Ausführung von Anfragen

Status Quo:

Wir wissen ganz genau wie man Entscheidungen trifft.

Aber:

Wir wissen noch nicht so genau, wie diese Entscheidungen ausgeführt werden.

Übersicht: Wir bauen eine Verteilte Query Engine

1. Das Iteratormodell
2. Standardoperatoren im Iteratormodell
3. Netzwerkoperatoren im Iteratormodell
4. Semijoinverfahren für verteilte Joins
5. Komprimierung und anderes Gedöns

Literatur: Götz Graefe, Computing Surveys, 1993

Das Iteratormodell

- Jeder Operator wird als "Iterator" implementiert.
  Ein Plan wird also in einen Baum von Iteratoren übersetzt.

- Alle Iteratoren haben dasselbe Interface:

  open ruft "open" der darunterliegenden Operatoren auf, legt Puffer und temporäre Files an, initialisiert interne Strukturen, macht alle Vorarbeiten

  next produziert Ergebnistupel und liefert Zeiger auf Ergebnistupel als Ergebnis

  close ruft (ggf.) "close" der darunterliegenden Operatoren auf, gibt Puffer frei und zerstört temporäre Files.

- Vorteile des Iteratormodells:

  - Modularität: Es kann jederzeit ein neuer Iterator hinzugenommen werden; Iteratoren brauchen Eigenheiten anderer Iteratoren oder ganzer Pläne nicht zu kennen.
  - Materialisieren von Zwischenergebnissen und Pipelining können realisiert werden.
  - Es kann sogar "intra-operator" Parallelität implementiert werden.
    (Dazu kommen wir noch in ein paar Wochen.)

Beispieliteratoren: *TBScan*

open Vorbereitungen

1. Öffene Segment auf Platte zum Lesen
2. Allokiere M Seiten Puffer im Hauptspeicher
3. Lese ersten Block von M Seiten des Segmentes in den Puffer
4. Initialisiere Zähler auf: 1. Tupel; 1. Seite

next Produziere Tupel

1. Lese Tupel vom Puffer, auf das der Zähler zeigt.
2. Erhöhe Zähler; evtl. lese nächsten Block von M Seiten.
3. (falls Prädikate anwendbar: lese + erhöhe Zähler, bis ein Tupel die Prädikate erfüllt)
4. Gebe gelesenes (bzw. winner) Tupel als Ergebnis zurück.

close Aufräumen

1. Schließe Segment auf Platte
2. Gebe M Seiten Puffer frei

Frage: Wieso allokiert man hier $M \geq 1$ Seiten Puffer?

Beispieliteratoren: *Materialize*

open hier spielt sich alles ab:

1. **open** auf Produzent-Iterator.
2. Lege temporäres Segment auf Platte an.
3. Allokiere M Seiten Puffer.
4. wiederholt **next** auf Produzent-Iterator.
   Schreibe Tupel in Blöcken von M Seiten auf Platte.
5. **close** auf Produzent-Iterator.
6. Gebe M Seiten Puffer wieder frei.

next illegal!!!
(Es geht kann nur mit einem *TBScan* Iterator weitergehen.)

close Zerstöre das temporäre Segment auf Platte.

1. Frage: Was passiert, wenn alle Tupel in die M Puffer passen?

2. Frage: Wieso ruft man zuerst **open** auf Produzent-Iterator, bevor man selber Ressourcen allokiert?

Beispieliteratoren: *Sort-Merge Join*

open Verwurste komplette Eingabe der beiden Produzenten zu Läufen

next Merge Läufe, um matchende Tupel zu produzieren.

close Zerstöre Läufe und gebe Puffer frei.

Kommunikationsiteratoren (verteilt)

Zwei Sachen, die man vermeiden möchte:

1. Tupel sollten nicht einzeln sondern in Blöcken verschickt werden.
   (Reduziert #Nachrichten und damit Kommunikationskosten)
2. Sender sollte Empfänger nicht überfluten, falls Empfänger mal stockt.

Lösung: Blockweises verschicken von Tupeln mit Anforderung

- Etabliere *Send* und *Receive* Iteratoren auf den jeweiligen Sites.
- Gebe beiden einen möglichst großen Puffer auf ihren jeweiligen Sites.
  (Aber Puffer gleich groß auf Sender wie auf Empfängerseite.)

- *Send* füllt seinen Puffer mit Tupeln; auf Anfrage von *Receive* verschickt *Send* Inhalt des gesamte Puffers (Block) und fängt nach erfolgreicher Übertragung erneut an, seinen Puffer zu füllen; *Receive* kriegt Block von Tupeln und übergibt Tupel für Tupel gemäß Iteratormodell seinem Verbraucheriterator; falls *Receive* alle Tupel des Blockes abgearbeitet hat, schickt *Receive* eine Nachricht an *Send*, um den nächsten Block anzufordern.

Zusätzliche Vorteile:

1. Empfänger Site und Sender Site führen ihre Operationen wirklich parallel aus.
   (Wäre bei tupelweise verschicken auf Anforderung nicht der Fall.)
2. Empfänger Site hat noch kleinen Vorrat, mit dem sie weiterarbeiten kann, selbst wenn Sender Site stockt.

Grundsätzlich:

In Datenbanken ist sehr häufig eine blockweise Verarbeitung gut, und das verträgt sich wunderbar mit dem Iteratormodell.

(Tupelschnittstelle ist GGT von allen nur vorstellbaren Iteratoren, und Iteratoren können blockweise Verarbeitung sehr schön dahinter verbergen.)

Implementierung von Multicasts

Betrachte den folgenden Plan:

$$(A \bowtie B_1) \cup (A \bowtie B_2) \cup (A \bowtie B_3)$$

A liegt in New York; B_1 liegt in Grubweg; B_2 liegt in Hals; B_3 liegt in der Innstadt. A soll nach Grubweg, Hals und in die Innstadt geschick werden, damit die Joins dort ausgeführt werden.

Wie macht man das?

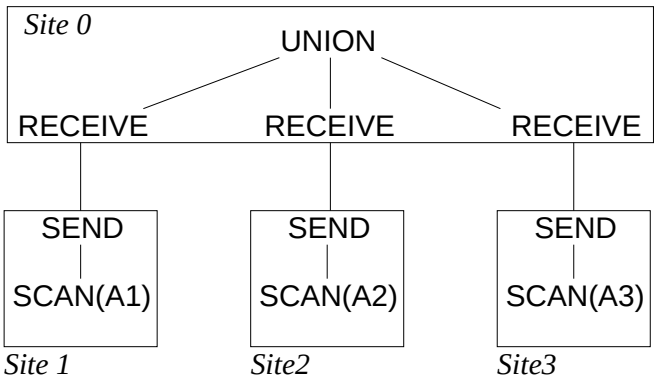
- New York schickt A nach Grubweg (mittels *Send* und *Receive* Operatoren)
- Grubweb verteilt A weiter an Hals und Innstadt
- Alternative: Hals, Innstadt oder gar Straubing (unbeteiligt) dienen als Verteiler
  Optimierer muß entscheiden!
- Vorsicht Plan ist kein reiner Baum mehr!

Vorteile

- niedrigere Kommunikationskosten in hierarchischen Netzwerken
- ggf. bessere Lastbalancierung (kann also auch in LANs sinnvoll sein)

Multithreading und Scheduling

Wie hoch ist die Antwortzeit des folgenden Planes im klassischen Iteratormodell?

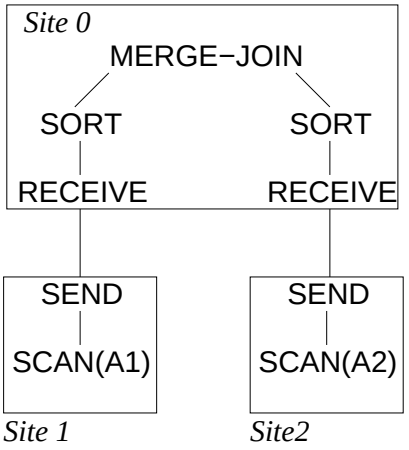


Beobachtung:

- reines Iteratormodell forciert tupelweise, sequentielle Ausführung → keine unabhängige Parallelität sondern nur Parallelität durch Pipelining
- Lösung: Multithreading
- Tradeoff: höhere Parallelität vs. Overhead (Semaphore für Shared Memory Kommunikation)
- (Übrigens Netscape macht es auch)

Multithreading und Scheduling (Teil 2)

Sollte in diesem Fall Multithreading eingesetzt werden?



Fazit:

- Optimierer muß wiederum entscheiden!
- Frage: wie macht der Optimierer das? (Überlegt es Euch!)

Semijoinverfahren

Das Problem:

Berechne $R_1 \bowtie R_2$ mit R_1 liegt auf Site I und R_2 liegt auf Site II. (Sei $R_1.a = R_2.b$ das Joinprädikat.)

Lösung 1: Verwende Standardjointechniken

1. Schicke R_1 von Site I nach Site II.
2. Führe den Join auf Site II als NLJ, index NLJ, SMJ oder HHJ aus.
3. (umgekehrt geht es natürlich auch; d.h. R_2 nach Site I schicken.)

Lösung 2: Semijoinprogramme

1. Schicke $R_1.a$ von Site I nach Site II.
2. Berechne $R_2' = R_2 \ltimes R_1$ auf Site II, und schicke das Ergebnis nach Site I.
3. Berechne das vollständige Ergebnis $R_1 \bowtie R_2'$ auf Site I. (mit NLJ, SMJ oder HHJ.)
4. (umgekehrt geht es natürlich auch)

Tradeoffs zwischen den beiden Lösungen

Kosten für Lösung 1:

Kommunikation: Schickt R_1 komplett übers Netz.

Lokale Berechnung: Join (wie zentral) an Site II.

Kosten für Lösung 2:

Kommunikation: Schickt nur eine Spalte (und ohne Duplikate) von R_1 übers Netz. Schickt nur die Tupel von R_2, die wirklich gebraucht werden übers Netz.

Lokale Berechnung: Projektion mit Duplikatelimination auf Site I. Semijoin auf Site II. Voller Join (auf hoffentlich kleinem R_2) auf Site I.

Fazit: In der Regel haben Semijoinprogramme niedrigere Kommunikationskosten aber dafür höhere lokale Berechnungskosten (CPU, Disk).

Wie immer muß Kostenmodell/Optimierer (oder so) entscheiden, was für eine bestimmte Anfrage besser ist.

Frage: In Lösung 2 sollte die größere oder die kleinere Relation außen sein? (Übung!)

Bloom Hash-Filter

Semijoinprogramm mit folgendem Twist:

1. Anstatt $R_1.a$ schicke einen Bitvektor.
2. Bitvektor wird wie folgt bestimmt:
   setze Bitvektor$[i]$, falls $\exists v \in R_1.a : h(v) = i$
3. Bilde R_2' aus Tupeln, für die gilt
   Bitvektor$[h(R_2.b)]$ ist gesetzt
4. Berechne wie gehabt Ergebnis als $R_1 \bowtie R_2'$ auf Site I.

Hinweise:

1. Wie immer beim Hashen geht dies nur bei Equijoins;
   d.h. nur bei Joinprädikaten mit einem "=" Zeichen.
2. Größe des Bitvektors ist ein Tuningparameter (s.u.)
3. Den Bitvektor nennt man auch Bloom Hash-Filter
   Literatur: Bloom, CACM 1971

Vorteile von Bloom Hash-Filtern:

- Je nach Größe des Bitvektors, müssen nur sehr geringe Datenmengen von Site I nach Site II geschickt werden.
- Geringer Aufwand, um den Bitvektor zu erstellen (auf Site I) und den Bitvektor auf R_2 (auf Site II) anzuwenden. (Duplikatelimination und Semijoin sind sehr viel teurere Operationen.)

Nachteile von Bloom Hash-Filtern:

- Bloom Hash-Filter läßt einige Looser von R_2 durch.
- $\rightarrow$ erhöhte Kommunikation durch größeres R_2'
- $\rightarrow$ erhöhte Kosten auf Site I zur Berechnung von $R_1 \bowtie R_2'$

Deferred Fetch
(Verallgemeinerung der Semijoin Idee)

Schema:

pic(Name, Gif) liegt in New York
audio(Name, Tape) liegt in London

Anfrage:

Zeige mir auf meinem PC in Passau die Photos von allen hübschen Leuten, die gut singen können.

Plan:

$$\mathbf{FETCH}(\mathbf{FETCH}(\pi(\sigma(pic), Name) \bowtie \pi(\sigma(audio), Name), \mathbf{Gif}), \mathbf{Tape})$$

- schicke alle *Namen* von hübschen Leuten nach Passau
- schicke alle *Namen* von musikalischen Leuten nach Passau
- schicke aber nur Bilder und Bänder von hübschen und musikalischen Leuten nach Passau
- (im Grunde ist **FETCH** wie ein zusätzlicher Join. Welche Joinmethode wird hier emuliert?)

Komprimierung

Wieso sollte man Daten nicht komprimieren, um Kommunikationskosten zu sparen? Das tut man ja auf dem Internet auch.

Entscheidungen:

- Daten werden komprimiert gespeichert vs. Daten werden während der Bearbeitung der Anfrage komprimiert
- Granularität der Komprimierung
- Wahl der Kompressionsverfahren

Granularität

In welcher Einheit sollten Daten komprimiert werden?

- komplette Datenbasis: oh weh!
- ganze Tabelle: paßt nicht ins Iteratormodell; Indexe sind Hölle
- einzelne Blöcke: gut für "on-the-fly" Komprimierung; schlecht für permanente Komprimierung, da Dateiorganisation schwierig
- einzelne Tupel: okay aber immer noch kein Winner
- einzelne Felder: ja!!!

Worauf man achten sollte:

- Lazy Decompression; dekomprimiere nur das, was auch wirklich benötigt wird. Lazy Decompression nur bei einer sehr feinen Granularität möglich.
- spezielle Komprimierung für spezielle Datentypen auch nur bei einer sehr feinen Granularität möglich
- halte Daten in komprimierter Form im Puffer ist auch nur bei einer feinen Granularität möglich

N.B.: Wir komprimieren einzelne Felder, aber wir wenden dasselbe Verfahren auf alle Felder einer Spalte einer Tabelle an. Wieso?

Kompressionsverfahren

Die Klassischen:

- Ziv-Lempel
- Huffman Coding
- ...

... nur geeignet bei grober (Block) Granularität; in der Regel langsam; Problem mit der Ordnungserhaltung bei Indexen

stattdessen:

- null string surpression
- besondere Behandlung von NULL Werten
- dictionary-based compression

Genereller Tradeoff:

- hohe Komprimierungsraten (sparen IO Kosten und Speicherkosten)
- schnelle Komprimierung/Dekomprimierung (spart CPU Kosten)
- Regel in DBMS: achte zuerst auf schnelle Komprimierung/Dekomprimierung

Komprimierungsverfahren

Komprimierung eines 4-Byte Unsigned Ints:

- streiche führende Nullen
- 0 bytes für NULL Wert
- 1 byte für Zahlen 0 bis 255
- 2 byte für Zahlen 256 bis 65535
- ...
- Vorzeichen werden separat kodiert

Komprimierung eines 8-Byte Doubles:

- 0 bytes für NULL Wert
- 4 bytes, falls Präzision von 4 bytes ausreicht
- 8 bytes, falls Präzision von 4 bytes nicht ausreicht

Komprimierung eines Datums:

- repräsentiere Datum als Integer
- definiere Basisdatum: z.B. 1.1.1980
- bilde Differenz zum Basisdatum:
  8.1.1980 wird repräsentiert durch Zahl 7
- komprimiere Zahl 7 wie normales integer

Strings:

- konvertiere String zu einem Varchar
  i.e., Lücken am Ende werden gelöscht
- komprimiere Längeninformation des Varchars als normalen Integer
- Nach Lust und Laune kann man noch zusätzlich Huffman Coding et al. auf das String anwenden

Dictionary-Based Verfahren

- funktioniert für beliebige Typen
- Annahme: domain hat nur n Werte
  (z.B. es gibt nur 5 Kontinente; Kontinente sind aber strings)
- lege separates Dictionary für diese Werte an; kodiere Werte mit $\mathbf{log}(n)$ bits
- (es gibt cleverere Verfahren wenn n nicht apriori bekannt ist; Huffman Coding ist im Grunde ein solches cleveres Verfahren)

N.B.: Es gibt spezielle Komprimierungsverfahren für Indexe; z.B. Präfix Komprimierung für B-Bäume.

Storage Layout und Dekomprimierung

Ein Tupel besteht aus bis zu fünf Teilen:

1. code für dictionary-based komprimierte Felder (fix)
2. Kodierung für andere Komprimierungsverfahren (fix)
3. unkomprimierte Werte (fix)
4. komprimierte Werte (variable Länge)
5. varchars und lange Felder (variable Länge)

Schnelle Dekomprimierung

- materialisiere Dekodierungsinformation in Tabellen (siehe Tafel)
- Freiheitsgrad: "Breite" (Speicherkosten - Laufzeit Tradeoff)

Distributed Object Assembly

- Erinnert Ihr Euch an Pointer-Based Joins aus der OO Vorlesung?
- Wie funktionieren Pointer-Based Joins im verteilten Fall?
- Alternative für Transitive Hüllen, lange Pfadausdrücke.

Nearest-First anstatt Breadth-First Search

1. group *Emps* such that the corresponding *Depts* referenced by a group of *Emps* are stored on the same site;
2. consider the first group of *Emps* and visit the site that stores the *Depts* for that group of *Emps*; at that site, fetch all the referenced *Dept* objects and, if any, also fetch all *Divisions* objects stored at that site and referenced by the *Dept* objects; return the *Dept* and *Division* objects;
3. at the original site (the site of the *Emp* objects), *Emp, Dept, Division* triplets can directly be printed out as query results; *Emp/Dept* need to be further expanded by grouping them in such a way that the *Divisions* referenced by a group of *Emp,Dept* pairs are stored on the same site; that is, the grouping of

Emp,Dept pairs is carried out just as the grouping of *Emps* in Step 1, and a group of *Emp,Dept* pairs can be expanded just as in Step 2; even better, when we visit a site in Step 2, we can expand *Emp* groups (generated in Step 1) and *Emp,Dept* groups (generated in this step) in one batch;

4. repeat Steps 2 and 3 until all *Emp* and *Emp,Dept* groups have been expanded

Was gibt es für Varianten?