Physischer DB-Entwurf

Überblick

- Ausgangslage: *Konzeptuelles* und *externes* Schema sind erstellt: ER Modell, Schemaverfeinerung und Definition von Sichten
- Nächster Schritt: *Physischer Entwurf:* Auswahl von Indexen, Clustering, Verfeinerung von konzeptuellem und externen Schemas (wenn notwendig), um die gewünschte Performance zu erzielen:
- Dazu erforderlich Analyse des *Lastprofils*:
  - Die wichtigsten Anfragen und deren Häufigkeit
  - Die wichtigsten Updates und deren Häufigkeit
  - Die benötigte Performance für diese Anfragen und Updates
- Für jede Anfrage im Lastprofil:
  - Auf welche Relationen wird zugegriffen?
  - Welche Attribute werden zurückgegeben?
  - Welche Attribute sind involviert in Selektion/Join Bedingungen? Wie selektiv werden diese Bedingungen wohl sein?
- Für jedes Update im Lastprofil:
  - Welche Attribute sind beteiligt in Selektion/Join Bedingung? Wie selektiv sind diese Bedingungen wahrscheinlich?
  - Typ des Updates (INSERT/DELETE/UPDATE), und die betroffenen Attribute

Zu treffende Entscheidungen

- Welche Indexe sollten angelegt werden?
  - Welche Relationen sollten Indexe haben?
  - Welche Attribute sollten Suchschlüssel im Index sein?
  - Sollten mehrere Indexe auf einer Relation angelegt werden?
- Für jeden Index: Welche Art von Index sollte es sein?
  - Geclustert oder nicht?
  - Hash- oder Baumbasierter Index
  - Dynamischer oder statischer Index
  - Dicht- oder dünnbesetzter Index
- Sind Änderungen im konzeptuellen Schema erforderlich?
  - Untersuche alternative normalisierte Schemas (es gibt mehrere Möglichkeiten einer Dekomposition in BCNF etc.)
  - Sollten einige Dekompositionsschritte rückgängig gemacht werden, d.h. Umwandlung in eine niedrigere Normalform (*Denormalisierung)*
  - Horizontale Partitionierung
  - Replikation
  - Views

Auswahl von Indexen

- Ansatz:
  - Untersuche die wichtigsten Queries
  - Untersuche den besten Ausführungsplan mit den vorhandenen Indexen
  - Falls ein besserer Plan möglich ist mit zusätzlichem Index, erzeuge diesen
- Vor dem Erzeugen eines Index muß auch der Einfluß von Updates im Lastprofil berücksichtigt werden
  - Trade-Off (Abwägung):
    - Index beschleunigt Abfragen
    - Index verlangsamt Updates
    - Index braucht zusätzlichen Plattenplatz

Regeln für Indexauswahl

1. Index benötigt?
   Indexe so auswählen, daß sie möglichst viele Queries unterstützen (wenn kein Bedarf auf Index verzichten)
2. Wahl des Suchschlüssels
   Attribute in einer WHERE-Klausel sind Kandidaten für Suchschlüssel eines Index
   - Bedingungen mit exakter Wertübereinstimmung (exact match) verlangen Hash-Index
   - Wertbereichsbedingungen (range query) verlangen Baum-Index
     - Clustering sinnvoll für Wertbereichs-Anfragen, auch nützlich bei Gleichheits-Anfragen, wenn Duplikate vorhanden sind
3. Clustering
   Nur ein Index kann pro Relation geclustert werden
   - Auswahl entsprechend der wichtigen Queries, die am meisten von diesem Index profitieren können (Kriterium: Häufigkeit, Selektivität)
   - Range Queries profitieren am meisten vom Clustering

Auswahl eines Index (Forts.)

4. Mehr-Attribut-Suchschlüssel
   - Suchschlüssel aus mehreren Attributen sollten untersucht werden, wenn eine WHERE-Klausel mehrere Bedingungen enthält
   - Erlauben Index-Only Ausführungspläne (ohne Zugriff auf die eigentliche Relation) für wichtige Queries
   - Wenn Range-Queries erwartet werden: Reihenfolge der Attribute im Suchschlüssel beachten, damit diese den Queries entspricht
5. Unterstützung von Join-Bedingungen
   - B+ Baum i. allg. gut geeignet weil Unterstützung für Lookups und Range Queries
     - Geclusterter B+ Baum auf Join Spalten gut für Sort Merge Join
   - Hash-Index in bestimmten Situationen gut:
     - Unterstützt Index Nested Loop Joins, Suchschlüssel enthält die Join-Spalten, Index auf der inneren Relation

Beispiel 1

```
SELECT  E.ename, D.mgr
FROM  Emp E, Dept D
WHERE  D.dname='Toy' AND E.dno=D.dno
```

- Hash-Index auf *D.dname* unterstützt 'Toy' Selektion
  - Damit wird Index auf D.dno nicht benötigt (weil die Tupel von Departments über den *dname*-Index gelesen werden)
- Hash-Index auf *E.dno* erlaubt es, die matchenden Tupel der inneren Relation Emp für jedes selektierte Tupel der äußeren Relation Dept zu lesen
- Was wäre wenn WHERE enthalten würde: " ... AND E.age=25" ?
  - Könnten Emp-Tupel lesen mit einem Index auf *E.age*, dann Join mit den Dept-Tupeln, die die *dname* Selektionsbedingung erfüllen. Vergleichbar mit der Strategie, die den *E.dno* Index verwendet
  - Wenn ein *E.age* Index bereits kreiert wurde, gibt es bei dieser Query viel weniger Motivation einen zusätzlichen Index für *E.dno* anzulegen

Beispiel 2

```
SELECT  E.ename, D.mgr
FROM  Emp E, Dept D
WHERE  E.sal BETWEEN 10000 AND 20000
  AND E.hobby='Stamps' AND E.dno=D.dno
```

- Emp muß die äußere Relation sein
  - Damit ergibt sich ein Hash-Index auf *D.dno.*
- Welcher Index sollte auf Emp kreiert werden?
  - B+ Baum auf *E.sal* ODER ein Index auf *E.hobby* könnte genutzt werden. Nur einer davon benötigt. Die Auswahl hängt von der Selektivität der Bedingungen ab
    - Daumenregel: Gleichheitsanfrage ist selektiver als Wertbereichsanfrage.
- Beide Beispiele zeigen:
  Auswahl der Indexe wird bestimmt durch die Ausführungspläne, die von einem Optimierer für eine Query angelegt werden
  Verständnis des Optimierers erforderlich!

Beispiele für Clustering

- B+ Baum Index auf E.age kann genutzt werden, um die qualifizierten Tupel zu erhalten
  - Wie selektiv ist die Bedingung?
  - Ist der Index geclustert?
- Untersuche die GROUP BY Klausel
  - Mit vielen Tupeln, die die Bedingung *E.age* > 10 erfüllen, kann die Verwendung eines *E.age* Index und die Sortierung der erhaltenen Tupel teuer werden
  - Geclusterter *E.dno* Index kann besser sein!
- Lookup-Anfragen und Duplikate:
  - Clustering auf *E.hobby* hilft!

```
SELECT  E.dno
FROM  Emp E
WHERE  E.age>40
```

```
SELECT  E.dno, COUNT (*)
FROM  Emp E
WHERE  E.age>10
GROUP BY E.dno
```

```
SELECT  E.dno
FROM  Emp E
WHERE  E.hobby='Stamps'
```

Clustering und Joins

```
SELECT  E.ename, D.mgr
FROM  Emp E, Dept D
WHERE  D.dname='Toy' AND E.dno=D.dno
```

- Clustering ist vor allem wichtig, wenn innere Tupel zugegriffen werden bei einem Index Nested Loop Join
  - Innere Relation Emp, Emp.dno ist kein Schlüsselkandidat
  - Index auf *E.dno* sollte geclustert sein (da Lookup auf dno wiederholt aufgerufen wird)
- Annahme, die WHERE-Klausel wäre wie folgt:

  WHERE E.hobby='Stamps' AND E.dno=D.dno
  - Sort Merge Join sinnvoll, falls viele Angestellte Briefmarken sammeln, mit geclustertem Index auf *D.dno* (vermeidet das Sortieren der Abteilungen)
  - Ungeclusterter Index nicht gut, weil alle Tupel gelesen werden (mit 1 I/O pro Tupel)
- *Fazit*: Cluster sind immer dann sinnvoll, wenn viele Tupel zurückgegeben werden (bei Lesen einzelner Tupel kein Unterschied)

Mehr-Attribut-Indexschlüssel

- Um Emp-Tupel mit der Bedingung *age*=30 AND *sal*=4000 zu lesen, wäre ein Index auf <*age,sal*> besser als ein Index auf *age* oder ein Index auf *sal*.
  - Solche Indexe heißen auch *komposite* oder *zusammengesetzte* Indexe
  - Wahl des Indexschlüssel orthogonal zur Frage Clustering, dichter oder dünner Index etc.
- Wenn die Bedingung: 20<*age*<30 AND 3000<*sal*<5000:
  - Geclusterter Baum-Index auf <*age,sal*> oder <*sal,age*> ist am besten
- Wenn die Bedingung: *age*=30 AND 3000<*sal*<5000:
  - Geclusterter <*age,sal*> Index viel besser als <*sal,age*> Index!
- Komposite Indexe sind größer und werden öfter geändert

Index-Only-Pläne

- Eine Reihe von Queries kann bearbeitet werden, ohne Zugriff auf die eigentlichen Tupel aus einer oder mehreren Relationen, wenn ein passender Index verfügbar ist.

Ungeclustert
<E.dno>

```
SELECT D.mgr
FROM Dept D, Emp E
WHERE D.dno=E.dno
```

<E.dno,E.eid>
Baum-Index!

```
SELECT D.mgr, E.eid
FROM Dept D, Emp E
WHERE D.dno=E.dno
```

<E.dno>

```
SELECT E.dno, COUNT(*)
FROM Emp E
GROUP BY E.dno
```

<E.dno,E.sal>
Baum-Index!

```
SELECT E.dno, MIN(E.sal)
FROM Emp E
GROUP BY E.dno
```

<E. age,E.sal>
oder
<E.sal, E.age>
Baum!

```
SELECT AVG(E.sal)
FROM Emp E
WHERE E.age=25 AND
  E.sal BETWEEN 3000 AND 5000
```

Zusammenfassung

- Datenbank-Entwurf besteht aus verschiedenen Phasen:
  - Anforderungsanalyse
  - Konzeptueller Entwurf
  - Schemaverfeinerung
  - Physischer Entwurf
  - Tuning
  - Im allgemeinen sind mehrere Iterationen zwischen diesen Phasen notwendig, um einen Datenbank-Entwurf zu verfeinern. Entwurfsentscheidungen in einer Phase haben einen Einfluß auf andere Phasen

- Das Verständnis des *Lastprofils* der Anwendung und der Performance-Ziele ist wesentlich für die Entwicklung eines guten Entwurfs:
  - Was sind die wichtigsten Anfragen und Updates?
  - Welche Relationen und Attribute sind beteiligt?

Zusammenfassung (Forts.)

- Indexe müssen eingerichtet werden, um wichtige Anfragen (und auch manche Updates) zu beschleunigen
  - Pflege des Index schafft Overhead bei Updates auf Schlüsselfeldern
  - Wähle Indexe, die viele Anfragen unterstützen wenn möglich
  - Baue Indexe, um Index-Only-Strategien zu unterstützen
  - Clustering ist wichtige Entscheidung: nur ein Index kann auf einer Relation geclustert werden
  - Ordnung der Felder in einem zusammengesetzten Index kann wichtig sein
- Statische Indexe müssen periodisch reorganisiert werden (z.B. bei zu vielen Überlaufseiten)
- Statistiken müssen periodisch aktualisiert werden