Databases and Information Systems

10. Übungsblatt

(Testatwoche: 22. - 24. Juni 2010)

Einführung in Datenbanksysteme
Datenbanken für die Bioinformatik

Heinz Schweppe, Katharina Hahn

Aufgabe 1 (Concurrency Control) *4+3 Punkte*

a) Betrachten Sie folgende Historie:

r1(a), r1(b), r2(a), r2(b), w2(a), c2, w1(b), c1

(1) Geben Sie die Ausgabe für das strikte 2PL an. Geben Sie dort, wo es im Protokoll vorgesehen ist, die entsprechenden Lock- und Unlock-Operationen an. Die Locks werden schrittweise vergeben, d.h. bei read und write Zugriff erst ein Read-Lock (lockS), dann ein Write-Lock (lockX).

T1	T2
lockS(a) r1(a) lockS(b) r1(b)	
	lockS(a) r2(a) lockS(b) r2(b) lockX(a) → wait f. T1
lockX(b) → wait f. T2 deadlock	

(2) Geben Sie die Ausgabe für das Snapshot Isolation Protokoll für obige Historie an. Geben Sie die Ausgabe für die Variante „first-committer-wins“ an.

Die Write-Sets der beiden Transaktionen sind disjunkt, daher läuft das Protokoll durch und beide Transaktionen werden abgeschlossen.

T1	T2
r1(a0) r1(b0)	
	r2(a0) r2(b0) w2(a2) c2
w1(b1) c1	

b) MVCC: **r2(a), w2(a) r1(b), r3(b), w3(b), r1(a), c1, r2(c), c2, r3(c),w3(c), c3**

T1 – read-only	T2	T2
	lockS(a) r2(a0) lockX(a) w2(a2)	
r1(b0)		
		lockS(b) r3(b0) lockX(b) w3(b3)
r1(a0) c1		
	lockS(c) r2(c0) unlock(All) + c2	
		lockS(c) r3(c0) lockX(c) w3(c3) unlock(All) + c3

Aufgabe 2 (Recovery) *2+2+2 Punkte*

a) Die Commit-Regel (auch „force at commit“) besagt bekanntlich, dass der Commit-Log-Eintrag in den persistenten Speicher geschrieben werden muss. Gilt das auch für den Abort-Log-Eintrag für abgebrochene Transaktionen (mit Begründung)?

Nein. Wenn ein Systemfehler auftritt, werden sowieso Winner- und Loser-TAs ermittelt. TAs, die abgebrochen worden sind, zählen automatisch als Verlierer und durch das Wiederherstellungsverfahren ist gewährleistet, dass ihr Effekt nicht in die DB geschrieben wird.

b) Damit logisches Protokollieren (logical log – es werden die Operationen protokolliert) funktioniert, muss die jeweilige Operation ein Inverses haben. Geben Sie ein Beispiel für eine Datenbankoperation an, die kein Inverses hat. Wie könnte ein Verfahren aussehen, dass mit dieser Situation umgehen kann.

Logisches Protokollieren bedeutet, dass nur die Operationen gespeichert werden, die vom alten Zustand zum neuen Zustand führen anstatt – wie bei Physischem Logging – den alten und den neuen Zustand zu sichern, was deutlich aufwendiger ist. Problematisch ist das Logische Protokollieren, wenn es zu einer Operation kein Inverses gibt. Beispiele dafür sind die mathematischen Operationen Modulo und Quadrieren. Hier kann vom neuen Zustand und der durchgeführten Operation nicht auf den alten Zustand geschlossen werden.

Eine Lösung, mit dieser Situation umzugehen, ist folgende: Man speichert bei allen Operationen, die ein Inverses haben, lediglich die Operation. Bei denen, die kein

Inverses haben, muss jedoch auch der alte Zustand gespeichert werden. Der neue Zustand braucht nicht geloggt werden, der lässt sich ja jederzeit durch Anwendung der Operation herleiten.

c) Wir betrachten die Undo-Verarbeitung für eine Transaktion TA1, die abgebrochen wird. Sie hat ein Objekt in einer Seite verändert, der LSN-Eintrag 233. Danach hat TA2 ein Objekt der Seite verändert, LSN 234, und die Seite wurde zurückgeschrieben (LSN der Datenseite jetzt 234). TA2 endet mit commit, danach erfolgt das abort von TA1. Welche LSN hat die Datenseite nach der Undo-Verarbeitung?

Nach der Undo-Verarbeitung ist die LSN der Datenseite >= 235. Denn auch die Undo-Operation, die die Änderung durch Transaktion TA1 kompensiert, muss geloggt werden. Da die LSNs monoton wachsend vergeben werden, muss die nach Undo-Verarbeitung größer sein als 234.

(Im Detail: jede Veränderung eines Wertes zieht einen Log-Eintrag mit sich, wird dieser Wert wieder zurück geändert, muss auch dies protokolliert werden → LSN 235.)

Aufgabe 3 (Data-Warehouse) *3+3 Punkte*

Für diese Aufgabe wird Ihnen die sogenannte TPC-H Datenbank zur Verfügung gestellt. Hierbei handelt es sich um eine Datenbasis für einen Benchmark im Bereich Decision Support Systeme. Die Datenbank bildet den Beschaffungs- und Verkaufsprozess eines global agierenden Unternehmens ab. Details entnehmen Sie bitte der Dokumentation des TPC-H Benchmarks (**http://www.tpc.org/tpch/spec/tpch2.11.0.pdf**): Für Sie relevant ist Seite 12, auf der das Schemas abgebildet ist.
Die Datenbank steht Ihnen auf *esel* **nur unter Oracle** im Schema *tpch* zur Verfügung. Beantworten Sie die folgenden Anfragen in jeweils gültigem SQL mit Hilfe des CUBE bzw. ROLLUP Operators.

a) Schreiben Sie eine Anfrage, die den durchschnittlichen Preisnachlass (*discount*) der Einzelposten von Bestellungen nach Herkunft der Kunden (*customer*) gruppiert. Dabei sollen die Hierarchisierungsstufen Nation (*nation*) und Region (*region*) der Dimension Herkunft betrachtet werden. Das Ergebnis soll sowohl den Preisnachlass pro Nation als auch pro Region auflisten.

```
SELECT r.r_name, n.n_name, AVG(l.l_discount)
FROM tpch.lineitem l
    JOIN tpch.orders o ON l.l_orderkey = o.o_orderkey
    JOIN tpch.customer c ON c.c_custkey = o.o_custkey
    JOIN tpch.nation n ON c.c_nationkey = n.n_nationkey
    JOIN tpch.region r ON n.n_regionkey = r.r_regionkey
GROUP BY ROLLUP(r.r_name, n.n_name)
ORDER BY 1,2;
```

-- 31 Tuples

b) Wie hoch ist der Durchschnittspreis von Waren (*part*.*retailprice*) einer bestimmten Marke (*part*.*brand*) bei unterschiedlichen Lieferanten (*supplier*) aus den USA, deren Kontoguthaben größer als 9950 ist? Ordnen Sie das Ergebnis aufsteigend nach Marke, dann nach Durchschnittspreis. In dem Ergebnis sollen sowohl die

Durchschnittspreise pro Marke und Lieferant, als auch pro Marke, als auch pro Lieferant enthalten sein.

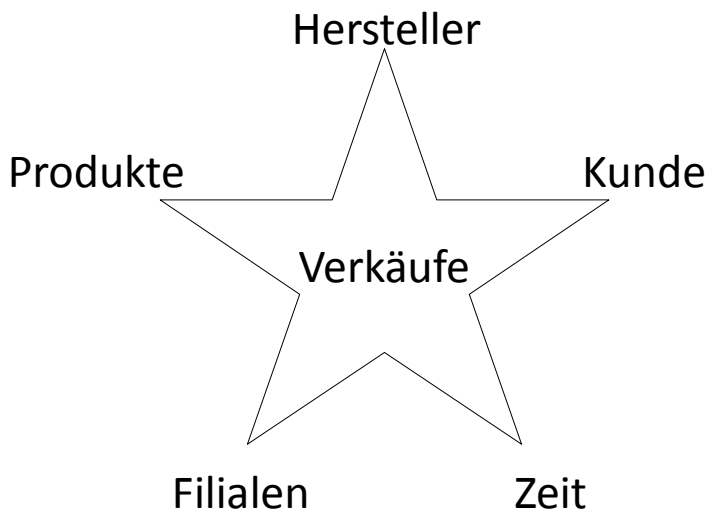
```
SELECT p.p_brand, s.s_name, avg(p.p_retailprice)
FROM tpch.part p
    JOIN tpch.partsupp ps ON p.p_partkey = ps.ps_partkey
    JOIN tpch.supplier s on ps.ps_suppkey = s.s_suppkey
    JOIN tpch.nation n ON s.s_nationkey = n.n_nationkey
WHERE n.n_name = 'UNITED STATES' and s.s_acctbal > 9950
GROUP BY cube (p.p_brand, s.s_name)
ORDER BY 1,3;
```

-- 529 tuples

Aufgabe 4 (Data-Warehouse) *Zusatzaufgabe*

a) In der ER-Modellierung eines Sternschemas modelliert man die Faktentabelle oft als Beziehung. Erläutern Sie das unter Verwendung eines Beispiels.

Die Faktentabelle setzt oft die stellt oft eine n-näre Relation zwischen den Dimensionstabellen her. Daher wird sie als Beziehung der Dimensionen modelliert. Beispiel *Verkäufe* (nach Kemper/Eickler): Ein einzelner Verkauft stellt die Verbindung zwischen was, wann, wo, von wem, gekauft wurde, also Produkte, Zeit, Filialen, Kunde und Hersteller.



b) Entwerfen Sie einen Auswertungsalgorithmus für den ROLLUP-Operator, der sich dadurch auszeichnet, dass bereits berechnete Aggregationen wiederverwendet werden, statt sie neu zu berechnen.

Um Neuberechnen zu verhindern, kann man eine Tabelle vorberechnen, die bereits aggregierte Teilergebnisse enthält, und dann auf diese Tabelle zugreifen und alle Zeilen herausfiltern, die gewünscht sind. Beachten Sie, dass das Ergebnis einer ROLLUP Anfrage eine Teilmenge des Ergebnisses der entsprechenden Anfrage mit Hilfe des CUBE-Operators ist.

Im folgenden wird dies an einem Beispiel verdeutlicht. Gegeben ist das obige Schema. Wir wollen folgende Anfrage effizient ausführen können, die Informationen über Handyverkäufe pro Hersteller und pro Hersteller und Jahr ausgibt:

```
SELECT
  h.name,
  z.jahr,
  SUM(anzahl)
FROM verkäufe v
JOIN hersteller h ON (v.herstellernr = h.id)
JOIN zeit z ON (v.zeit = z.id)
JOIN produkt p ON (v.pid = p.id)
WHERE
  p.type = 'Handy'
GROUP BY ROLLUP (h.name, z.jahr);
```

Wird diese Anfrage häufiger ausgeführt, so kann man diese effizient ausführen, in dem man folgende Tabelle vorberechnet, anstatt die Aggregation jedes Mal neu zuberechnen:

```
CREATE TABLE Handy2DCube
    (name VARCHAR(20), jahr INTEGER, anzahl INTEGER);
```

Folgende Tupel werden nun in die Tabelle eingefügt (Gruppierung nach Hersteller, nach Zeit und nach beidem):

```
INSERT INTO Handy2DCube (
    SELECT h.name, z.jahr, SUM(anzahl)
    FROM verkäufe v
    JOIN hersteller h ON (v.herstellernr = h.id)
    JOIN zeit z ON (v.zeit = z.id)
    JOIN produkt p ON (v.pid = p.id)
    WHERE p.type = 'Handy'
    GROUP BY h.name, z.jahr;
  UNION
    SELECT h.name, NULL, SUM(anzahl)
    FROM verkäufe v
    JOIN hersteller h ON (v.herstellernr = h.id)
    JOIN zeit z ON (v.zeit = z.id)
    JOIN produkt p ON (v.pid = p.id)
    WHERE p.type = 'Handy'
    GROUP BY h.name;
  UNION
    SELECT NULL, z.jahr, SUM(anzahl)
    FROM verkäufe v
    JOIN hersteller h ON (v.herstellernr = h.id)
    JOIN zeit z ON (v.zeit = z.id)
    JOIN produkt p ON (v.pid = p.id)
    WHERE p.type = 'Handy'
    GROUP BY z.jahr;
);
```

Die obige Anfrage kann dann effizient auf der Tabelle Handy2DCube durch einfache Selektion ausgeführt werden:

```
SELECT *
FROM Handy2DCube
WHERE name IS NOT NULL OR (name IS NULL AND jahr IS NULL);
```

D.h., da wir ein ROLLUP verwenden und zuerst nach Namen und dann nach Jahr gruppieren, darf zwar das Jahr `NULL` sein, nicht aber der Name (es sei denn, beides ist `NULL`).

Da es sich um die vollständige CUBE-Tabelle handelt, kann man dieselbe Tabelle jetzt auch verwenden, um die umgekehrte Abfrage zu implementieren, die erst nach Jahr und dann nach Namen gruppiert (`GROUP BY ROLLUP (z.jahr, h.name)`):

```
SELECT *
FROM Handy2DCube
WHERE jahr IS NOT NULL OR (jahr IS NULL AND name IS NULL);
```

Dies ist möglich, weil der gesamte Daten-Würfel in Handy2DCube enthalten ist. Berechnet man nur ROLLUP-Teilmengen vor, so müssen jeweils unterschiedliche Tabellen für Anfragen GROUP BY ROLLUP (a1, a2) bzw. GROUP BY ROLLUP (a1, a2) vorberechnet werden. Daher ist die Würfeldarstellung in diesem Fall günstiger. Somit kann auch die CUBE-Anfrage folgendermaßen gestellt werden:

```
SELECT *
FROM Handy2DCube;
```