Anfrageverarbeitung

Einführung

- Wir betrachten die Implementierung der Operatoren der Relationenalgebra:
- Basis-Operationen:
  - *Selektion* (σ) Auswahl einer Teilmenge von Tupeln aus der Relation
  - *Projektion* (π) Auswahl von Spalten aus einer Relation
  - *Join* ($\bowtie$) Kombination zweier Relationen
  - *Mengendifferenz* (-) Menge der Tupel aus Relation 1, die nicht in Relation 2 sind
  - *Aggregation* (SUM, MIN, etc.) und GROUP BY
- Jede Operation liefert eine Relation, somit sind die Operationen kombinierbar!
- Zunächst Betrachtung der Operationen, anschließend Optimierung der Anfragen, die durch Komposition gebildet werden

Unäre Operationen

- Relationen-Scan: durchläuft alle Tupel einer Relation in beliebiger Reihenfolge (Reihenfolge wird intern durch die Verteilung der Tupel auf Blöcke und evtl. durch Puffer-Strategie bestimmt)
  - Kann die Blockung der Tupel ausnutzen
  - Alle Tupel müssen gelesen werden
- Index-Scan: nutzt einen Index zum Auslesen der Tupel in Sortierreihenfolge (kein Hash-Index!)
  - Bereichs-Anfragen und Lookups (Exact Match) gut unterstützt
  - Unabhängig von der Blockung (u.U. höherer Aufwand als beim Relationen-Scan)

Binäre Operationen

- Basieren auf einem tupelweisen Vergleich der Tupelmengen, um Join-Bedingungen zu prüfen oder Duplikate zu eliminieren (Ausnahmen Kreuzprodukt und Vereinigung ohne Duplikateliminierung)
- 3 Basis-Techniken:
  - Nested Loops-Technik (Schleifeniteration)

    Für jedes Tupel einer äußeren Relation wird eine innere komplett durchlaufen
  - Merge-Technik (Mischmethode)

    Beide Relationen werden schrittweise in der vorgegebenen Tupelreihenfolge durchlaufen. Als Vorbereitung müssen die beiden Relationen nach gleichen Sortierkriterium sortiert vorliegen. Nur für Tupelvergleiche ‘=‘ geeignet
  - Hash-Methoden

    Die kleinere der beiden Relationen wird in einer Hash-Tabelle gespeichert. Die Tupel der zweiten Relation finden ihren Vergleichspartner mittels Hash-Funktion. Nur Tupelvergleich ‘=‘ unterstützt

Beispiel-Schema

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- Relationen über Segler und die Reserviert-Beziehung zwischen Seglern und Booten (*rname* hinzugefügt, Name der Reservierung)
- Reserves:
  - Jedes Tupel ist 40 Bytes lang, 100 Tupel pro Seite, 1000 Seiten
- Sailors:
  - Jedes Tupel ist 50 Bytes lang, 80 Tupel pro Seite, 500 Seiten

Equi-Join mit einer Join-Spalte

```
SELECT  *
FROM    Reserves R1, Sailors S1
WHERE   R1.sid=S1.sid
```

- In Algebra: R ⋈ S. Wichtig! Muß sorgfältig optimiert werden. R × S ist teuer; deshalb wäre R × S mit anschließender Selektion ineffizient.
- Annahme: M Seiten mit Tupeln in R, p_R Tupel pro Seite, N Seiten mit Tupeln in S, p_S Tupel pro Seite.
  - In unseren Beispielen gilt: R = Reserves and S = Sailors.
- Komplexere Join-Bedingungen später
- *Kostenmetrik*: Anzahl von I/Os (Transfer von Seiten). Wir ignorieren Output-Kosten (d.h. Ausgabe des Ergebnisses)

Einfacher Nested Loop Join

```
foreach tuple r in R do
        foreach tuple s in S do
                if r_i == s_j then add <r, s> to result
```

- Für jedes Tupel in der äußeren Relation R, scannen wir die gesamte innere Relation S:
  - Kosten: $M + p_R * M * N = 1000 + 100*1000*500$ I/Os.
- Seitenorientierter Nested Loops Join: Für jede Seite von R, lese jede Seite von S, und schreibe Paare von Tupeln <r, s>, die die Join-Bedingung erfüllen, in die Ergebnisrelation, wobei gilt: r ist auf Seite R und s auf Seite S.
  - Kosten: $M + M*N = 1000 + 1000*500$
  - Wenn die kleinere Relation (S) die äußere ist:
    Kosten = 500 + 500*1000

Nested Loop Join mit Index

foreach tuple r in R do
    foreach tuple s in S where r_i == s_j do
        add <r, s> to result

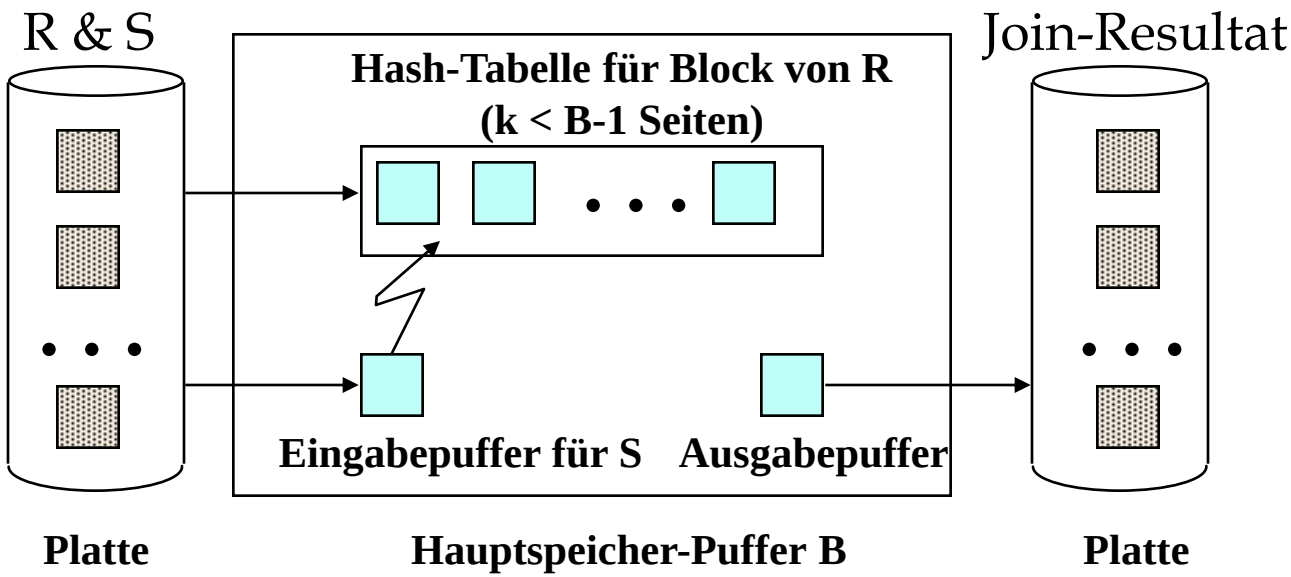
- Wenn es einen Index auf der Join-Spalte einer Relation gibt (z.B. S), kann diese zur inneren Relation gemacht und der Index ausgenutzt werden.
  - Kosten: M + ((M*p_R) * Kosten, um matchende Tupel zu finden)
- Für jedes Tupel aus R sind die Kosten der Bedingungsprüfung im Index von S ca. 1.2 für Hash-Index, 2-4 für B+ Baum. Kosten, um die Datentupel zu finden, hängt von der Cluster-Variante ab (Art der Dateneinträge im Index kann variieren).
  - Geclusterter Index: 1 I/O (typisch), ungeclustert: bis zu 1 I/O pro matchendem S-Tupel

Beispiel für Nested Loop Joins mit Index

- Hash-Index auf *sid* von Sailors (als innere Relation):
  - Scannen von Reserves: 1000 Page I/Os, 100*1000 Tupel.
  - Für jedes Tupel aus Reserves: 1.2 I/Os, um Dateneintrag im Index von Sailors zu finden, plus 1 I/O, um das matchende Tupel zu finden (genau 1, da *sid* = Schlüssel)
    Gesamt: 220,000 I/Os
- Hash-Index auf *sid* von Reserves (als innere Relation):
  - Scannen von Sailors: 500 Page I/Os, 80*500 Tupel
  - Für jedes Tupel aus Sailors: 1.2 I/Os, um Indexseite mit den Dateneinträgen zu finden, plus Retrieval-Kosten für die matchenden Tupel aus Reserves.
    - Annahme: gleichmäßige Verteilung, 2.5 Reservierungen pro Segler (100,000 / 40,000)
    - Retrieval-Kosten betragen somit 1 (Index geclustert, 1 Seitenzugriff) oder 2.5 I/Os (Index ungeclustert, Seitenzugriff pro matchendem Tupel)

Block Nested Loop Join

- Nehme eine Seite als Eingabe-Puffer zum Scannen der inneren Relation S, eine Seite als Ausgabe-Puffer, und nutze alle restlichen Seiten, um einen Block der äußeren Relation R in den Hauptspeicher aufzunehmen (äußere ist kleinere Relation)
- Falls Speicher nicht reicht: Mehrere Blöcke für R
- Zusätzliche Optimierung: Hash-Tabelle für Block von R
  - Für jedes matchende Tupel r im R-Block und s auf S-Seite, füge <r, s> zum Ergebnis hinzu. Dann S weiterscannen, lese nächsten R-Block, scanne S, usw.



Beispiel: Block Nested Loop Join

- Kosten: Scan der äußeren Relation + Anzahl Blöcke äußere Relation * Scan der inneren Relation
  - # Blöcke äußere Relation = $\lceil \#Seiten \quad äußere\ Relation\ /\ Blockgröße \rceil$
- Mit Reserves (R) als äußere Relation, und Blöcke mit 100 Seiten von R:
  - Kosten des Scannens von R =1000 I/Os; insgesamt 10 Blöcke
  - Pro Block von R scannen wir Sailors: 10*500 I/Os
  - Gesamtkosten: 1000 + 10*500 = 6000 I/Os
  - Wenn Platz ist für 90 Seiten von R, würden wir S 12mal scannen (somit Gesamtkosten: 1000 + 12*500 = 7000 I/Os)
- Mit Sailors (S) als äußere Relation, Blöcke mit 100 Seiten:
  - Kosten des Scannens von S = 500 I/Os; insgesamt 5 Blöcke
  - Pro Block von S scannen wir Reserves: 5*1000 I/Os
  - Gesamtkosten: 500 + 5*1000 = 5500 I/Os
- Besser als nur eine Puffer-Seite für innere Relation ist gleichmäßige Aufteilung des Puffers zwischen R und S

Sort Merge Join $R \bowtie_{i=j} S$

- Ablauf
  - Sortiere R and S nach der Join-Spalte (bzw. den Join-Spalten): Tupel, die in der Join-Spalte übereinstimmen, werden somit gruppiert
  - Mische die vorsortierten Relationen auf der Join-Spalte
  - Ausgabe der Ergebnistupel
  - Scanne R bis das aktuelle R-Tupel >= aktuelles S-Tupel, dann scanne S, bis das aktuelle S-Tupel >= aktuelles R-Tupel; tue das solange bis aktuelles R-Tupel = aktuelles S-Tupel.
  - An diesem Punkt stimmen alle R-Tupel mit demselben Wert in Ri (*aktuelle R-Gruppe*) und alle S-Tupel mit demselben Wert Sj überein; Ausgabe <r, s> aller solchen Paare von Tupeln aus R und S.
  - Dann fortsetzen mit dem Scannen von R und S.
- R wird einmal gescannt; jede S-Gruppe wird einmal pro matchendem R-Tupel gescannt (Mehrfache Scans einer S-Gruppe finden wahrscheinlich die benötigte Seiten im Puffer)

Beispiel für Sort Merge Join

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

- Kosten: M log M + N log N + (M+N)
  - Sortieren M + Sortieren N + Scannen M und N (falls keine S-Gruppe mehrfach gescannt werden muß)
  - Die Kosten des Scannens, M+N, könnten M*N werden (sehr unwahrscheinlich!)
- Mit 35, 100 oder 300 Pufferseiten, können Reserves und Sailors in 2 Pässen sortiert werden;
- Gesamte Join-Kosten:
- Sort Reserves (2*2*1000=4000) + Sort Sailors (2*2*500=2000) + Scan Reserves(1000) + Scan Sailors(500) = 7500

(*Block Nested Loop Kosten: 2500 bis 15000 I/Os*)

Einfache Selektion

```
SELECT  *
FROM    Reserves R
WHERE   R.rname < 'C%'
```

Form $\sigma_{R.attr\,op\,value}(R)$

- Größe des Resultats angenähert als *size of R * Reduktionsfaktor*; Abschätzung des Reduktionsfaktors nötig
- Ohne Index, unsortiert: Erfordert im wesentlichen einen Scan auf der ganzen Relation; Kosten = M (#Seiten in R)
- Mit einem Index auf dem Selektionsattribut: Verwende Index, um Tupel zu finden, die die Bedingung zu erfüllen, gebe die korrespondierenden Dateneinträge aus (Hash-Index sinnvoll nur für Lookups)

Selektion mit Index

- Kosten sind abhängig von der Anzahl der gefundenen Tupel und dem Clustering
  - Kosten, um Dateneinträge zu finden, die die Selektionsbedingung erfüllen (typischerweise klein) + Kosten für das Retrieval der eigentlichen Datensätze (können sehr groß sein ohne Cluster)
  - In unserem Beispiel sei eine gleichmäßige Verteilung von Namen angenommen, 10% der Tupel, die die Bedingung erfüllen (=100 Seiten, 10000 Tupel). Mit einem geclusterten Index betragen die Kosten etwas mehr als 100 I/Os; wenn ungeclustert: bis zu 10000 I/Os!
- *Bedeutende Verfeinerung für ungeclusterte Indexe:*
  1. Finde qualifizierende Dateneinträge im Index
  2. Sortiere die IDs der Datensätze, die gelesen werden (somit auch Sortierung nach Seiten-ID als Bestandteil der Satz-ID)
  3. Lese die Satz-IDs in der Reihenfolge. Somit gewährleistet, daß jede Datenseite genau einmal eingelesen (Retrieval-Kosten = Anzahl der Seiten, die qualifizierende Tupel enthalten)

Allgemeine Selektionsbedingung*

(day<8/9/94 AND rname='Paul') OR bid=5 OR sid=3

- Solche Selektionsbedingungen zunächst konvertiert in die *Konjunktive Normalform* (KNF) = Konjunktion von Disjunktionen
  (day<8/9/94 OR bid=5 OR sid=3) AND
  (rname='Paul' OR bid=5 OR sid=3)
- Wir betrachten nur den Fall ohne ORs (eine Konjunktion von *Termen* der Form *attr* ***op*** *value*):
- Ein Index *matcht* (eine Konjunktion von) Termen, die nur Attribute in einem Präfix des Suchschlüssels beinhalten
  - Index auf *<a, b, c>* matcht *a=5 AND b= 3*, aber nicht *b=3*.

Allgemeine Selektion (Forts.)*

- Heuristische Auswahl eines Terms, das sich besonders gut über den Index auswerten läßt (d.h. ermittle den *Zugriffspfad höchster Selektivität*)
  - *Zugriffspfad höchster Selektivität:* Ein Index oder File-Scan, der schätzungsweise die wenigsten Page I/Os erfordert.
- Einlesen der Tupel über den gewählten Index
  - Terme, die den Index matchen, verringern die Anzahl Tupel, die eingelesen werden
- Anwendung der verbleibenden Terme, die nicht dem Index entsprechen (Verschärfung der Selektionsbedingung)
  - Entfernen von Tupeln, die im vorigen Schritt gefunden werden
  - Beispiel: *day<8/9/94 AND bid=5 AND sid=3.* Nutze einen B+ Baum auf *day*; dann prüfe die (Teil-)Bedingung *bid=5* and *sid=3* für jedes erhaltene Tupel.
    Oder andersherum: Hash-Index auf <*bid, sid*>; prüfe anschließend *day<8/9/94*
- Optimierung: Operiere mit Listen von Tupel-IDs (anstelle von Tupeln), Auswertung mehrerer Terme und Schneiden dieser Listen

Projektions-Operation

```
SELECT   DISTINCT
         R.sid, R.bid
FROM     Reserves R
```

- Aufwand abhängig, ob Duplikate-Eliminierung durchgeführt werden muß oder nicht
- Standard-Lösung beruht auf Sortierung
  - Sortierte Ausgabe eines Index hilft bei Duplikateliminierung (da Duplikate benachbart)
  - Projektion auf indexierte Attribute ohne Zugriff auf die gespeicherten Tupel möglich (da alle Attribute aus Index gelesen werden können)
  - Wenn Index alle benötigten Attribute als Präfix des Suchschlüssels enthält noch besser:
    - Lesen der Einträge aus dem Index, Ausblenden ungewünschter Spalten, Vergleich benachbarter Tupel ob Duplikate?
- Auch andere Verfahren zur Duplikaterkennung möglich, etwa Hash-Verfahren

Mengen-Operationen

- Durchschnitt und Kreuzprodukt sind Spezialfälle des Join
- Vereinigung (DISTINCT) und Differenz ähnlich
- Ansätze für UNION (als Beispiel)
  - Sortierverfahren
    - Sortiere beide Relationen (auf der Kombination aller Attribute)
    - Scannen der sortierten Relationen und mischen, so daß Sortierreihenfolge erhalten bleibt
    - *Alternative*: Mische beide Relationen bereits im 1. Paß
  - Hash-basierter Ansatz:
    - Partitioniere R und S mit einer Hash-Funktion *h*
    - Für jede S-Partition *l*, baue eine Hash-Tabelle im Speicher (Hash-Funktion *h2*), scanne die korrespondierende R-Partition und füge Tupel zur Tabelle hinzu unter Beseitigung von Duplikaten

Aggregations-Operatoren (AVG, MIN, etc.)

- Ohne Gruppierung
  - Im allgemeinen Scannen der gesamten Relation erforderlich
  - Falls Index, dessen Suchschlüssel alle Attribute der SELECT- und WHERE-Klausel enthält: Index-Scan genügt
- Mit Gruppierung:
  - Sortieren nach den GROUP BY-Attributen, dann Scannen der Relation und Berechnung der Aggregate für jede Gruppe
  - Ähnlicher Ansatz mit Hashing der GROUP BY-Attribute
  - Optimierungsmöglichkeiten, falls Suchschlüssel des Index alle Attribute von SELECT, WHERE und GROUP BY-Klausel enthält

Zusammenfassung

- Vorzug von relationalen DBMS:
  - Queries sind zusammengesetzt aus wenigen, primitiven Operatoren
  - Diese Operatoren können sorgfältig optimiert und getunt werden
- Es gibt viele verschiedene Implementierungstechniken für jeden Operator
  - Keine universell gültige Technik für die meisten Operatoren
- Optimierung der Anfragebearbeitung ist Teil der Query-Optimierung
  - Auswahl der günstigsten Alternative für jede Operation
  - Zugriffsstatistik auswerten (z.B. Zugriffsmuster, Selektivität von Anfragen)