TU München, Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme
Prof. Alfons Kemper, Ph.D.

Übung zur Vorlesung *Grundlagen: Datenbanken* im WS20/21

Christoph Anneser, Josef Schmeißer, Moritz Sichert, Lukas Vogel (gdb@in.tum.de)
https://db.in.tum.de/teaching/ws2021/grundlagen/

Blatt Nr. 11

Hausaufgabe 1

Gegeben sei die Anfrage:

```
select *
  from R, S, T
 where R.A = S.A and S.B = T.B and T.C = R.A
```

Des Weiteren soll gelten:

- `S.A` und `T.C` seien Fremdschlüssel auf `R`
- `S.B` sei Fremdschlüssel auf `T`
- `R.A`, `T.B` seien Primärschlüssel von `R` respektive `T`
- Ihre Query-Engine unterstützt nur Nested-Loop-Joins
- Kardinalitäten: $|R| = 100, |S| = 1000, |T| = 10$
- Es gibt keine Indexe

Bestimmen Sie, wie in der Vorlesung gezeigt, den optimalen Ausführungsplan als Baum mit Kosten-/Kardinalitätsabschätzungen mit Hilfe von Dynamischem Programmieren.

Lösung:

Für dieses einfache Beispielsytem, welches nur Nested-Loop-Joins unterstützt, kann als Kostenfunktion C verwendet werden:

$$C(T) = \begin{cases} 0 & \text{falls } T \text{ eine Basisrelation } R_i \text{ ist} \\ |T_1| \cdot |T_2| + C(T_1) + C(T_2) & \text{falls } T = T_1 \bowtie^{\text{NL}} T_2 \end{cases}$$

Da ein Nested-Loop-Joins effektiv jedes Tupel aus dem Kreuzprodukt $T_1 \times T_2$ als Zwischenergebnis betrachtet, sind seine Kosten $|T_1 \times T_2| = |T_1| \cdot |T_2|$. Des weiteren müssen natürlich auch die Kosten von T_1 und T_2 mit einberechnet werden. Die Kardinalitäten können wie gewohnt berechnet werden:

$$|T| = \begin{cases} |R_i| & \text{falls } T \text{ eine Basisrelation } R_i \text{ ist} \\ (\prod_{R_i \in T_1, R_j \in T_2} f_{i,j}) |T_1||T_2| & \text{falls } T = T_1 \bowtie^{\text{NL}} T_2 \end{cases}$$

Für Equijoins über den Fremdschlüssel gilt die Abschätzung:

$$f_{i,j} = \frac{1}{|R_i|}, \text{mit Fremdschlüssel in } R_j$$

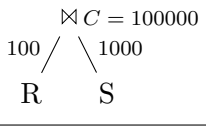
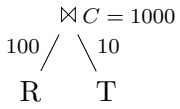
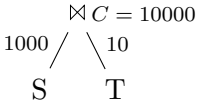
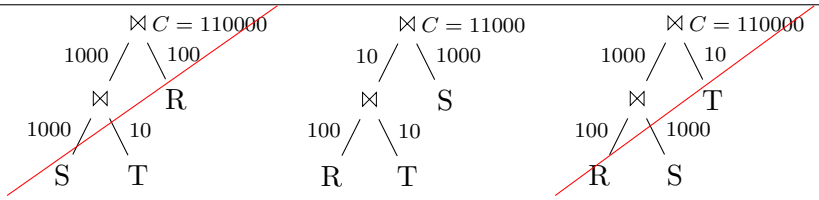
Da alle Relationen über Fremdschlüssel verbunden sind, gelten (vereinfachend) folgende Kardinalitäten:

$$|R \bowtie S| = |S \bowtie R| = \frac{1}{|R|} \cdot |R| \cdot |S| = |S| = 1000$$

$$|R \bowtie T| = |T \bowtie R| = \frac{1}{|R|} \cdot |R| \cdot |T| = |T| = 10$$

$$|S \bowtie T| = |T \bowtie S| = \frac{1}{|T|} \cdot |S| \cdot |T| = |S| = 1000$$

Für die Berechnung der Kosten ergibt sich dann folgende DP-Tabelle:

DP-Tabelle		
Index	Pläne	Kosten
R	R	0
S	S	0
T	T	0
R,S		100000
R,T		1000
S,T		10000
R,S,T		11000

Hausaufgabe 2

Wofür stehen die vier Buchstaben ACID? Erklären Sie für jeden der vier Konzepte, warum es für eine Datenbank wichtig ist. Geben Sie dazu jeweils ein Beispiel an, was passieren könnte, wenn dieses Konzept nicht gelten würde.

Lösung:

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

Wenn eine Datenbank keine Atomarität garantieren kann, kann es zu inkonsistenten Daten kommen (unabhängig von der Konsistenzeigenschaft/Consistency). Hierzu dient eine Überweisung in einer Bank als Beispiel: Wenn eine Transaktion daraus besteht, einen Kontostand zu verringern und einen andern zu erhöhen, entsteht ein inkonsistenter Zustand, wenn nur eine der beiden Operationen tatsächlich gespeichert wird.

Bei einer Datenbank, die nicht konsistent ist, können weitreichende Probleme entstehen. Wenn z.B. garantiert werden muss, dass Kontonummern eindeutig sind, die Datenbank aber inkonsistente Daten zulässt, kann nicht mehr garantiert werden, dass Überweisungen tatsächlich beim richtigen Kontobesitzer ankommen.

Bei dem Beispiel einer Überweisung kann es auch zu Problemen kommen, wenn die Datenbank Transaktionen nicht korrekt voneinander isoliert. Zwei parallele Überweisungen können gleichzeitig Kontostände ändern, was zu inkorrekten Kontoständen nach der Ausführung beider Transaktionen führen kann.

Wenn eine Datenbank eingesetzt wird, die keine Dauerhaftigkeit garantieren kann, kann nie darauf vertraut werden, dass ein commit eine Transaktion tatsächlich festschreibt. Das ist aber z.B. bei einem Geldautomaten notwendig, der erst Geld ausgeben sollte, sobald die Datenbank garantieren kann, dass die Abhebetransaktion verbucht ist.

Hausaufgabe 3

Sie verwenden ein Datenbanksystem mit Write-Ahead-Logging und der Strategie $\neg$*force* und *steal*. Die Datenbank verwaltet lediglich zwei Datenobjekte, X mit dem Anfangswert 10 und Y mit dem Anfangswert 100.

Sie starten die 3 Transaktionen T_1, T_2 und T_3 zum gleichen Zeitpunkt:

T_1	T_2	T_3
BOT	**BOT**	**BOT**
$r(X, x_1)$	$r(Y, y_2)$	$r(X, x_3)$
$x_1 := x_1 + 1$	$r(X, x_2)$	$x_3 := x_3 \cdot 10$
$w(X, x_1)$	$y_2 := y_2 \cdot 2$	$w(X, x_3)$
COMMIT	$x_2 := x_2 + 5$	**COMMIT**
	$w(Y, y_2)$	
	$w(X, x_2)$	
	COMMIT	

Während der Ausführung stürzt Ihre Datenbank ab. Sie wissen nicht, ob - und wenn ja, welche - Transaktionen festgeschrieben wurden. Sie wissen nur, dass die Datenbank ausschließlich *serielle Historien* erzeugt, also dass Transaktionen immer atomar ausgeführt werden und somit keine Verzahnung möglich ist. Bevor Sie die Datenbank neu starten, durchsuchen Sie die Festplatte und stellen fest, dass Y dort den Wert 200 hat. Nachdem die Datenbank neu gestartet wurde und der Recovery-Prozess abgeschlossen ist, liefert sie für X den Wert 110.

Sie wollen nun dem Fehler auf den Grund gehen:

a) Finden Sie zunächst anhand der Zwischenwerte für X und Y heraus, welche Transaktionen *winner* sind, und welche *loser*.

b) Geben Sie das Log an, wie es zum Zeitpunkt des Absturzes auf der Platte stand (verwenden Sie logische Protokollierung).

c) Geben Sie das Log nach Beendigung des Recovery-Prozesses an.

Lösung:

a) Zum Zeitpunkt des Absturzes hatte Y auf der Festplatte den Wert 200. Da Y zu Beginn den Wert 100 hatte, muss Transaktion T_2 zu diesem Zeipunkt schon gestartet und mindestens bis zur Aktion $w(Y, y_2)$ ausgeführt worden sein.

   Betrachten wir nun den Wert von X nach Abschluss der Recovery. Aus $X = 110$ folgt, dass zuerst T_1 (X hat nun den Zwischenwert 11) und dann T_3 (X hat nun

den Endwert 110) ausgeführt wurde, T_2 aber nicht ausgeführt wurde! T_2 muss also eine Losertransaktion sein, welche während des Recovery-Prozesses wieder rückgängig gemacht wurde.

Das Datenbanksystem hat die Transaktionen also in der logischen Reihenfolge T_1, T_3, T_2 ausgeführt, wobei T_1 und T_3 *winner* sind und T_2 *loser* ist.

b) Hier ein mögliches Log. Bitte beachten: auf die Festplatte wird nur das Log selbst (also die „Log"-Spalte) geschrieben.

Schritt	T_1	T_2	T_3	Log
1.	**BOT**			$[\#1, T_1, \textbf{BOT}, 0]$
2.	$r(X, x_1)$			
3.	$x_1 := x_1 + 1$			
4.	$w(X, x_1)$			$[\#2, T_1, P_X, X \mathrel{+}= 1, X \mathrel{-}= 1, \#1]$
5.	**commit**			$[\#3, T_1, \textbf{commit}, \#2]$
6.			**BOT**	$[\#4, T_3, \textbf{BOT}, 0]$
7.			$r(X, x_3)$	
8.			$x_3 := x_3 \cdot 10$	
9.			$w(x, x_3)$	$[\#5, T_3, P_X, X \mathrel{\cdot}= 10, X \mathrel{/}= 10, \#4]$
10.			**commit**	$[\#6, T_3, \textbf{commit}, \#5]$
11.		**BOT**		$[\#7, T_2, \textbf{BOT}, 0]$
12.		$r(Y, y_2)$		
13.		$r(X, x_2)$		
14.		$y_2 := y_2 \cdot 2$		
15.		$x_2 := x_2 + 5$		
16.		$w(Y, y_2)$		$[\#8, T_2, P_Y, Y \mathrel{\cdot}= 2, Y \mathrel{/}= 2, \#7]$
17.		$w(X, x_2)$		$[\#9, T_2, P_X, X \mathrel{+}= 5, X \mathrel{-}= 5, \#8]$

Ob Schritt 17 tatsächlich ausgeführt wurde, können wir nicht wissen. Der Vollständigkeit zuliebe gehen wir hier vom *worst case* aus. Die Datenbank stürzt direkt vor dem *commit* von T_2 ab.

c) Da T_2 eine *loser*-Transaktion ist, wird die Datenbank während der Recovery in der *Undo-Phase Compensation Log Records* schreiben, um die partiell durchgeführte Transaktion rückgängig zu machen. Das endgültige Log sieht dann wie folgt aus:

$$[\#1, T_1, \textbf{BOT}, 0]$$
$$[\#2, T_1, P_X, X \mathrel{+}= 1, X \mathrel{-}= 1, \#1]$$
$$[\#3, T_1, \textbf{commit}, \#2]$$
$$[\#4, T_3, \textbf{BOT}, 0]$$
$$[\#5, T_3, P_X, X \mathrel{\cdot}= 10, X \mathrel{/}= 10, \#4]$$
$$[\#6, T_3, \textbf{commit}, \#5]$$
$$[\#7, T_2, \textbf{BOT}, 0]$$
$$[\#8, T_2, P_Y, Y \mathrel{\cdot}= 2, Y \mathrel{/}= 2, \#7]$$
$$[\#9, T_2, P_X, X \mathrel{+}= 5, X \mathrel{-}= 5, \#8]$$
$$\langle \#9', T_2, P_X, X \mathrel{-}= 5, \#9, \#8 \rangle$$
$$\langle \#8', T_2, P_Y, Y \mathrel{/}= 2, \#9', \#7 \rangle$$
$$\langle \#7', T_2, -, -, \#8', 0 \rangle$$

Hausaufgabe 4

Formulieren Sie die folgende Anfrage auf dem bekannten Unischema in SQL: Ermitteln Sie für jede Vorlesung, wie viele Studenten diese vorgezogen haben. Ein Student hat eine Vorlesung vorgezogen, wenn er in einem früheren Semester ist als der „Modus“ der Semester der Hörer dieser Vorlesung. Der Modus ist definiert als der Wert, der am häufigsten vorkommt – für diese Anfrage also das Semester, in dem die meisten Hörer dieser Vorlesung sind. Falls es mehrere Semester dieser Art gibt, soll nur das niedrigste zählen.

Beachten Sie, dass auch Vorlesungen ohne Hörer, sowie Vorlesungen deren Hörer alle im gleichen Semester sind, ausgegeben werden sollen.

Geben Sie für jede Vorlesung die Vorlesungsnummer, den Titel und die Anzahl der „Vorzieher“ aus.

Lösung:

Die Aufgabe lässt sich am leichtesten lösen, wenn man sie in mehrere Teile aufbricht:

Zunächst erstellen wir eine Anfrage, welche für jede Vorlesung aufschlüsselt, von wie vielen Stundenten sie pro Semester gehört wird:

```
with vorl_semester_anz as (
    select h.vorlnr, s.semester, count(*) as anzahl
    from hoeren h, Studenten s
    where h.matrnr = s.matrnr
    group by h.vorlnr, s.semester
)
```

Mithilfe dieser Sicht können wir nun für jede Vorlesung den Modus der Hörersemester bestimmen. Der Modus ist dasjenige Semester, dem die meisten Anhörer angehören. Unter diesen Einträgen könnten auch Duplikate sein. Wenn eine Vorlesung z.B. gleich oft von Erst- und Drittsemestern gehört wird, wollen wir sie dem ersten Semester zuordnen. Mit einem einfachen *group by* finden wir das Minimum.

```
with vorl_modus as (
    select v1.vorlnr, min(v1.semester) as modus
    from vorl_semester_anz v1
    where v1.anzahl = (
        select max(v2.anzahl)
        from vorl_semester_anz v2
        where v1.vorlnr = v2.vorlnr
    )
    group by v1.vorlnr
)
```

Nun müssen wir nur noch für jedes dieser Vorlesung-Semester-Paare alle diejenigen Studenten finden und aufsummieren, welche die Vorlesung hören und in einem niedrigeren Semester als der Modus sind. Da wir auch Vorlesungen ohne Hörer ausgeben wollen, müssen wir den *left outer join* verwenden. Wichtig: Die Bedingung `s.semester < vm.semester` muss als Bedingung des outer joins angegeben werden, und *nicht* in der where-Klausel, da ansonsten alle Vorlesungen ohne Studenten aus niedrigeren Semestern wieder herausgefiltert werden würden.

```
select v.vorlnr, v.titel, count(s.matrnr) as anzahl
from
```

```
        vorlesungen v left outer join
        vorl_modus vm on v.vorlnr = vm.vorlnr left outer join
        hoeren h on h.vorlnr = v.vorlnr left outer join
        studenten s on s.matrnr = h.matrnr and s.semester < vm.modus
    group by v.vorlnr, v.titel
```

Insgesamt ergibt sich also beispielsweise folgende Anfrage:

```
with vorl_semester_anz as (
    select h.vorlnr, s.semester, count(*) as anzahl
    from hoeren h, Studenten s
    where h.matrnr = s.matrnr
    group by h.vorlnr, s.semester
), vorl_modus as (
    select v1.vorlnr, min(v1.semester) as modus
    from vorl_semester_anz v1
    where v1.anzahl = (
        select max(v2.anzahl)
        from vorl_semester_anz v2
        where v1.vorlnr = v2.vorlnr
    )
    group by v1.vorlnr
)

select v.vorlnr, v.titel, count(s.matrnr) as anzahl
from
    vorlesungen v left outer join
    vorl_modus vm on v.vorlnr = vm.vorlnr left outer join
    hoeren h on h.vorlnr = v.vorlnr left outer join
    studenten s on s.matrnr = h.matrnr and s.semester < vm.modus
group by v.vorlnr, v.titel
```