

Datenbanksysteme

Wintersemester 2015/16

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Verschachtelte (Engl. Nested) Anfragen

Wieso und wo gibt es verschachtelte Anfragen?

- Unteranfragen können beinahe überall in SQL auftreten, z.B. im SELECT, als "Relation" in FROM oder natürlich in der WHERE Klausel.
- Macht Formulierung von Anfragen oft angenehmer, da natürlicher, vgl. z.B. WHERE EXISTS
- Für Anfrageverarbeitung in der Regel allerdings nicht wünschenswert, da naive Ausführung teuer

Literaturempfehlung: Ganski und Wong. Optimization of Nested SQL Queries Revisited. SIGMOD 1987.

<http://www.csd.uoc.gr/~hy460/pdf/p23-ganski.pdf>

bzw. auch Won Kim. On optimizing SQL-like nested query. ACM TODS, 1982.

Klassifizierung (Typen) von Schachtelungen

Typ A Verschachteltes Prädikat, welches einen (aggregierten Wert) liefert und **unabhängig** von äußerer Anfrage ist.

```
SELECT *  
FROM exams e  
WHERE e.note = (SELECT max(note) FROM exams)
```

Typ N Verschachteltes Prädikat, bei dem innere Anfrage eine Relation zurückgibt, aber **unabhängig** ist von äußerer Anfrage.

```
SELECT *  
FROM exams e  
WHERE e.department IN (SELECT f.id  
                        FROM department f  
                        WHERE f.name='Informatik')
```

Typen nach Won Kim '82. On optimizing SQL-like nested query.

Klassifizierung (Typen) von Schachtelungen

Typ J Verschachteltes Prädikat. Innere Anfrage referenziert äußere Anfrage, ist also **abhängig (korreliert)**. Keine Aggregation in innereren Anfrage.

```
SELECT BestNr
FROM Bestellung B
WHERE ProdNr IN (SELECT ProdNr
                   FROM Lieferung L
                   WHERE L.LiefNr = B.BestNr
                   AND L.Datum = current_date)
```

Typ JA Wie J nur nun mit Aggregation.

Typen nach Won Kim '82. On optimizing SQL-like nested query.

Entschachteln von Anfragen: Konstante Unteranfragen

```
SELECT *  
FROM exams e  
WHERE e.note = (SELECT max(note) FROM exams)
```

- **Naive Ausführung:** Für jedes Tupel aus Exams e wird ein Mal über Exams iteriert. **Teuer.**
- **Unterfrage berechnet einen konstanten Wert**
- **Kann vom Anfrageoptimierer erkannt, berechnet und als Konstante bereitgestellt werden.**

```
DEFINE m = (SELECT max(note) from Exams)
```

```
SELECT *  
FROM exams e  
WHERE e.note = m
```

Entschachteln von Anfragen: Unabhängige Unteranfragen

```
SELECT *  
FROM exams e  
WHERE e.department IN (SELECT f.id  
                        FROM department f  
                        WHERE f.name='Informatik')
```

- **Unterfrage ist unabhängig von der äußeren Anfrage**
- Hier eine **Existenzaussage** via **IN**, ähnlich/identisch zu **EXISTS** oder **ANY**.
- Kann einfach entschachtelt werden mit Hilfe eines Joins bzw. Semijoin, unten dargestellt.
- Oder Unteranfrage wird berechnet, materialisiert und als Filter benutzt.

```
SELECT DISTINCT e.*  
FROM exams e, department f  
WHERE e.department = f.id AND f.name='Informatik'
```

Verschachtelte vs. Unverschachtelte Anfrage

```

SELECT p.pname,
  ( SELECT c.city
    FROM company c
    WHERE p.cid = c.cid ) AS city
FROM product p;
  
```

**Kann leicht
entschachtelt
werden:**

```

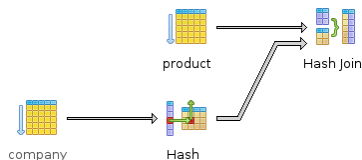
SELECT p.pname, c.city
FROM product p, company c
WHERE p.cid = c.cid
  
```

Achtung: Annahme im Beispiel, dass jedes Produkt auch Firma und somit City hat, sonst unten LEFT OUTER JOIN nötig.

Sequential Scan auf company
mit Filter p.cid=c.cid:



**Nach Entschachtelung
Plan mit Hashjoin:**



Entschachteln von Anfragen: Abhängige Unteranfragen

```
SELECT BestNr
FROM Bestellung B
WHERE ProdNr IN (SELECT ProdNr
                  FROM Lieferung L
                  WHERE L.LiefNr = B.BestNr
                  AND L.Datum = current_date)
```

Kann entschachtelt werden zu

```
SELECT BestNr
FROM Bestellung B, Lieferung L
WHERE B.ProdNr = L.ProdNr
      AND L.LiefNr = B.BestNr
      AND L.Datum = current_date
```


Entschachteln von Anfragen: Abhängige Unteranfragen

Unteranfragen müssen nicht nur im WHERE Block stehen: Berechne für jede Firma die Anzahl an Produkten, die dort hergestellt werden.

```
SELECT DISTINCT C.cname, (SELECT count(*)  
                        FROM Product P  
                        WHERE P.cid = C.cid)  
FROM Company C
```

Kann mit Hilfe von GROUP BY entschachtelt werden zu

```
SELECT C.cname, count(*)  
FROM Company C, Produkt P  
WHERE C.cid = P.cid  
GROUP BY C.cname
```

Ist das so korrekt?

Entschachteln von Anfragen: Abhängige Unteranfragen

Nein, die Entschachtelung war nicht korrekt. Was ist mit Firmen, die gar keine Produkte haben?

```
SELECT C.cname, count(p.name)
FROM Company C LEFT OUTER JOIN Product P
ON C.cid = P.cid
GROUP BY C.cname
```

Verschachtelte Anfragen mit EXIST, IN, ANY

Finde alle Firmen mit Produkten günstiger als 150.

```
SELECT DISTINCT c.cname  
FROM company c  
WHERE EXISTS (SELECT *  
                FROM product p  
                WHERE c.cid=p.cid and p.price < 150);
```

Wie kann man diese Anfrage mittels **IN** oder **ANY** ausdrücken?

Verschachtelte Anfragen mit EXIST, IN, ANY (2)

```
SELECT DISTINCT c.cname  
FROM company c  
WHERE c.cid IN (SELECT p.cid  
                FROM product p  
                WHERE c.cid=p.cid and p.price < 150);
```

```
SELECT DISTINCT c.cname  
FROM company c  
WHERE 150 > ANY (SELECT p.price  
                FROM product p  
                WHERE c.cid=p.cid);
```

Nun zur Entschachtelung ...

Verschachtelte Anfragen mit EXIST, IN, ANY (3)

```
SELECT DISTINCT c.cname  
FROM company c  
WHERE EXISTS (SELECT *  
                FROM product p  
                WHERE c.cid=p.cid and p.price < 150);
```

Kann entschachtelt geschrieben werden als

```
SELECT DISTINCT c.cname  
FROM company c, product p  
WHERE c.cid = p.cid and p.price < 150
```

Existenzquantoren sind einfach zu entschachteln.

Allquantor

Product (pname, price, cid) Company(cid, cname, city)

Finde alle Firmen, die nur Produkte mit Preis unter 150 haben.

```
SELECT DISTINCT C.cname
FROM Company C
WHERE NOT EXISTS (SELECT *
                   FROM Product P
                   WHERE P.cid = C.cid AND P.price >= 150)
```

Wie sehen die **alternativen Formulierungen** mit **ALL** und **NOT IN** aus?

Entschachteln von Anfragen

Beispielanfrage, die für jeden Studenten dessen bestes Examen berechnet:

```
SELECT s.name, e.course
FROM students s, exams e
WHERE s.id = e.sid AND
      e.grade = (SELECT min(e2.grade)
      FROM exams e2
      WHERE s.id = e2.sid)
```

- **Unteranfrage ist abhängig von der äußeren Anfrage**
- Für jedes Paar von Student und Examen (s,e) muss im Prinzip geschaut werden ob diese Prüfung auch die beste des Studenten ist.
- Ein sogenannter **dependent Join**.

Entschachteln von Anfragen

Die vorherige Anfrage kann umgeformt werden zu

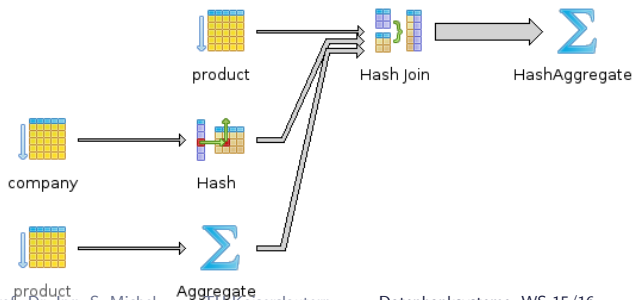
```
SELECT s.name, e.course
FROM students s, exams e,
      (SELECT e2.sid as id, min(e2.grade) as best
       FROM exams e2
       GROUP BY e2.sid) m
WHERE s.id = e.sid AND
       s.id=e.sid AND m.id=s.id AND e.grade=m.best
```

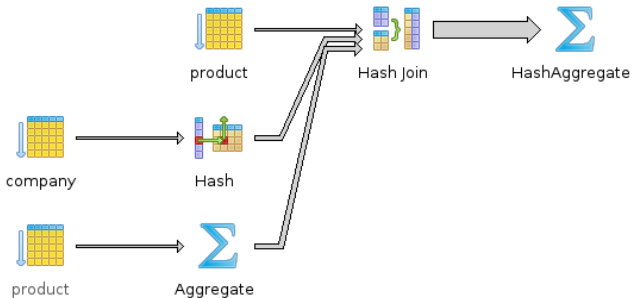
D.h. wir haben zwar eine Verschachtelung, aber keine abhängige Unteranfrage. Prima.

Dependent Join

Analog, finde Firmennamen und Produktnamen der teuersten Produkte der Firma.

```
SELECT DISTINCT c.cname, px.pname
FROM company c, product px
WHERE c.cid = px.cid
      and px.price = (SELECT max(p.price)
                     FROM product p
                     WHERE c.cid=p.cid);
```





	QUERY PLAN text
1	HashAggregate (cost=2182.64..2182.65 rows=1 width=436)
2	Group Key: c.cname, px.pname
3	-> Hash Join (cost=14.25..2182.64 rows=1 width=436)
4	Hash Cond: ((px.cid = c.cid) AND (px.price = (SubPlan 1)))
5	-> Seq Scan on product px (cost=0.00..13.10 rows=310 width=226)
6	-> Hash (cost=11.70..11.70 rows=170 width=222)
7	-> Seq Scan on company c (cost=0.00..11.70 rows=170 width=222)
8	SubPlan 1
9	-> Aggregate (cost=13.88..13.89 rows=1 width=4)
10	-> Seq Scan on product p (cost=0.00..13.88 rows=2 width=4)
11	Filter: (c.cid = cid)

Dependent Join

Dependent Join

- Naiver Nested Loops Join, bei dem die rechte Relation von dem aktuellen Tupel der linken Seite abhängt.
- Zur Erinnerung, “normaler Join”

$$T_1 \bowtie_p T_2 := \sigma_p(T_1 \times T_2)$$

- Dependent Join hingegen ist definiert als

$$T_1 \bowtie_p T_2 := \{t_1 \circ t_2 \mid t_1 \in T_1 \wedge t_2 \in T_2(t_1) \wedge p(t_1 \circ t_2)\}$$

- Wenn wir mit $\mathcal{A}(T)$ die Attribute eines Ausdrucks T der rel. Algebra bezeichnen und mit $\mathcal{F}(T)$ die freien Variablen in T , dann muss $\mathcal{F}(T_2) \subseteq \mathcal{A}(T_1)$ gelten.

Dependent Join und Potenzial von Entschachtelung

Ziel der Entschachtelung

- Ersetze dependent Join durch “normalen” Join
- Wieso?

Potenzial

- Nach der Entschachtelung können andere Implementierungen des Join-Operators benutzt werden, nicht nur Nested Loops Join.
- Dies bringt natürlich mehr Flexibilität und bringt in der Regel große Gewinne in der Performance der Anfrage.

Weitere Literatur: Aktuelle Arbeit von Neumann und Kemper. Unnesting Arbitrary Queries. Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW), 2015.

Optimierung von mehreren Anfragen

Multi Query Optimization

- Gegeben eine Menge von Anfragen, die von einem oder mehreren Benutzern “zeitgleich” an die DB geschickt wurden.
- DB kann nun schauen ob Anfragen Gemeinsamkeiten haben, um Berechnungskosten für die gesamte Menge zu verringern.

Beispiel: Gegeben sind folgende Anfragen

```
SELECT * FROM (R NATURAL JOIN T) NATURAL JOIN S  
SELECT * FROM (R NATURAL JOIN U) NATURAL JOIN S
```

Welche Gemeinsamkeiten gibt es hier?

Optimierung von mehreren Anfragen (2)

```
SELECT * FROM (R NATURAL JOIN T) NATURAL JOIN S  
SELECT * FROM (R NATURAL JOIN U) NATURAL JOIN S
```

- Der Joinoperator ist kommutativ und assoziativ, wie wir gesehen haben.
- Also, beide Anfragen haben gemeinsam den Join zwischen R und S.

Optimierung von mehreren Anfragen (3)

Wir können die Anfragen also “umschreiben” zu ...

```
SELECT * FROM (R NATURAL JOIN S) NATURAL JOIN T  
SELECT * FROM (R NATURAL JOIN S) NATURAL JOIN U
```

- ... und die **Teilanfrage (R NATURAL JOIN S) nur ein Mal ausführen.**
- Was gibt es dabei zu beachten?
- Evtl. ist die Teilanfrage (R **NATURAL JOIN** S) sehr teuer.
- **Mögliche Vorgehensweise:** Jede Anfrage wird getrennt voneinander übersetzt und dann erst geschaut ob es gemeinsame Teilausdrücke gibt.
- Eine einfache Möglichkeit: **Shared Scans**, d.h. lesen der Relationen wird nur ein Mal ausgeführt, für versch. Anfragen.

Materialisierte Sichten (Materialized Views)

Vgl. (Dynamische) Sicht

- =Makro für Query
- Ergebnis der Anfrage wird nicht vorberechnet
- Rechenaufwand zum Zeitpunkt der Anfrage (=query time)
- erst wenn die Sicht benutzt wird, wird auch das Ergebnis der Sicht berechnet

Materialisierte Sicht

- Ergebnis der Sicht wird **vorberechnet**
- wenn eine Anfrage die Sicht benutzt, ist das Ergebnis der Sicht bereits vollständig berechnet
- Rechenaufwand vorher
- Problem bei Updates:
Tabellen, die zur Vorbereitung der Sicht benutzt werden, ändern sich ⇒ Materialisierte Sicht muss angepasst werden

Ausnutzen von Materialisierten Sichten

Gegeben eine **materialisierte Sicht**, die durch folgende Anfrage erzeugt wurde:

```
SELECT  $L_V$   
FROM  $R_V$   
WHERE  $C_V$ 
```

Daneben haben wir eine **Anfrage**, die wie folgt aufgebaut ist

```
SELECT  $L_Q$   
FROM  $R_Q$   
WHERE  $C_Q$ 
```

L_V ist eine Liste von Attributen, R_V eine Liste von Relationen und C_V ein Prädikat. **Kann teil der Anfrage durch Sicht V ersetzen, falls folgendes gilt:**

- Alle Relationen auf R_V treten auch in der Liste R_Q auf
- Das Prädikat C_Q ist äquivalent zu C_V **AND** C , für ein Prädikat C . Oder einfach nur identisch zu C_V .
- Für den Fall, dass C gebraucht wird müssen entsprechende Attribute in L_V vorhanden sein, bzw. in den Relationen R_V .
- Attribute aus L_Q sind auch in L_V

Annutzen von Materialisierten Sichten (2)

```
SELECT  $L_V$   
FROM  $R_V$   
WHERE  $C_V$ 
```

```
SELECT  $L_Q$   
FROM  $R_Q$   
WHERE  $C_Q$ 
```

... also falls die Bedingungen von zuvor zutreffen, dann kann man die Anfrage so anpassen, dass die View benutzt wird, durch:

- Ersetze die Liste von Relationen R_Q durch die View V und alle restlichen Relationen, die nicht in R_V sind.
- Ersetze C_Q durch C , falls C nicht gebraucht wird (weil $C_V = C_Q$), dann gibt es keine WHERE Klausel.

Materialisierte Sichten Bestimmung

(engl. **Materialized View Selection**)

- Welche Views sollen materialisiert werden?
- Frage ist ähnlich zur Frage welche Indexe angelegt werden sollen.
- Auch hier müssen Information zum **Workload** (welche Queries, wie oft, welche Indexe gibt es, welche Updates) gegeben sein.

Richtlinien, welche Sichten evtl. angelegt werden sollen

- Sichten sollten Relationen in **FROM** Klausel haben, die Teilmenge der FROM Klausel von Anfragen des Workloads sind.
- Die **WHERE** Klausel besteht aus einem Prädikat, das Teil eines konjunktiven (AND) Prädikats von Anfragen ist.
- Attribute, die in der **SELECT** Klausel sind ausreichend um Anfragen des Workloads zu beantworten.

Updates und Materialisierte Sichten

- Wenn sich die Tabellen, die einer materialisierten Sicht zugrunde liegen ändern, so haben diese Änderungen auch Auswirkungen auf die Sicht.
- Für jede kleine Änderung eine komplette Neuberechnung der Sicht auszuführen ist natürlich keine gute Idee.

Inkrementelle Instandhaltung von materialisierten Sichten

- Z.B. Sicht $V = R \bowtie S$. Relation R wird verändert. Wir haben also V^{alt} , R^{alt} und R^{neu} .
- Versuchen dann V^{neu} zu bestimmen, ohne neue Berechnung.
- **Fall Insert:** Betrachte dazu in R^{alt} eingefügte Tupel i_R . Dann $V^{neu} = V^{alt} \cup (i_r \bowtie S)$. Ähnlich bei Delete, Selektion, ...
- Auch periodische oder durch andere Kriterien angestoßene Neuberechnung, z.B. nachts.

Zusammenfassung Anfrageverarbeitung und Optimierung

- Wie kann auf Daten zugegriffen werden? Gibt es Indexe und wie liegen die Daten auf der Festplatte?
- Diverse Pufferstrategien wie CLOCK, LRU.
- Wie können Joins implementiert werden? Hash vs. Sort-Merge vs. Nested-Loops
- Welche Implementierung bzw. welcher Index eignet sich für Anfrage?
- Welche Option ist am günstigsten?
- Regelbasierte und Kostenbasierte Optimierung.
- Wie werden Kosten berechnet bzw. abgeschätzt?
- Suchraum der Anfrageoptimierung.
- Weitere Tricks zur Anfrageoptimierung, z.B. Entschachteln von Anfragen.