

---

# Kapitel 7

## Datenbank-Tuning

---

Folien zum Datenbankpraktikum  
Wintersemester 2012/13 LMU München

© 2008 Thomas Bernecker, Tobias Emrich © 2010 Tobias Emrich, Erich Schubert  
unter Verwendung der Folien des Datenbankpraktikums aus dem Wintersemester 2007/08 von Dr. Matthias Schubert

# Übersicht

7.1 Ziele und Tuning-Ebenen

7.2 Relationenschema

7.3 Cluster

7.4 Verwendung von Indexstrukturen

## Ziele von DB-Tuning

- Verkürzung der Antwortzeit (Online Analytical Processing - **OLAP**):  
Wichtig bei Einbenutzer-Datenbankapplikationen (z.B. Data Mining): der Benutzer möchte möglichst schnell eine Antwort auf seine Anfrage.
- Erhöhung des Durchsatzes (Online Transaction Processing - **OLTP**):  
Wichtig bei Hochleistungs-Transaktionssystemen (z.B. Flugbuchung): pro Zeiteinheit sollen möglichst viele Anfragen bearbeitet werden.
- Zielkonflikt:  
Zwischen der Verkürzung der Antwortzeit und der Erhöhung des Durchsatzes besteht häufig ein Zielkonflikt, da z.B. die Verlagerung von Programmfunktionalität in den DB-Server (gespeicherte Prozeduren) die Antwortzeit im Einbenutzermodus verkürzt, dagegen im Mehrbenutzermodus den Durchsatz hemmt.

## Tuning-Ebenen (Auszug)

- Relationenschema: Normalisierung, Denormalisierung
- Physisches DB-Schema: Cluster, Index
- Anfrageoptimierung: Query Rewriting, Hinweise an den SQL-Optimierer (mehr: Vortrag von Gruppe E am 6.12.)
- Client-Server-Architektur
- Parallelisierung und Verteilung (mehr: Kapitel 8)
- Transaktionsdesign
- Betriebssystemparameter und DB-Voreinstellungen: Größe eines DB-Blocks, Größe des DB-Cache

# Übersicht

7.1 Ziele und Tuning-Ebenen

7.2 Relationenschema

7.3 Cluster

7.4 Verwendung von Indexstrukturen

## Normalisierung

- Wenn auf bestimmte Attribute einer Relation erheblich häufiger zugegriffen wird, als auf andere (speicherintensive) Attribute der selben Relation, kann es günstig sein, die Relation zu zerlegen.

- Beispiel:

ERSATZTEILLAGER (TeileNr, Bestand, Teilebeschreibung)

Wenn auf Bestand viel häufiger zugegriffen wird als auf Teilebeschreibung, dann ist folgendes Relationenschema günstiger:

LAGERBESTAND (TeileNr, Bestand)

TEILE (TeileNr, Beschreibung)

- Zerlegung ist z.B. bei Joins mit einer anderen Relation günstiger im Leistungsverhalten, aber auch platzintensiver. Durch Zerlegung über *Composite Indexes* kann i.d.R. ein noch besserer Leistungsgewinn erzielt werden (siehe Abschnitt 7.4).

## Denormalisierung:

- Vermeidung von Joins (“Materialisierter Join”)
- Beispiel:

PRAKTIKUMSTEILNEHMER (PraktikumsNr, MatrikelNr)

STUDENT (MatrikelNr, Name, Adresse, Telefonnummer)

Bei den meisten Zugriffen auf die PRAKTIKUMSTEILNEHMER wird auch gleichzeitig der Name des Studenten benötigt (Join). Eine redundante Speicherung von Name kann daher sinnvoll sein:

PRAKTIKUMSTEILNEHMER (PraktikumsNr, MatrikelNr, Name)

STUDENT (MatrikelNr, Name, Adresse, Telefonnummer)

- aber: Gefahr von Update-Anomalien
  - Formulieren von Integritätsbedingungen
  - Überwachung durch Trigger (aufwändig)

→ Problem auf physischer Ebene lösen: *Cluster verwenden*

# Übersicht

7.1 Ziele und Tuning-Ebenen

7.2 Relationenschema

**7.3 Cluster**

7.4 Verwendung von Indexstrukturen



# Cluster

- physisch vorberechneter Join
- redundanzfrei (im Gegensatz zur Denormalisierung)
- Zwei oder mehr Relationen werden ineinander verschränkt gespeichert
- aber: spürbar erhöhte Update-Kosten
- Beispiel:

```
BESTELLUNG (KundenNr, TeileNr, Menge)
```

```
KUNDEN (KundenNr, Name)
```

Speicherorganisation eines Clusters auf KundenNr:

```
(KundenNr1, Name1) [(TeileNr1,1, Menge1,1), (TeileNr1,2, Menge1,2),  
                    (TeileNr1,3, Menge1,3), ... ],
```

```
(KundenNr2, Name2) [(TeileNr2,1, Menge2,1), ...], ...
```

- Schwierige Speicherplatzorganisation, daher nicht in allen RDBMS verfügbar
- Syntax in *Oracle*:

```
CREATE CLUSTER <clustername> (attribut1 typ1, attribut2 typ2, ...);
```

```
DROP CLUSTER <clustername>;
```

```
ALTER CLUSTER <clustername> ...;
```

- Beispiel Praktikum/Student:

```
CREATE CLUSTER psc (MatrikelNr NUMBER) ;
```

```
CREATE TABLE student (  
    MatrikelNr NUMBER PRIMARY KEY,  
    Name VARCHAR2(20),  
    Adresse VARCHAR2(40),  
    TelNr NUMBER
```

```
) CLUSTER psc (MatrikelNr);    -- hinzufügen zum Cluster
```

```
CREATE TABLE praktikumsteilnehmer (  
    PraktikumsNr NUMBER NOT NULL,  
    MatrikelNr NUMBER REFERENCES student,  
    PRIMARY KEY (PraktikumsNr, MatrikelNr)
```

```
) CLUSTER psc (MatrikelNr);    -- hinzufügen zum Cluster
```

```
CREATE INDEX psc_idx ON CLUSTER psc ;
```

*Erst jetzt können Tupel in die Relationen eingefügt werden.*

# Übersicht

7.1 Ziele und Tuning-Ebenen

7.2 Relationenschema

7.3 Cluster

7.4 Verwendung von Indexstrukturen

## Indexstrukturen

- Relation ohne Index: ungeordnete Ansammlung einzelner Tupel  
⇒ Zugriff: vollständiger Durchlauf der gesamten Relation nötig (*full table scan*)
- Index: Datenstruktur (z.B. B<sup>+</sup>-Baum oder Hashtabelle) mit Verweisen auf die Tupel in spezifizierter Sortierordnung zur effizienten Anfragebearbeitung
- *Oracle* erzeugt automatisch einen Index auf den Attributen des Primärschlüssels
- Fremdschlüssel: keine automatische Indexunterstützung  
⇒ explizit erzeugen, wenn nicht identisch mit dem gesetzten Primärschlüssel
- Effiziente Anfragebearbeitung über Nicht-Schlüsselattributen:  
⇒ explizit erzeugen
- mehr dazu: Vorlesung *Anfragebearbeitung und Indexstrukturen in Datenbanksystemen*

## Index: Syntax

- Index-Generierung:

```
CREATE INDEX <index-name> ON <table> (attr_1, attr_2,  
                                     ..., attr_n);
```

- Ein **Composite Index** besteht aus mehr als einer Spalte. Die Tupel sind dann nach den Attributwerten (lexikographisch) geordnet:

$t_1 < t_2$  gdw.  $t_1.a_1 < t_2.a_1$  oder  $(t_1.a_1 = t_2.a_1$  und  $t_1.a_2 < t_2.a_2)$  oder ...

Für den Vergleich der einzelnen Attribute gilt die jeweils übliche Ordnung: numerischer Vergleich für numerische Typen, lexikographischer Vergleich bei CHAR, Datums-Vergleich bei DATE usw.

- Löschen eines Index:

```
DROP INDEX <index-name>;
```

- Verändern eines Index: (betrifft u.a. Speicherungs-Parameter und Rebuild)

```
ALTER INDEX <index-name> ...;
```

## Durch Index unterstützte Anfragen

- Exact match query für Attribute  $a_1$  bis  $a_n$ :

```
SELECT * FROM t WHERE a1=... AND ... AND an=...
```

- Partial match query:

```
SELECT * FROM t WHERE a1=... AND ... AND ai=...
```

*für  $i < n$ , d.h. wenn die exakt spezifizierten Attribute ein Präfix der indizierten Attribute sind. Eine Spezifikation von  $a_{i+1} = a_j$  kann i.A. nicht genutzt werden, wenn  $a_i$  nicht spezifiziert ist.*

- Range query:

```
SELECT * FROM t WHERE a1=... AND ... AND aj<=...
```

*auch z.B. für '>' oder 'BETWEEN'*

- Point Set query:

```
SELECT * FROM t WHERE a1=... AND ... AND aj IN (7,17,77)
```

*auch z.B. (aj=... OR aj=... OR ...)*

- Pattern matching query:

```
SELECT * FROM t WHERE a1=... AND ... AND aj LIKE 'abc%'
```

- **Function-based Index:** Aufbau nicht auf den Attributen direkt, sondern nach Anwendung einer Funktion auf ihnen

## Index-unterstützte Anfragebearbeitung

- Für manche Anfragen reicht der Indexzugriff zur Beantwortung aus, z.B.
  - `SELECT COUNT(*) FROM t WHERE a1=...`
  - `... WHERE EXISTS (SELECT ...) ...`
  - `SELECT a1, a2, ..., ai FROM t WHERE a1=... AND ...`
- ⇒ Index umfasst alle Attribute aus `select-List` und `where-Klausel`
- Für die meisten Anfragen ist aber eine sog. *tuple materialization* erforderlich: Hierbei wird über die *RowID* (= physische Position des Datensatzes in der DB) auf das Tupel zugegriffen; i.A. ist dann pro Ergebnistupel ein zusätzlicher Plattenzugriff erforderlich.
- Abhilfe: *Index Organized Tables (IOT)* speichern die Tupel vollständig in den Datenseiten der Indexstruktur (B<sup>+</sup>-Baum) ⇒ Zugriff durch logische *RowID*



## Weitere Eigenschaften von Indexen

- Effizientere Anfragebearbeitung, aber INSERT/DELETE/UPDATE müssen auch auf dem Index ausgeführt werden.  
⇒ Schlecht: viele Updates auf vielen Indexen
- „Größere“ Updates (z.B. Einlesen der gesamten Relation oder großer Teile aus einer Datei): Index vor dem Einlesen zu löschen und nachher neu aufzubauen („bulk load“) ist oft günstiger, u.U. auch für impliziten Primärschlüsselindex

```
DROP INDEX i;
```

```
UPDATE t SET ... ;
```

```
CREATE INDEX i ON t (...);
```

oder:

```
ALTER TABLE t DROP PRIMARY KEY;
```

```
INSERT INTO t ... ;
```

```
ALTER TABLE t ADD CONSTRAINT p PRIMARY KEY (...);
```

- Indexe lohnen sich nur bei großen Relationen, die nicht in den DB-Cache passen.