Kapitel 7
Datenbank-Tuning

Folien zum Datenbankpraktikum
Wintersemester 2010/11 LMU München

unter Verwendung der Folien des Datenbankpraktikums aus dem Wintersemester 2007/08 von Dr. Matthias Schubert

Übersicht

Ziele von DB-Tuning

- Verkürzung der Antwortzeit (Online Analytical Processing - ***OLAP***):

  Wichtig bei Einbenutzer-Datenbankapplikationen (z.B. Data Mining): der Benutzer möchte möglichst rasch 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*: Clustering, Index
- *Anfrageoptimierung*: Query Rewriting, Hinweise an den SQL-Optimierer
- *Client-Server-Architektur*
- *Parallelisierung und Verteilung*
- *Transaktionsdesign*
- *Betriebssystemparameter und DB-Voreinstellungen*: Größe eines DB-Blocks, Größe des DB-Cache

Übersicht

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

Cluster

- Ein Cluster ist ein physisch vorberechneter Join.
- Im Gegensatz zur Denormalisierung ist er redundanzfrei.
- Zwei oder mehr Relationen werden ineinander verschränkt gespeichert.
- Allerdings ergeben sich spürbar erhöhte Update-Kosten.
- Beispiel:

  `BESTELLUNG (KundenNr, TeileNr, Menge)`

  `KUNDEN (KundenNr, Name)`

  Speicherorganisation eines Clusters auf `KundenNr`:

  $(KundenNr_1, Name_1)\ [(TeileNr_{1,1}, Menge_{1,1}),(TeileNr_{1,2}, Menge_{1,2}),$
  $(TeileNr_{1,3}, Menge_{1,3})\ \ldots\ ],$

  $(KundenNr_2, Name_2)\ [(TeileNr_{2,1}, Menge_{2,1}),\ \ldots],\ \ldots$
- 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 (matrnr NUMBER(15)) ;

CREATE TABLE student (
    matrnr NUMBER (15) PRIMARY KEY,
    name char(80)
) CLUSTER psc (matrnr) ;

CREATE TABLE praktikum (
    praktnr NUMBER (4) NOT NULL,
    matrnr NUMBER (15) NOT NULL,
    PRIMARY KEY (matrnr, praktnr)
) CLUSTER psc (matrnr) ;

CREATE INDEX psc_idx ON CLUSTER psc ;
```

Erst jetzt können Tupel in die Relationen eingefügt werden.

Übersicht

Indexstukturen

- Werden die Relationen ohne Index gespeichert, dann bestehen sie nur aus einer ungeordneten Ansammlung einzelner Tupel. Jeder Zugriff erfordert dann einen vollständigen Durchlauf der gesamten Relation (*full table scan).*
- Ein Index ist eine Datenstruktur (z.B. B^+-Baum oder Hash Table), in der Verweise auf die Tupel in einer spezifizierten Sortierordnung gespeichert werden, um effiziente Anfragebearbeitung zu ermöglichen.
- *ORACLE* erzeugt automatisch einen Index auf den Attributen des Primärschlüssels, wenn ein `Primary Key`-Constraint gesetzt ist.
- Fremdschlüssel-Beziehungen werden nicht automatisch durch einen Index unterstützt; dieser muss explizit erzeugt werden, wenn er nicht identisch mit dem gesetzten Primärschlüssel ist.
- Sollen Anfragen über Nicht-Schlüsselattributen effizient bearbeitet werden, so muss ebenfalls explizit ein Index aufgebaut werden.

→ mehr dazu: Vorlesung *Index- und Speicherungsstrukturen für Datenbanksysteme*

Index: Syntax

- Index-Generierung:

```
CREATE INDEX <index-name> ON <table> (attr1, attr2,
                                      ..., attrn);
```

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

  $t_1 < t_2$ *gdw.* $t_1.a_1 < t2.a_1$ *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 *kann i.a. nicht genutzt werden, wenn* a_i *nicht spezifiziert ist.*

- Range query:

```
SELECT * FROM t WHERE a1=... AND ... AND ai=... AND ai+1<=...
```

  auch z.B. für '>' oder 'BETWEEN'

- Point Set query:

```
SELECT * FROM t WHERE a1=... AND ... AND ai=...
                      AND ai+1 IN (7,17,77)
```

auch z.B. (`ai+1=... OR ai+1=... OR ...`)

- Pattern matching query:

```
SELECT * FROM t WHERE a1=... AND ... AND ai=...
                      AND ai+1 LIKE 'c1c2...ck%'
```

Problem: Anfragen wie

```
wort LIKE '%system'
```

werden nicht unterstützt. Man kann aber z.B. eine Relation aufbauen, in der alle Wörter revers gespeichert werden und dann effizient nach

```
revers_wort LIKE 'metsys%'
```

suchen lassen. Dies lässt sich auch mit einem ***Function-based Index*** (wird nicht auf den Attributen direkt, sondern nach Anwenung einer Funktion auf ihnen erzeugt) elegant lösen.

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.

Weitere Eigenschaften von Indexen:

- Anfrage wird effizienter bearbeitet, aber INSERT/DELETE/UPDATE müssen auch auf dem Index ausgeführt werden. Das Leistungsverhalten verschlechtert sich, wenn viele Updates auf vielen Indexen durchgeführt werden.
- Bei „größeren" Updates (Einlesen der gesamten Relation oder großer Teile aus einer Datei) kann es günstiger sein, den Index vor dem Einlesen zu löschen und nachher neu aufzubauen („bulk load"). Dies gilt u.U. auch für den 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. Wenn eine Relation im Arbeitsspeicher einer Session („DB-Cache") Platz hat, kann sie auch ohne Index schnell durchsucht werden.