgt formalisiert werden.

gegeben:
- m Relationen mit insgesamt n Attributen
- vorhandene Plattenplatzkapazität: P
- Häufigkeit von Exact−Match−Queries auf Attribut i: S_i
- Häufigkeit von Updates auf Attribut i: U_i
- Platzbedarf für einen Index auf Attribut i: P_i
- Kosten einer Indexsuche auf Attribut i: C−$Search_i$
- Kosten eines Scans auf Attribut i (bzw. der entsprechenden Relation): C−$Scan_i$
- Kosten eines Updates auf Attribut i: C−$Update_i$

Ziel:
Minimiere die Gesamtkosten

$$C = \sum_{i=1}^{n} x_i * S_i * C\text{–}Search_i + x_i * U_i * C\text{–}Update_i + (1 - x_i) * S_i * C\text{–}Scan_i$$

mit 0−1−Entscheidungsvariablen $x_i = \begin{cases} 1 \textit{ falls ein Index auf Attribut i angelegt wird} \\ 0 \textit{ sonst} \end{cases}$

unter der Nebenbedingung $C = \sum_{i=1}^{n} x_i * P_i \leq P$

Die Kostenfaktoren C−$Search_i$, C−$Scan_i$, C−$Update_i$ sind analog zur Kostenschätzung bei der Query−Optimierung zu bestimmen.

Ein derartiges Optimierungsmodell kann auch um Selektionen mit komplexeren Booleschen Suchbedingungen, Joins usw. erweitert werden. Die Kosten einer bestimmten Indexkonfiguration hängen dann aber auch wesentlich von den Entscheidungen des Query−Optimierers ab. Daher sollte das Kostenmodell mit dem des Query−Optimierers übereinstimmen.

Das Problem der optimalen Indexauswahl ist NP−hart. Die Wahl einer guten Indexkonfiguration wird noch durch weitere Optionen zusätzlich erschwert:
- Indizes auf Attributkombinationen
- die Wahl von höchstens einem Cluster−Index pro Relation
- die Wahl der Art des Index (Hash−Index oder B*−Baum)
- die Möglichkeit, Join−Indizes anzulegen
- mehrdimensionale Indizes

Dies legt den Einsatz von Heuristiken nahe. Eine einfache Familie von Greedy−Heuristiken sind die ADD/DROP−Heuristiken.

ADD−Heuristik für die Auswahl von Indizes auf einer Relation R(A1, ..., An):

1) Bestimme für jede einzelne Query die Menge der potentiell nützlichen Indizes
   (alle Indizes, die der Query−Optimierer überhaupt verwenden würde).
2) Vereinige die nützlichen Indizes aller Queries.
3) Bestimme die Gesamtkosten für eine Konfiguration ohne Indizes.
4) repeat
   Wähle einen Index aus (z.B. den mit der höchsten Einzelselektivität
   oder denjenigen, der die höchste Kostenverbesserung erzielt)
   und füge ihn zur aktuellen Konfiguration hinzu.
   Bestimme die Gesamtkosten der neuen Konfiguration.
   until keine Kostenverbesserung or alle überhaupt nützlichen Indizes ausgewählt

DROP−Heuristik für die Auswahl von Indizes auf einer Relation R(A1, ..., An):

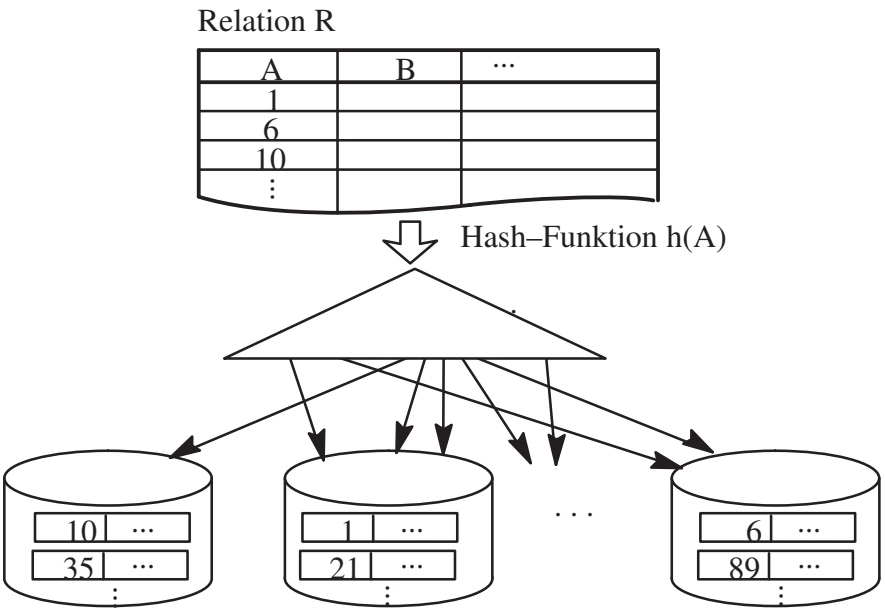
1) Bestimme für jede einzelne Query die Menge der potentiell nützlichen Indizes
   (alle Indizes, die der Query−Optimierer überhaupt verwenden würde).
2) Vereinige die potentiell nützlichen Indizes aller Queries.
3) Bestimme die Gesamtkosten
   für die Konfiguration mit allen potentiell nützlichen Indizes.
4) repeat
   Wähle einen Index aus (z.B. den mit der niedrigsten Einzelselektivität
   oder denjenigen, der die höchste Kostenverbesserung erzielt)
   und entferne ihn von der aktuellen Konfiguration.
   Bestimme die Gesamtkosten der neuen Konfiguration.
   until keine Kostenverbesserung or keine Indizes in der Konfiguration

Werkzeuge, die auf solchen Heuristiken basieren, rufen z.T. intern den Query−Optimierer auf, um potentiell nützliche Indizes zu bestimmen, sowie zur Ermittlung der Gesamtkosten einer Konfiguration. Dadurch ist sichergestellt, daß die Kostenmodelle des Tuning−Werkzeugs und des Query−Optimierers übereinstimmen.

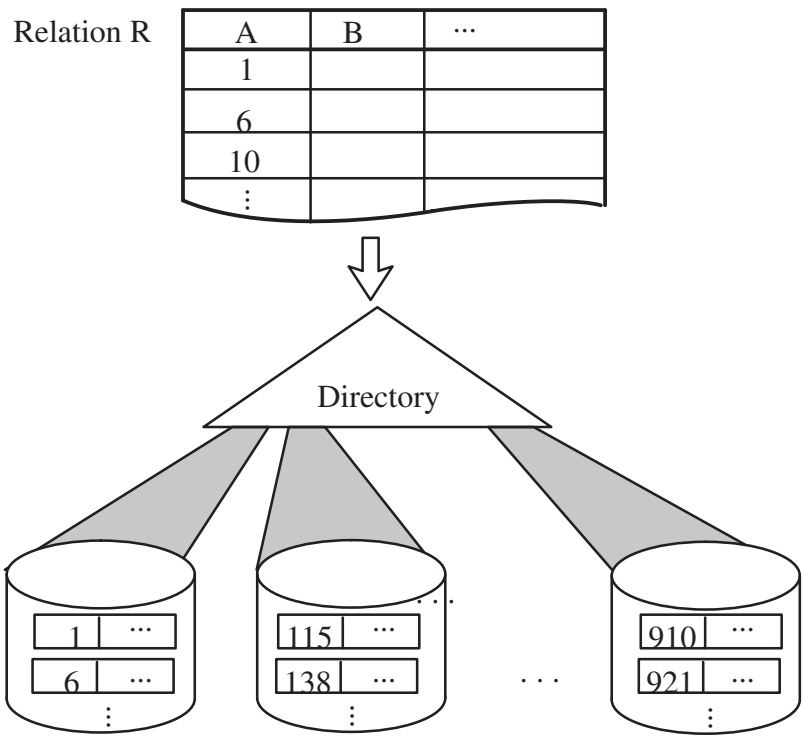
10.1.2 Datenpartitionierung in parallelen DBS

In parallelen DBS können Relationen in der Regel auf drei Arten partitioniert und über mehrere Platten bzw. Verarbeitungsknoten eines Parallelrechners verteilt werden:

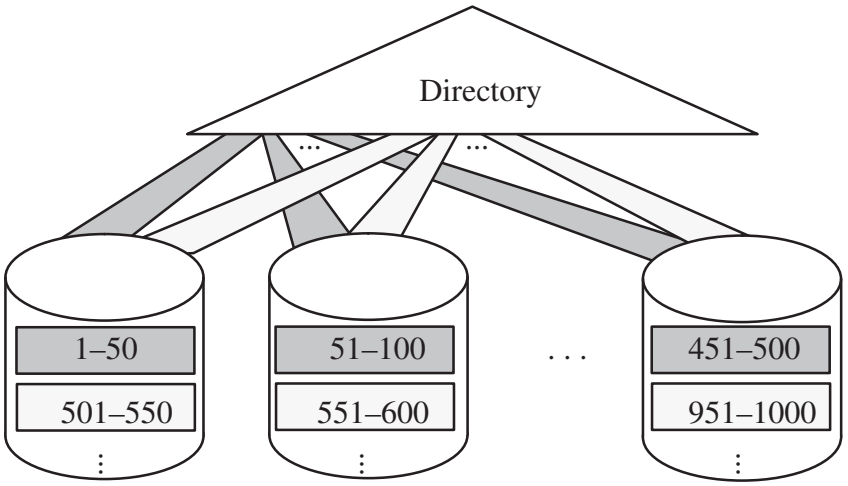
Hash−Partitionierung:
Eine Hash−Funktion wird auf den Wert eines Attributes angewendet;
alle Tupel mit demselben Hash−Funktionswert bilden eine Partition.



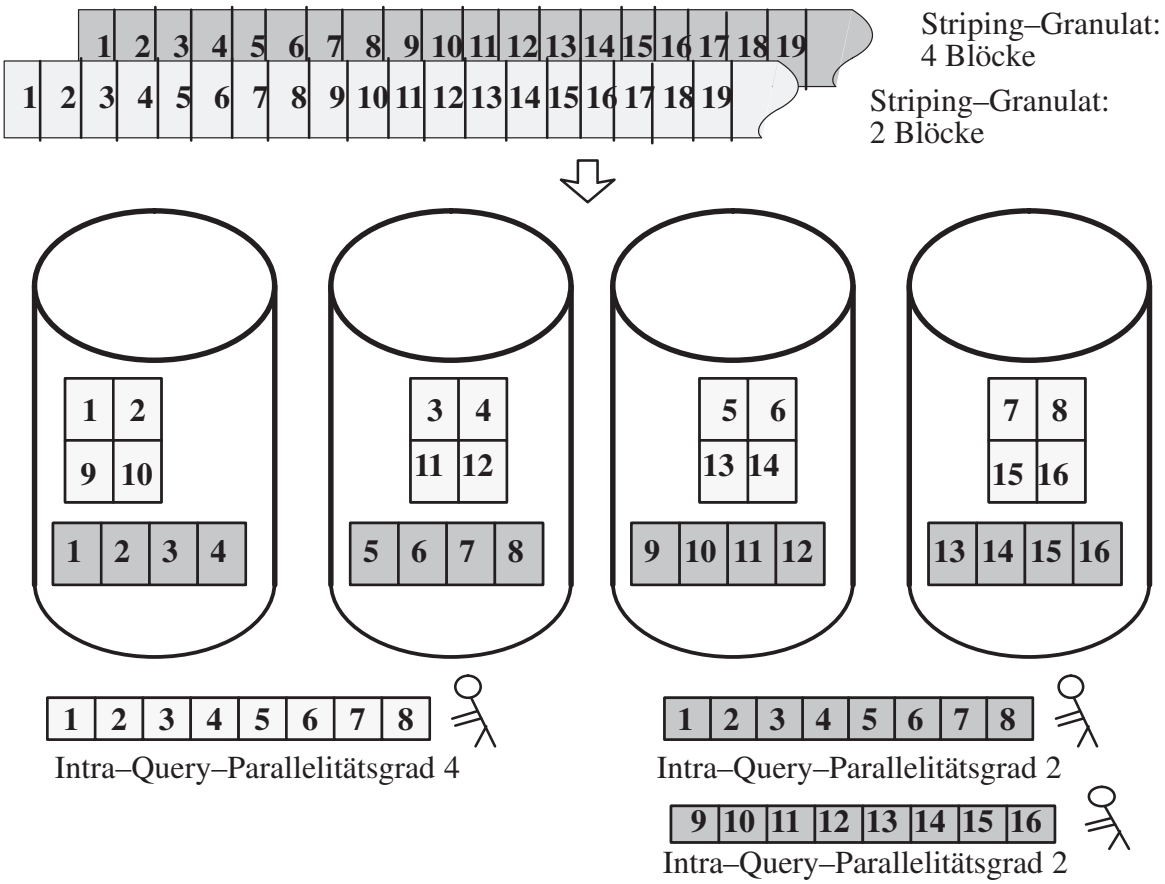
Bereichspartitionierung:
Der Wertebereich eines Attributs wird in n Bereiche $(-\infty,high_1)$, $[low_2,high_2)$, ..., $[low_n, \infty)$ unterteilt (also mittels eines einfachen Bereichsprädikats der Form low $\leq$ Attribut < high); alle Tupel, die in denselben Bereich fallen, bilden eine Partition.



Falls n größer ist als die Anzahl der Platten bzw. Verarbeitungsknoten, werden die resultierenden Partitionen in der Regel reihum (round−robin) plaziert.



Striping:
Der einer Relation zugrundeliegende Tablespace wird in Blocksequenzen konstanter Größe unterteilt; diese Blocksequenzen werden über die Platten verteilt. Eine solche Blocksequenz (d.h. die größte Sequenz logisch aufeinanderfolgender Blöcke, die auf derselben Platte liegen) heißt Striping−Granulat.



Die Zuordnung der Striping−Granulate zu den Platten erfolgt in der Regel reihum (round−robin). Bei einem Striping−Granulat von s Blöcken und d Platten mit den Nummern liegt Block i auf der Platte $\left\lfloor \frac{i}{s} \right\rfloor \bmod d$.
Alternativ könnte das DBS (oder das zugrundeliegende File−System) ein Verzeichnis mit der Zuordnung von Striping−Granulaten zu Platten verwalten.

Tuning der Partitionsgrößen

Je kleiner die Partitionen sind, umso stärker werden (Bereichs−) Queries oder Plattenzugriffe auf logisch aufeinanderfolgende Blöcke (z.B. bei Tablescans oder Zugriffen auf BLOBs) parallelisiert. Bis zu einem gewissen Grad ist diese Intra−Query−Parallelität sinnvoll, da sie die Antwortzeit einer einzelnen Query verringert; ab einem bestimmten Punkt aber reduziert sie aufgrund ihres mit der Parallelität steigenden Ressourcenverbrauchs den möglichen Durchsatz, also die Inter−Query−Parallelität.
Deswegen sollten die Partitionsgrößen in Abhängigkeit von den Lastcharakteristika und Leistungsanforderungen der jeweiligen Anwendung sorgfältig optimiert werden.

Gegeben seien
- D Platten bzw. Verarbeitungsknoten
- Aufträge (Queries, Plattenzugriffe), die mit einer Rate von X Aufträgen/Sekunde initiiert werden und jeweils auf einen Datenbereich der Größe R zugreifen (z.B. ein Intervall mit R Attributwerten bei einer Bereichs−Query oder R aufeinanderfolgende Blöcke bei einem Tablescan)

Die Bedienzeit S eines Auftrags in Abhängigkeit vom Intra−Query−Parallelitätsgrad P sei

$$S = a + bP + c\frac{R}{P}$$

mit entsprechenden Konstanten a, b, c, die die Latenz, den Synchronisations− bzw. Kommunikations−Overhead und die eigentliche Verarbeitungszeit reflektieren

Der Ressourcenverbrauch B ist dann

$$B = P * S$$

Der maximal erreichbare Durchsatz X_{max} (Anzahl der ausgeführten Aufträge pro Sekunde) ist

$$X_{\max} = \frac{D}{B}$$

Gesucht ist dann derjenige Wert P_{opt} von P, der S minimiert unter der Nebenbedingung $\lambda \leq X$.
Dieser Lösungsansatz ignoriert allerdings die mit λ ansteigende Konkurrenz um die Platten bzw. Verarbeitungsknoten und die daraus resultierenden Wartezeiten (Queueing Delays).
Um dies zu berücksichtigen, ist statt S die Antwortzeit T als Funktion von X und P zu betrachten.

10.1.3 Datenallokation und Lastbalancierung

Gegeben seien m zu allozierende Fragmente − Partitionen von Relationen, komplette Relationen, Indizes oder auch einzelne BLOBs − und d Platten bzw. Verarbeitungsknoten eines Parallelrechners. Für jedes Fragment j sei eine Maßzahl für die durch dieses Fragment ausgelöste Last gegeben, die "Hitze" H_j des Fragments. Die Hitze bestimmt sich (ungefähr) aus der Häufigkeit von Operationen, die auf das Fragment zugreifen, und der Arbeit auf dem Fragment (z.B. Tupel− oder Blockzugriffe), die dabei im Mittel anfällt.

Gesucht ist eine Allokation der Fragmente, so daß die resultierende Gesamtlast pro Platte bzw. Verarbeitungsknoten für alle Platten in etwa gleich ist. Ein entsprechendes Optimierungsziel ist die Minimierung der Gesamthitze der heißesten Platte.
Als formales Optimierungsproblem sieht dies folgendermaßen aus:

$$\text{Minimiere } \max\left\{ \sum_{j=0}^{m-1} x_{ij} * H_j \mid 0 \le i \le d-1 \right\}$$

$$\text{mit 0−1−Entscheidungsvariablen } x_{ij} = \begin{cases} 1 \textit{ falls Fragment j auf Platte i alloziert wird} \\ 0 \textit{ sonst} \end{cases}$$

und Nebenbedingungen bzgl. der Speicherkapazität einer Platte usw.

Dieses Optimierungsproblem ist NP−vollständig. Eine gutes approximatives Verfahren ist die *LPT−Heuristik* (Longest−Processing−Time−First), die im Bereich des Multiprozessor−Scheduling entwickelt wurde und für das Allokations− und Lastbalancierungsproblem sinngemäß Largest−Heat−First heißen müßte.
Dies ist eine einfache Greedy−Heuristik, die Allokationsentscheidungen in einer geschickten Reihenfolge trifft und einmal getroffene Entscheidungen nie mehr rückgängig macht. Sie verfährt in den folgenden Schritten:

1) Sortiere alle Fragmente nach absteigender Hitze H_j
   O.B.d.A. sei die Sortierreihenfolge 0, ..., m−1
2) Initialisiere die Gesamthitze aller Platten: GHi := 0 für i=0..d−1
3) for j := 0 to m−1 do
   a) Wähle diejenige Platte i mit dem kleinsten Wert GH_i
   (unter denjenigen Platten, die die spezifizierten Nebenbedingungen erfüllen)
   b) Alloziere Fragment j auf Platte i
   c) Erhöhe die Gesamthitze von Platte i: $GH_i := GH_i + H_j$

Theorem:
Sei GH_{opt} die Gesamthitze der heißesten Platte bei optimaler Allokation,
und sei GH_{LPT} die Gesamthitze der heißesten Platte bei einer Allokation nach der LPT−Heuristik. Dann gilt stets:

$$GH_{LPT} \le \left(\frac{4}{3} - \frac{1}{3d}\right) * GH_{opt}$$

Das durchschnittliche Verhalten der LPT−Heuristik ist wesentlich besser als die Worst−Case−Schranke des Theorems.

10.2 Weitere Tuning−Maßnahmen

10.2.1 Query−Tuning

In der Praxis neigt man oft dazu, Schwächen des jeweiligen Query−Optimierers durch explizite Umformulierung von Queries oder zusätzliche "Optimizer−Hints" auszugleichen, um den Ausführungsplan zu beeinflussen. Diese Vorgehensweise verletzt die Programm−Daten−Unabhängigkeit und kann langfristig zu drastisch höheren Kosten bei der Wartung komplexer Anwendungen führen.

Beispiele für "Optimizer−Hints" in Oracle:

- SELECT /*+ FULL(Produkte) */ PNr, Bez, Preis
  FROM Produkte WHERE Preis < 100

  erzwingt einen Tablescan, auch wenn ein Index auf Preis existiert

- SELECT /*+ ORDERED FULL(B)
  USE_MERGE(K) ORDERED FULL(K)
  USE_NL(P) INDEX(P, ProduktePNrIndex) */
  P.Bez, K.Name
  FROM Bestellungen B, Produkte P, Kunden K
  WHERE B.PNr=P.PNr AND B.KNr=K.KNr AND K.Rabatt > 40

  erzwingt einen Sort−Merge−Join mit vorhergehenden Tablescans zwischen B und K und anschließend einen Nested−Loop−Join mit Indexzugriff auf die innere Relation P

10.2.2 Datenpartitionierung und −allokation in verteilten DBS

Gegeben seien
- ein konzeptionelles Datenbankschema,
- ein Mengengerüst der Datenbank,
- eine Lastbeschreibung (Queries, Transaktionen) sowie
- eine (kleine) Menge verteilter Server−Rechner, die über ein WAN verbunden sind und jeweils eine (große) Menge lokaler Clients bedienen.

Für jeden Server seien Häufigkeiten gegeben, mit denen bestimmte Queries bzw. Transaktionen auf diesem Server initiiert werden.

Gesucht ist eine Partitionierung der Relationen sowie eine Allokation der resultierenden Partitionen, d.h. eine Zuteilung zu den Server−Rechnern, so daß die Gesamtkosten der Last minimiert werden. Die Gesamtkosten sind dabei häufig durch die Kommunikationskosten im WAN dominiert.

Die Partitionierung der Relationen kann horizontal mittels Selektionen, vertikal mittels Projektionen oder gemischt erfolgen. Bei der Allokation ist auch die Replikation von Partitionen möglich.

10.2.3 Materialisation von Views

Views, auf denen häufig Queries ausgeführt werden und deren Berechnung (auf den Basisrelationen) selbst sehr aufwendig ist, sollten u.U. materialisiert werden. Dabei wird die Ausprägung der View einmal berechnet und als eigene Relation abgespeichert.

Änderungen der zugrundeliegenden Basisrelationen werden entweder periodisch in die materialisierte View eingebracht (u.U. einfach durch komplette Neuberechnung der View−Ausprägung), z.B. einmal am Tag, oder kontinuierlich und inkrementell durch Extraktion relevanter Daten aus den Änderungsoperationen, die auf den Basisrelationen ausgeführt werden. Letzteres kann durch Inspektion des Log−Files oder durch Trigger auf den Basisrelationen realisiert werden.

View−Materialisation ist in keinem DBS direkt unterstützt, sondern erfordert spezielle Anwendungsprogramme oder andere Zusatzkomponenten. Diese werden vor allem für sogenannte "Data Warehouses" angeboten. Ein Data Warehouse umfaßt eine Kollektion materialisierter Views, die eine kompakte, häufig aggregierte Sicht auf die Produktionsdaten (bzw. Vertriebsdaten) eines Unternehmens als Entscheidungsunterstützung zur Verfügung stellen (z.B. für Marketing−Zwecke).

Beispiel:

Gegeben sei eine Datenbank mit dem folgenden Schema:

Bestellungen (BestNr, Datum, KNr, PNr, Menge, Summe, ...)
Kunden (KNr, Name, Adresse, Stadt, ...)
Staedte (Stadt, Region)
Regionen (Region, Land)
Produkte (PNr, Bez, Preis, Sparte, ...)

Typischerweise ist dabei die Relation "Bestellungen" sehr groß (u.U. $> 10^8$ Tupel), und die anderen Relationen sind relativ dazu sehr klein.

Für Marketing−Zwecke sei die folgende View von Interesse:

```
CREATE VIEW Vertrieb AS
SELECT B.PNr, Bez, Sparte, Stadt, Region, Land, SUM(Summe) AS Umsatz
FROM Bestellungen B, Staedte S, Regionen R, Produkte P
WHERE B.KNr=K.KNr AND K.Stadt=S.Stadt AND S.Region=R.Region
AND B.PNr=P.PNr
GROUP BY B.PNr, Stadt [, Bez, Sparte, Region, Land]
```

Wenn diese View materialisiert wird, können damit Queries der folgenden Art sehr effizient ausgeführt werden:

- SELECT PNr, Bez, Region, SUM(Umsatz) FROM Vertrieb GROUP BY PNr, Bez, Region
  WHERE Sparte='Wintersportartikel'
- SELECT PNr, Bez, Land SUM(Umsatz) FROM Vertrieb GROUP BY PNr, Bez, Land
  WHERE Sparte='Wintersportartikel'

usw.

Das Tuning−Problem besteht darin, festzulegen, welche Views materialisiert werden sollen (und in welchem Modus sie aktualisiert werden sollen).

10.2.4 Limitierung des Mehrbenutzerbetriebs

Im Mehrbenutzerbetrieb konkurrieren parallele Queries und Transaktionen um Ressourcen. Wenn unbeschränkt viele Queries bzw. Transaktionen zur gleichzeitigen Ausführung zugelassen werden, kann dies zu einer Leistungskatastrophe führen, dem sogenannten "Thrashing", bei dem die Antwortzeiten drastisch anschnellen und das DBS mehr oder weniger zusammenbricht.

Um dies zu verhindern, bieten alle DBS die Möglichkeit, die maximale Anzahl gleichzeitig bearbeiteter Queries bzw. Transaktionen – das sogenannte "Multiprogramming Level" (MPL) – zu limitieren.

In Verbindung mit einem TP–Monitor, der die Verbindungen zwischen Clients und dem DBS–Server mittels verschiedener Queues verwaltet, kann ein solches Limit u.U. spezifisch für bestimmte Klassen von Queries bzw. Transaktionen festgelegt werden (z.B. max. 20 einfache Queries, max. 5 komplexe Queries, max. 10 Änderungstransaktionen gleichzeitig – sowie u.U. auch Limiten für Kombinationen).

Die geeignete Wahl dieser Limiten ist mehr oder weniger eine schwarze Kunst (von speziellen Fällen abgesehen).

10.2.5 Parallelitätsgrade von Queries

In parallelen DBS kann man in der Regel einen maximalen Parallelitätsgrad für eine parallelisierte Query spezifizieren, beispielsweise in Form von "Optimizer–Hints". Der Query–Optimierer wird dann höchstens entsprechend viele Prozesse bzw. Threads für die Ausführung der Query vorsehen und damit die Anzahl der gleichzeitig benutzten Prozessoren und Platten beschränken. Dies kann einerseits sinnvoll sein, wenn Queries in ihrem überhaupt möglichen Speedup inhärent beschränkt sind, und kann andererseits notwendig sein, um im Mehrbenutzerbetrieb eine Monopolisierung von Ressourcen durch einzelne Queries zu verhindern.

Um die Konkurrenzsituation im Mehrbenutzerbetrieb zu beeinflussen, gibt es in einigen parallelen DBS außerdem die Wahl zwischen zwei verschiedenen Optimierungskriterien des Query–Optimierers: Minimierung der Antwortzeit der einzelnen Query (u.U. um den Preis eines sehr hohen Ressourcenverbrauchs) oder Minimierung des Ressourcenverbrauchs (und damit Maximierung des Durchsatzes im Mehrbenutzerbetrieb).

Beispiele für derartige "Optimizer–Hints" in Oracle:

- SELECT /*+ FULL(Produkte) PARALLEL(Produkte,4) */ PNr, Bez, Preis
  FROM Produkte WHERE Preis < 10
  limitiert den Parallelitätsgrad für den (erzwungenen) Produkte–Tablescan auf max. 4
- SELECT /*+ FIRST_ROWS */ PNr, Bez, Preis FROM Produkte WHERE Preis < 10
  minimiert die Antwortzeit dieser Query (für die Rückgabe des ersten Resultattupels)
- SELECT /*+ ALL_ROWS */ PNr, Bez, Preis FROM Produkte WHERE Preis < 10
  minimiert den Ressourcenverbrauch der Query

Ergänzende Literatur:

S. Finkelstein, M. Schkolnick, P. Tiberio, Physical Database Design for Relational Databases, ACM Transactions on Database Systems Vol.13 No.1, 1988

S. Rozen, D. Shasha, A Framework for Automating Physical Database Design, International Conference on Very Large Data Bases, Barcelona, 1991

S. Chaudhuri, V. Narasayya, An Efficient Cost−Driven Index Selection Tool for Microsoft SQL Server, International Conference on Very Large Data Bases (VLDB), Athens, 1997

P. Scheuermann, G. Weikum, P. Zabback, Data Partitioning and Load Balancing in Parallel Disk Systems, VLDB Journal Vol.7 No.1, 1998

D.E. Shasha, Database Tuning − A Principled Approach, Prentice−Hall, 1992

P. Corrigan, M. Gurry, Oracle Performance Tuning, O'Reilly & Associates, Inc., 1993

M.J. Corey, M. Abbey, D.J. Dechichio Jr., Tuning Oracle, Oracle Press, 1995

J.A. Blakeley, N. Coburn, P.A. Larson, Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates, ACM Transactions on Database Systems Vol.14 No.3, 1989

Bulletin of the IEEE TC on Data Engineering Vol.18 No.2, June 1995, Special Issue on Materialized Views and Data Warehousing

D. Theodoratos, T. Sellis, Data Warehouse Configuration, International Conference on Very Large Data Bases (VLDB), Athens, 1997

W. Labio, D. Quass, B. Adelberg, Physical Database Design for Data Warehouses, International Conference on Data Engineering (ICDE), Birmingham, UK, 1997

G. Weikum, C. Hasse, A. Moenkeberg, P. Zabback, The COMFORT Automatic Tuning Project, Information Systems Vol.19 No.5, 1994