Query Rewrite

Prinzip und Überblick

- Transformiere eine Anfrage.
- Zur Transformation werden Regeln definiert. Die Regeln können in beliebiger Reihenfolge angewendet werden. Die genauen Regeln sind ein streng gehütetes Geheimnis des DBMS Herstellers. Beispielregeln für IBM DB2 werden in [Pirahesh et al., SIGMOD 92] beschrieben.
- Beachte: Regelmenge wird ständig erweitert; z.B. Regeln für Outerjoins bei SQL 92.
- Ein Regelauswerter wendet die Transformationsregeln auf eine Anfrage an. Genaue Vorgehensweise des Regelauswerters ist ebenfalls für jedes System geheim. Es ist eine hohe Kunst einen solchen Regelauswerter zu bauen – er muß erweiterbar sein, und die Regelmenge hat keine schönen Eigenschaften wie z.B. *Konfluenz*; i.e.: **Es gibt keine Normalform zu einer Anfrage.**
- Anfrage liegt in einer internen Darstellung nicht in SQL vor. Dies ermöglicht Transformationen, die mit reinem SQL nicht möglich wären. Also auch Transformationen, die ein Benutzer nicht machen kann.

Query Rewrite

Ziele

- Führe Optimierungen durch, die garantiert (oder fast immer) gut sind – unabhängig von der Größe der Tabellen.
- Bereite Anfrage vor, so daß der kostenbasierte Optimierer möglichst viele Möglichkeiten zur Ausführung hat.

Beispiele

- Vereinfachung von Ausdrücken.
- Generierung von Joinprädikaten.
- **Unnesting** von Views und Subanfragen.
- **Predicate Move Around**
- (Transformation von globalen Anfragen in lokal ausführbare Anfragen. Siehe Verteilte DBMS Vorlesung.)

Interne Repräsentation

Grundsätze

- Jeder Block einer Anfrage (z.B. View oder Subanfrage) wird durch einen separaten Block in der internen Repräsentation dargestellt.
- Jeder Block hat einen Kopf mit Annotationen:
  - Muß die Ausgabe sortiert sein.
  - Duplikatelimination: *notwendig, erlaubt, verboten*
  - Verlangte Ergebnisspalten des Blocks.
- Jeder Block hat einen Körper:
  - Ausdrücke, die im Block berechnet werden, werden als Operatorbaum dargestellt.
  - "Joingraph" zwischen den Variablen (i.e., Tabellen), die im Block verarbeitet werden. D.h. Variablen werden durch Knoten und (Join-) Prädikate durch Kanten dargestellt.
- Abhängigkeiten zwischen Variablen aus verschiedenen Blöcken werden ebenfalls durch Kanten repräsentiert. Diese Quanten können zusätzlich quantifiziert werden (ALL, EXISTS, FROM).
- Üblicherweise erhalten Gruppierungen ihren eigenen Block.

Beispiel

```
SELECT e.name, e.salary
FROM Emp e
WHERE e.dno IN (SELECT d.dno
                FROM Dept d
                WHERE d.name = 'Research');
```

Rolle der internen Repräsentation

Anfragebearbeitung

- *Parser* legt die *erste* interne Repräsentation einer Anfrage an.
- *Rewrite* transformiert die interne Repräsentation in mehreren Schritten und erzeugt als Ergebnis die *beste* interne Repräsentation für eine Anfrage.
- Der *Optimizer* wird auf jeden Block der *besten* internen Repräsentation angewendet und erzeugt hierfür einen Plan.
- (Codegenerator stöpselt dann die einzelnen Pläne zusammen, so daß sie direkt ausführbar sind.)

Bemerkung

- Wenn Gruppierung als separater Block dargestellt wird, dann kann man keine kostenbasierte Optimierung mit dem Optimizer von GROUP-BY und Join machen. Das ist sehr schade!!!

Rewrite: Hinzunahme von Joinprädikaten

```
SELECT *
FROM A, B, C
WHERE A.a = B.b AND B.b = C.c;
```

wird durch Query Rewrite zu

```
SELECT *
FROM A, B, C
WHERE A.a = B.b AND B.b = C.c
AND A.a = C.c;
```

- Hierdurch wird der folgende Plan möglich:

  $$(A \bowtie C) \bowtie B$$

  (Ohne Query Rewrite wäre $A \bowtie C$ ein karthesisches Produkt.)
- Überflüssige Joinprädikate werden (nach der Optimierung) von Codegen entfernt.

Rewrite: Vereinfachung von Ausdrücken

```
SELECT *
FROM Squares s
WHERE s.length * s.length < 100;
```

wird durch Query Rewrite zu

```
SELECT *
FROM Squares s
WHERE s.length < 10;
```

- Nur möglich, wenn es eine Integritätsbedingung gibt, die besagt, daß $length \geq 0$.
- Verbilligt Auswertung des Prädikates.
- Macht die Anwendung eines Indexes möglich.

Rewrite: Unnesting

Wir betrachten drei Arten von Subqueries

- **Type A:** Subquery ist konstant und liefert einen einfachen Wert zurück.
- **Type N:** Subquery ist konstant und liefert eine Tabelle (i.e., Menge von Werten) zurück.
- **Type J:** Subquery ist abhängig von der äußeren Anfrage und liefert eine Tabelle zurück.

Bemerkung

- Es gibt noch viele weitere (speziellere) Anfragetypen und spezielle Regeln, diese zu optimieren.
- Wir konzentrieren uns auf die einfachsten Fälle. Es gibt viele subtile Feinheiten und auch kleine Unterschiede zwischen z.B. SQL und OQL, die das Regelwerk sehr stark beeinflußen.

Rewrite: Type A

```
SELECT *
FROM Emp e
WHERE e.salary = (SELECT max(salary)
                  FROM Emp);
```

wird durch Rewrite zu

```
define m = SELECT max(salary)
           FROM Emp;
SELECT *
FROM Emp e
WHERE e.salary = m;
```

- **define** weiteres Gimmick der internen Repräsentation.
- Statt n Iterationen durch die Emp Tabelle nur zwei.
- Schafft man es auf eine Iteration zu kommen?

Rewrite: Type N

```
SELECT *
FROM Emp e
WHERE e.dno IN (SELECT d.dno
                FROM Dept d
                WHERE d.name = 'Research');
```

wird durch Rewrite zu

```
SELECT e.*
FROM Emp e, Dept d
WHERE e.dno = d.dno AND d.name = 'Research';
```

- Jede beliebige Joinmethode ist jetzt anwendbar.
- **VORSICHT!** Diese Transformation klappt in dieser Art nur bei funktionalen Abhängigkeiten (i.e., `dno` ist Schlüssel der `Dept` Tabelle).
- Funktioniert auch, wenn die Anfragen nach *distinct(e.*)* fragt.
- Im Allgemeinen kann man Type N Anfragen durch *Semijoins* oder *Antijoins* ersetzen.

Rewrite: Type J

```
SELECT *
FROM Emp e
WHERE e.dno IN (SELECT d.dno
                FROM Dept d
                WHERE d.budget > e.salary * 10);
```

wird durch Rewrite zu

```
SELECT e.*
FROM Emp e, Dept d
WHERE e.dno = d.dno AND d.budget > e.salary * 10;
```

- Dieselbe Soße wie bei Type N.
- (In besonderen Fällen gibt es allerdings krisselige Unterschiede; z.B. bei Allquantoren.)
- Wiederum ist dieselbe Vorsicht bzgl. Duplikate geboten.

Rewrite: FROM Klausel

```
SELECT *
FROM Emp e, (SELECT dno
             FROM Dept
             WHERE name = 'Research') d
WHERE e.dno = d.dno;
```

wird durch Rewrite zu

```
SELECT e.*
FROM Emp e, Dept d
WHERE e.dno = d.dno AND d.name = 'Research';
```

- Meist ist Unnesting in der FROM Klausel viel einfacher, weil es weniger Fälle gibt (z.B. keine Allquantoren).
- Was haben wir durch Rewrite hier gewonnen?
- Was ist mit Nesting in der SELECT Klausel?

Rewrite: ALL und ANY

```
SELECT *
FROM Emp e
WHERE e.salary > ANY(SELECT m.salary
                     FROM Manager m
                     WHERE p(e, m));
```

wird durch Rewrite zu

```
SELECT *
FROM Emp e
WHERE e.salary > (SELECT MIN(m.salary)
                  FROM Manager m
                  WHERE p(e, m));
```

- Analoge Regeln gelten für ALL und andere Vergleichsoperatoren.
- Abhängig von p sind danach weitere (Type A oder J) Transformationen möglich.

Existenzquantoren

```
SELECT *
FROM Emp e
WHERE e.salary EXISTS(SELECT m.salary
                      FROM Manager m
                      WHERE p(e, m));
```

wird durch Rewrite zu

```
SELECT *
FROM Emp e
WHERE 0 < (SELECT count(m.salary)
           FROM Manager m
           WHERE p(e, m));
```

- Abhängig von p sind danach weitere (Type A oder J) Transformationen möglich.
- Analog funktioniert es für **not exists**.
- Allquantifizierung insgesamt etwas schwieriger. Siehe: Claußen et al., VLDB 1997.

Rewrite: Predicate Movearound

```
SELECT *
FROM Emp e, (SELECT name, dno
             FROM Dept
             WHERE dno = e.dno AND boss = e.name
                   AND boss like '%Affe%') d
WHERE e.dno BETWEEN 1 and 10;
```

wird durch Rewrite zu

```
SELECT *
FROM Emp e, (SELECT name, dno
             FROM Dept
             WHERE dno = e.dno AND boss = e.name
                   AND boss like '%Affe%' AND
                   d.dno BETWEEN 1 and 10) d
WHERE e.dno BETWEEN 1 and 10 AND
      e.name like '%Affe%';
```

- Gut, wenn Unnesting scheitert oder nicht implementiert ist.
- Ähnliche Regeln gibt es auch für *distinct* Annotationen, die man zwischen Blöcken bewegen kann.

Speicherallokation

Zur Erinnerung

- Der Systempuffer ist segmentiert.
- Operatoren wie Join und Group-By erhalten ihren eigenen statischen Pufferbereich.
- (Scans und Indexscans auf dieselben Tabellen und Indexe teilen sich einen Pufferbereich.)

Aufgabe

- Für jeden Operator (außer Scan und Indexscan) eines Planes muß bestimmt werden, wieviel Puffer er bekommt.

Ansätze

1. Systemadministrator setzt Tuningparameter. So wird es bei fast jedem DBMS heute gemacht. (Z.B. SortArea, HashArea, etc.)
2. Scheduler wirft zur Laufzeit eines Planes eine besondere Routine an. Das möchte ich heute vorstellen.
3. Optimierer erzeugt verschiedene Pläne mit unterschiedlichen Allokationen für jeden Operator zur Übersetzungszeit. Scheduler sucht sich einen der Pläne zur Laufzeit aus.

Speicherallokation

Bemerkungen

- Ansatz 3 ist potentiell der Beste.
- Ansatz 3 ist sehr komplex. Es hat noch niemand geschafft, das überzeugend hinzubekommen.
- Das ganze Thema ist sehr *schmutzig* und Ihr werdet bald sehen wieso.

Problembeschreibung

Gegeben:

- Menge von Operatoren $O_1, \ldots, O_k$.
- Minimaler und Maximaler Speicherbedarf jedes Operators. (Beispiel: Ein Hash Join benötigt minimal $\sqrt{N}$, um nicht rekursiv partitionieren zu müssen, und maximal $F * N$ Seiten Puffer, wobei N die Größe der kleineren Tabelle ist.)
- Größe M des zur Verfügung stehenden Puffers. (I.e., Puffer, der nicht für andere Operatoren oder Indexe etc. reserviert ist.)

Gesucht:

- Allokation M_i für jeden Operator O_i.
- $\Sigma_{i=1}^{k} M_i \leq M$.
- $sum_{i=1}^{k} \text{cost}(O_i, M_i)$ minimal.
- $\forall i : \min_i \leq M_i$.

Problembeschreibung

Bemerkungen

- Wir ignorieren längerfristige Puffereffekte für Indexe und Tabellen. (Wieso?)
- $O_1, \ldots, O_k$ ist eine Menge von Operatoren innerhalb eines Planes, die gleichzeitig ausgeführt werden. (Pipelining oder unabhängig parallel.)
- Pufferallokation muß also evtl. mehr als einmal pro Anfrage gemacht werden.
- **Vorsicht Falle:** Manche Operatoren (z.B. Hash Join) sind bei bestimmten Allokationen Pipelinebreaker und bei anderen nicht. Dieses Phänomen ignorieren wir hier.
- **Frage:** Beeinflußt die Pufferallokation der ersten Gruppe von Operatoren evtl. die zweite Gruppe von Operatoren?

Algorithmus: FAIR

Prinzip

- $M_a = M - \Sigma_{i=1}^{k} \min_i$
- Falls $M_a < 0$, Anfrage muß warten, bis mehr Puffer zur Verfügung steht. (Exit.)
- $M_n = \Sigma_{i=1}^{k} \max_i - \min_i$
- Falls $M_n > M_a$, gebe allen Operatoren maximalen Puffer. (Exit.)
- Allokiere $\min_i + \frac{M_a}{M_n} * (\max_i - \min_i)$ Puffer für O_i.

Bewertung

- Dieser Algorithmus bevorzugt Operatoren mit einem besonders hohen Speicherbedarf.
- Es zeigt sich, daß das ein besonders dummer Ansatz ist. (Wieso?)

Algorithmus: SMALL

Prinzip

- $M_a = M - \Sigma_{i=1}^{k} \min_i$
- Falls $M_a < 0$, Anfrage muß warten, bis mehr Puffer zur Verfügung steht. (Exit.)
- Sortiere die O_i steigend nach $\max_i - \min_i$.
- Solange $M_a > 0$, gebe $\min\{M_a, \max_j - \min_j\}$ an den nächsten Operator O_j.

Bewertung

- Dieser Algorithmus bevorzugt die *kleinen* Operatoren.
- Anders ausgedrückt: Dieser Algorithmus versucht möglichst vielen Operatoren maximale Allokation zu geben.
- Es zeigt sich, daß das ein sehr guter Ansatz ist.
- (Noch ein bißchen besser wird es, wenn man den Rucksack optimal packt.)

Algorithmus: BENEFIT

Prinzip

- $M_a = M - \Sigma_{i=1}^{k} \min_i$
- Falls $M_a < 0$, Anfrage muß warten, bis mehr Puffer zur Verfügung steht. (Exit.)
- setze $M_i = \min_i$
- **for** $l = 1$ **to** M_a: setze $M_i = M_i + 1$, falls $\mathrm{MG}(O_i, M_i)$ maximal. D.h. gebe Pufferseite l an den Operator mit dem höchsten *Marginal Gain* (MG).
- Der MG eines Operators O_i mit Allokation M_i ist definiert als:

$$\mathrm{MG}(O_i, M_i) = \mathrm{cost}(O_i, M_i) - \mathrm{cost}(O_i, M_i + 1)$$

- Logischerweise: $\mathrm{MG}(O_i, \max_i) = 0$.

Bewertung

- Dieser Algorithmus funktioniert nun wirklich gut.
- Beachte allerdings die vielen vereinfachenden Annahmen, die wir auf dem Weg gemacht haben.
- Hat außerdem höheren Aufwand. Dieser ist allerdings vertretbar, wenn die Marginal Gains eines Operators als konstant angenommen werden.

Scheduling

- Man kann noch zusätzlich einschränken, wann eine Anfrage (oder Menge von Operatoren) zugelassen wird, wenn sie *minimale* aber nicht *maximale* Allokation bekommt.
- Mein Lieblingsansatz verwendet hierzu Queueing Network Models, um zu entscheiden, ob sich der Durchsatz des Systems bei Zulassung der Anfrage erhöht oder senkt. Voraussetzung ist wie beim Benefit Ansatz genaue Kenntnisse über die Kosten der einzelnen Operatoren. Literatur: Faloutsos et al., VLDB 1991.
- Existierende Systeme machen solche (komplizierten) Einschränkungen natürlich nicht und lassen immer zu, sobald minimale Allokation gewährt werden kann.