DB-Tuning

Tuning des konzeptuellen Schemas

- Das Design des konzeptuellen Schemas ist beeinflußt durch das Lastprofil der Applikation zusätzlich zur Frage der Redundanz.
- Überlegungen:
  - 3NF Schema besser geeignet als BCNF?
  - Lastprofil hat Einfluß auf die Entscheidung über Dekomposition der Relation in 3 NF oder BCNF
  - Weitere Dekomposition eines BCNF Schemas sinnvoll?
  - *Denormalisierung* eines Schemas sinnvoll (d.h. ein Dekompositionsschritt wird rückgängig gemacht)
  - Hinzufügen von Feldern zu einer Relation
  - *Horizontale Dekomposition* (Partitionierung der Tupelmengen einer Relation)
- *Schema-Evolution:* Veränderung des Schemas einer Datenbank, die bereits operativ ist (kann einige Probleme verursachen!)
- Verbergen einiger dieser Änderungen vor den Applikationen durch die Definition von Sichten (*Views*)

Beispiel-Schema

Contracts (Cid, Sid, Jid, Did, Pid, Qty, Val)
Depts (Did, Budget, Report)
Suppliers (Sid, Address)
Parts (Pid, Cost)
Projects (Jid, Mgr)

- Beispiel Contracts, abgekürzt: CSJDPQV. Folgende Integritätsbedingungen (*Integrity Constraint IC*) sind gegeben:

  JP→C, SD →P, C ist Primärschlüssel

  – Was sind die Schlüsselkandidaten für CSJDPQV?
  – In welcher Normalform befindet sich das Relationenschema?

Festlegen 3NF oder BCNF

- CSJDPQV kann zerlegt werden in SDP und CSJDQV, und beide Relationen sind in BCNF.
  - Verlustfreie Dekomposition, aber nicht abhängigkeitsbewahrend
  - Hinzufügen von CJP sichert die Eigenschaft abhängigkeits-bewahrend
- Annahme, daß folgende Query sehr wichtig ist:
  - *Finde die Anzahl der Kopien, Q, des Teils P , das im Vertrag (Contract) C bestellt wurde.*
  - Erfordert einen Join auf dem dekomponierten Schema, aber kann genauso beantwortet werden durch einen Scan auf der Original-Relation CSJDPQV.
  - Könnte zu der Festlegung führen, das 3NF Schema CSJDPQV zu verwenden

Denormalisierung

- Angenommen, folgende Anfrage ist wichtig:
  - *Ist der Wert eines Vertrages geringer als das Budget der Abteilung?*
- Um diese Anfrage zu beschleunigen, könnten wir ein Feld *budget* B zur Relation Contracts hinzufügen
  - Dies erzeugt eine neue funktionale Abhängigkeit FD $D \rightarrow B$ innerhalb der Relation Contracts.
  - Somit gilt: Contracts ist nicht mehr in 3NF.
- Wir könnten entscheiden, Contracts zu modifizieren, wenn die Anfrage hinreichend wichtig und die Performance anderweitig nicht gesteigert werden kann (d.h. durch Anlegen neuer Indexe oder durch eine alternatives 3NF Schema)

Wahl der Dekomposition

- 2 Möglichkeiten, CSJDPQV in die BCNF zu zerlegen:
  - SDP and CSJDQV; verlustfreier Join, aber nicht abhängigkeitsbewahrend
  - SDP, CSJDQV and CJP; zusätzlich noch abhängigkeitsbewahrend
- Der Unterschied zwischen beiden ist der Aufwand, um die FD JP $\rightarrow$ C zu kontrollieren (interrelationale Constraints verursachen hohe Kosten)
  - 2. Dekomposition: Index auf JP in Relation CJP (JP ist dort Primärschlüssel)
  - 1. Dekomposition:

```
CREATE ASSERTION CheckDep
CHECK ( NOT EXISTS ( SELECT *
FROM PartInfo P, ContractInfo C
WHERE P.sid=C.sid AND P.did=C.did
GROUP BY C.jid, P.pid
HAVING COUNT (C.cid) > 1 ))
```

Wahl der Dekomposition (Forts.)

- Die folgenden ICs sollen kontrolliert werden: JP $\rightarrow$ C, SD $\rightarrow$ P, C ist Primärschlüssel
- Zusätzlich sei angenommen: Ein bestimmter Lieferant berechnet immer den gleichen Preis für ein bestimmtes Teil SPQ $\rightarrow$ V
- Wenn CSJDPQV in die BCNF zerlegt werden soll, gibt es nun eine dritte Möglichkeit:
  - Beginne mit der Zerlegung in SPQV and CSJDPQ
  - Dann zerlege CSJDPQ (nicht in 3NF) in SDP, CSJDQ
  - Dies führt zu der verlustfreien Dekomposition: SPQV, SDP, CSJDQ.
  - Um JP $\rightarrow$ C zu kontrollieren, können wir wieder die Relation CJP hinzufügen
- Wahl: { SPQV, SDP, CSJDQ } oder { SDP, CSJDQV } ?
- Mit der Variante { SDP, CSJDQV }: In BCNF und braucht nicht weiter zerlegt zu werden (Annahme, daß alle bekannten ICs FDs sind)
- Was ist, wenn folgende Anfragen wichtig sind?
  - *Finde die Verträge mit dem Lieferanten S.*
  - *Finde die Verträge, an denen Abteilung D beteiligt ist.*
- Weitere Zerlegung von CSJDQV in CS, CD and CJQV könnte diese Queries beschleunigen
- Andererseits können Anfragen verlangsamt werden, z.B.:
  - *Finde den Gesamtwert aller Verträge mit dem Lieferanten S.*

Horizontale Dekomposition

- Bisherige Definition von Dekomposition: Relation wird durch eine Menge von Relationen ersetzt, die *Projektionen* sind (wichtigster Fall)
- Manchmal möchten wir eine Relation durch eine Menge von Relationen ersetzen, die *Selektionen* sind
  - Jede neue Relation hat gleiches Schema wie Original-Relation, aber nur eine Teilmenge von Tupeln
  - Zusammen enthalten alle neuen Relationen alle Tupel der Original-Relation, Zerlegung ist disjunkt
- Angenommen, daß große Verträge (value > 10000) anderen Regeln unterliegen: Somit würden Abfragen häufig die Bedingung *val>10000* enthalten
- Erster Ansatz: Geclusterter B+ Baum-Index auf dem Feld *val* in Contracts
- Zweiter Ansatz: Ersetze Contracts durch zwei neue Relationen: LargeContracts und SmallContracts, mit den gleichen Attributen (CSJDPQV).
  - Verhält sich wie Index auf solchen Anfragen, aber ohne Index-Overhead
  - Zusätzlich geclusterter Index auf anderen Attributen denkbar

Verbergen von Schemaänderungen

```
CREATE VIEW Contracts(cid, sid, jid, did, pid, qty, val)
        AS SELECT *
        FROM LargeContracts
        UNION
        SELECT *
        FROM SmallContracts
```

- Die Ersetzung von Contracts durch LargeContracts und SmallContracts kann durch eine View verborgen werden
- Jedoch müssen Anfragen mit der Bedingung *val>10000* an die Relation LargeContracts gestellt werden für effiziente Ausführung
  - → Benutzer, für die Performance wichtig ist, müssen von Veränderung erfahren

Tuning von Queries und Views

- Wenn eine Anfrage langsamer als erwartet läuft:
  - Muß Index reorganisiert werden?
  - Ist die Statistik zu alt?
- Vielleicht führt das DBMS einen anderen Ausführungsplan als gewünscht aus.

  Mögliche Schwächen:
  - Selektionen mit Nullwerten
  - Selektionen mit arithmetischen oder String-Ausdrücken
  - Selektionen mit OR Bedingungen
  - Mangel Ausführungsmerkmalen für die Anfrage wie Index-Only-Strategien oder bestimmte Join-Methoden oder schlechte Größenabschätzung
- Prüfe den Plan, der verwendet wird! Dann korrigiere die Wahl des Index oder modifiziere die Query/View (Rewrite)

Umschreiben von SQL-Queries*

- Wird komplizierter durch die Interaktion von:
  - NULLs, Duplikaten, Aggregation, Subqueries
- *Richtlinie: Verwende nur einen "Query Block", wenn möglich*

```
SELECT DISTINCT *
  FROM Sailors S
 WHERE S.sname IN
       (SELECT Y.sname
          FROM YoungSailors Y)
```

=

```
SELECT DISTINCT S.*
  FROM Sailors S,
       YoungSailors Y
 WHERE S.sname = Y.sname
```

- *Nicht immer möglich ...*

```
SELECT *
  FROM Sailors S
 WHERE S.sname IN
       (SELECT DISTINCT Y.sname
          FROM YoungSailors Y)
```

≠

```
SELECT S.*
  FROM Sailors S,
       YoungSailors Y
 WHERE S.sname = Y.sname
```

Entschachteln von Anfragen (Unnesting)*

- DISTINCT auf Top-Level: *Kann Duplikate ignorieren*
  - Manchmal kann DISTINCT auf Top-Level abgeleitet werden! (z.B. Subquery liefert nur ein einziges Tupel)
- DISTINCT in Subquery ohne DISTINCT auf Top-Level: *Schwer zu konvertieren*
- Subqueries innerhalb von OR: *Schwer zu konvertieren*
- ALL Subqueries: *Schwer zu konvertieren*
  - EXISTS and ANY sind wie IN
- Aggregate in Subqueries: *Tricky.*
- Good News:
  Einige DBMS können selbst Anfragen konvertieren, ohne daß der Benutzer das merkt (z.B. DB2)

Weitere Tips zum Tuning von Queries

- Minimiere die Verwendung von DISTINCT: wird nicht gebraucht, wenn Duplikate akzeptable sind, oder wenn das Resultat einen Schlüssel enthält
- Minimiere die Verwendung von GROUP BY und HAVING:

```
SELECT MIN (E.age)
FROM Employee E
GROUP BY E.dno
HAVING E.dno=102
```

```
SELECT MIN (E.age)
FROM Employee E
WHERE E.dno=102
```

- Untersuche die Verwendung von Indexen durch das DBMS beim Einsatz arithmetischer Ausdrücke: *E.age=2*D.age* profitiert von einem Index auf *E.age*, aber wahrscheinlicht nicht von einem Index auf *D.age*!
  (mögliche Alternative: *E.age/2=D.age*)

Noch mehr Tips zum Tuning von Queries

- Vermeide Zwischenrelationen:

```
SELECT * INTO Temp
FROM Emp E, Dept D
WHERE E.dno=D.dno
      AND D.mgrname='Joe'
```

and

```
SELECT T.dno, AVG(T.sal)
FROM Temp T
GROUP BY T.dno
```

vs.

```
SELECT E.dno, AVG(E.sal)
FROM Emp E, Dept D
WHERE E.dno=D.dno
      AND D.mgrname='Joe'
GROUP BY E.dno
```

- Keine Materialisierung der Zwischenrelation Temp erforderlich
- Wenn es einen dichtbesetzten B+ Baum Index <*dno,sal*> gibt, kann ein Index-Only-Plan genutzt werden. Somit kein Zugriff auf die Tupel der Relation Emp (wie in der 2. Lösung) erforderlich!

Zusammenfassung

- Das konzeptuelle Schema sollte verfeinert werden unter Berücksichtigung von Performance und Lastprofil:
  - Wahl von 3NF oder niedriger Normalform gegenüber BCNF
  - Auswahl unter alternativen Dekompositionen in die BCNF (oder 3NF) basierend auf dem Lastprofil
  - Wir können *denormalisieren* oder einige Dekompositionen rückgängig machen
  - Wir können eine BCNF-Relation weiter dekomponieren!
  - Wir können eine *horizontale Dekomposition* für eine Relation wählen
  - Wie wichtig ist uns die Erhaltung einer bestimmten Abhängigkeit (Integritäts-Constraint)?
    - Bedeutung dieses Constraints für die geforderte Korrektheit der Daten
    - Kosten der Überprüfung des Integritäts-Constraints
      - Hinzufügen einer Relation, um die Bewahrung der Abhängigkeit zu sichern (bei einer 3NF Relation, nicht BCNF!)
      - Prüfung der Abhängigkeit durch einen Join

Zusammenfassung (Forts.)

- Nach einer bestimmten Zeit müssen Indexe fein getunt werden (wegwerfen, erzeugen, reorganisieren, ...) aus Performance-gründen
  - Ermittle den Ausführungsplan, der vom System genutzt wird, und korrigiere die Wahl der Indexe entsprechend
- Wenn das System trotzdem keinen guten Plan findet:
  - Optimierer des DBMS untersucht nur beschränkten Lösungsraum entsprechend vorgegebener Heuristiken (z.B. Left-Deep entsprechend System R-Ansatz)
  - Null-Werte, arithmetische Bedingungen, String-Ausdrücke, Verwendung von ORs, etc. kann den Optimierer verwirren
- Wir können selbst die Query/View umschreiben:
  - Vermeide geschachtelte Anfragen
  - Vermeide temporäre Relationen
  - Vermeide komplexe Bedingungen
  - Vermeide Operationen wie DISTINCT und GROUP BY