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

Übung zur Vorlesung *Einsatz und Realisierung von Datenbanken* im SoSe22

Alice Rey, Maximilian {Bandle, Schüle}, Michael Jungmair (i3erdb@in.tum.de)
http://db.in.tum.de/teaching/ss22/impldb/

Blatt Nr. 02

Hinweise Beliebige Historien können auf `https://transactions.db.in.tum.de/` auf ihre Eigenschaften hin getestet werden. Dort können Sie Ihre eigene Einschätzung überprüfen und sehen, ob Sie richtig liegen.

Hausaufgabe (wird nicht in der Übung besprochen)

Wir definieren $r_i(A)$ als das Lesen, $w_1(A)$ als das Schreiben des Datenobjektes A durch Transaktion T_i, sowie a_i als **abort** und c_i als **commit** der Transaktion T_i. Die verzahnte Ausführung mehrerer Transaktionen bezeichnen wir als *Historie*. Geben Sie mögliche Konfliktoperationen bezüglich eines Datenobjektes A an!

Lösung: Konflikte treten auf, sobald mindestens eine von mehreren Transaktionen dasselbe Datum A schreiben. Für diese Konfliktoperationen ist die Reihenfolge wichtig und die partielle Ordnung $<_H$ muss definiert sein. In Konflikt zueinander stehen bei zwei Transaktionen T_i und T_j:

- $r_i(A)$ und $w_j(A)$
- $w_i(A)$ und $r_j(A)$
- $w_i(A)$ und $w_j(A)$

Geben Sie außerdem an, wann eine Transaktion T_i von einer Transaktion T_j liest.

Lösung: T_i **liest** von T_j, wenn:

- T_j das Datum schreibt bevor T_i es liest: $w_j(A) <_H r_i(A)$,
- T_j nicht zurückgesetzt wird: kein a_j vor $r_i(A)$ und
- alle anderen Transaktion, die das Datum zwischendurch geschrieben haben, zurückgesetzt worden sind.

Schritt	T_i	T_j	T_k
1.		w(A)	
2.		~~a~~	
3.			w(A)
4.			a
5.	r(A)		

Wann ist eine Historie H serialisierbar (SR), rücksetzbar (RC), kaskadierendes Rücksetzen vermeidend (ACA) oder strikt (ST)?

Lösung:

- *SR*: Serialisierbarkeitsgraph $SG(H)$ azyklisch
- *RC*: Wenn T_i liest T_j, dann $c_j <_H c_i$
- *ACA*: Wenn T_i liest T_j bezüglich Datum A, dann $c_j <_H r_i(A)$
- *ST*: Wenn $w_j(A) <_H o_i(A)$, dann $c_j <_H o_i(A)$ oder $a_j <_H o_i(A)$

Hausaufgabe 1

Die Historie H für die Transaktionen T_1, T_2 und T_3 sei durch das folgende Diagramm gegeben:

$$H = \begin{array}{ccccccccc} & & & & r_1(x) & \to & r_1(y) & \to & w_1(x) & \to & c_1 \\ & & & & \uparrow & & & \nwarrow & & \downarrow & \\ & r_2(y) & \to & r_2(x) & \to & w_2(z) & & \to & w_2(y) & \to & c_2 \\ & \downarrow & \nearrow & \uparrow & \nwarrow & & & & & & \\ & w_3(x) & \to & r_3(y) & \to & r_3(z) & & \longrightarrow & c_3 & & \end{array}$$

a) Geben Sie alle Konfliktoperationen von H an.

b) Geben Sie eine total geordnete Historie H' an (also eine „lineare“ Abfolge von Operationen), die konfliktäquivalent zu H ist.

c) Geben Sie an, welche Transaktionen voneinander lesen.

d) Geben Sie den Serialisierbarkeitsgraphen von H an.

e) Geben Sie eine serielle Historie H'' an, die konfliktäquivalent zu H ist.

Lösung:

a) Geben Sie alle Konfliktoperationen von H an.

$r_2(x), w_1(x)$
$w_3(x), r_2(x)$
$w_3(x), r_1(x)$
$w_3(x), w_1(x)$
$w_2(y), r_1(y)$
$r_3(y), w_2(y)$
$r_3(z), w_2(z)$

Die Operationen $r_1(x), w_1(x)$ und $r_2(y), w_2(y)$ werden nicht als Konfliktoperationen bezeichnet, da sie jeweils der selben Transaktion angehören.

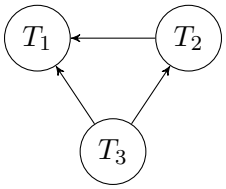
b) Geben Sie eine total geordnete Historie H' an (also eine „lineare“ Abfolge von Operationen), die konfliktäquivalent zu H ist.

Beispielsweise: $r_2(y), w_3(x), r_3(y), r_3(z), c_3, r_2(x), r_1(x), w_2(z), w_2(y), r_1(y), w_1(x), c_1, c_2$

c) Geben Sie an, welche Transaktionen voneinander lesen.

T_1 liest von T_3 und T_2, T_2 liest von T_3.

d) Geben Sie den Serialisierbarkeitsgraphen von H an.



e) Geben Sie eine serielle Historie H'' an, die konfliktäquivalent zu H ist.

H'' muss die Transaktionen in der Reihenfolge T_3, T_2, T_1 enthalten:

$H'' = T_3|T_2|T_1 = w_3(x), r_3(y), r_3(z), c_3, r_2(y), r_2(x), w_2(z), w_2(y), c_2, r_1(x), r_1(y), w_1(x), c_1$

Hausaufgabe 2

a) Geben Sie alle Eigenschaften an, die von der Historie erfüllt werden.

$$w_1(x), r_2(y), w_3(y), w_2(x), w_3(z), c_3, w_1(z), c_2, c_1$$

richtig	**falsch**	**Aussage**
	✔	Serialisierbar (SR)
✔		Rücksetzbar (RC)
✔		Vermeidet kaskadierendes Zurücksetzen (ACA)
	✔	Strikt (ST)

b) Geben Sie alle Eigenschaften an, die von der Historie erfüllt werden.

$$r_1(x), r_1(y), w_2(x), w_3(y), r_3(x), a_1, r_2(x), r_2(y), c_2, c_3$$

richtig	**falsch**	**Aussage**
	✔	Serialisierbar (SR)
	✔	Rücksetzbar (RC)
	✔	Vermeidet kaskadierendes Zurücksetzen (ACA)
	✔	Strikt (ST)

c) Gegeben die unvollständige Historie:

$$H = w_1(x), w_1(y), r_2(x), r_2(y)$$

1) Fügen Sie `commits` in H so ein, dass die Historie RC aber nicht ACA erfüllt.

$$w_1(x), w_1(y), r_2(x), r_2(y), c_1, c_2$$

2) Fügen Sie `commits` in das ursprüngliche H so ein, dass die Historie ACA erfüllt.

$$w_1(x), w_1(y), c_1, r_2(x), r_2(y), c_2$$

Hausaufgabe 3

a) Erläutern Sie kurz die zwei Phasen des 2PL-Protokolls.

b) Inwiefern unterscheidet sich das *strenge* 2PL?

c) Welche Eigenschaften (SR,RC,ACA,ST) haben Historien, welche vom 2PL und vom strengen 2PL zugelassen werden?

d) Wäre es beim strengen 2PL-Protokoll ausreichend, alle Schreibsperren bis zum EOT (Transaktionsende) zu halten, aber Lesesperren schon früher wieder freizugeben?

Lösung:

a) Jede Transaktion durchläuft zwei Phasen:

- Eine *Wachstumsphase*, in der sie Sperren anfordern, aber keine freigeben darf und
- eine *Schrumpfungsphase*, in der sie Sperren freigibt, jedoch keine neuen Sperren anfordern darf.

b) Alle Sperren werden bis zum Ende der Transaktion gehalten und gemeinsam freigegeben. Die Schrumpfungsphase entfällt somit.

c) 2PL garantiert Historien aus SR. Das strenge 2PL garantiert Historien aus SR $\cap$ ST.

d) Es ist ausreichend, beim strengen 2PL-Protokoll nur die Schreibsperren bis zum Ende der Transaktion zu halten. Lesesperren können analog zum normalen 2PL-Protokoll in der Schrumpfungsphase (nach wie vor jedoch nicht in der Wachstumsphase) peu à peu freigegeben werden. Die generierten Schedules bleiben serialisierbar und strikt.

Begründung

- Schon das normale 2PL bietet Serialisierbarkeit; diese ist also auch hier gegeben.
- Das Halten der Schreibsperren bis zum Ende der Transaktion stellt sicher, dass keine Transaktion von einer anderen lesen oder einen von ihr modifizierten Wert überschreiben kann, bevor diese nicht ihr **commit** durchgeführt hat.

  Es gilt:

$$\forall T_i : \forall T_j : (i \neq j)\ \forall A : (w_i(A) <_H r_j(A)) \vee (w_i(A) <_H w_j(A)) \Rightarrow (c_i <_H r_j(A)) \text{ bzw. } (c_i <_H w_j(A))$$

Hausaufgabe 4

SQL-92 spezifiziert mehrere Konsistenzstufen (*isolation level*) durch welche der Benutzer (bzw. die Anwendung) festlegen kann, wie "stark" eine Transaktion von anderen parallel laufenden Transaktionen isoliert werden soll.

a) Erläutern Sie kurz die Isolation Level. Geben Sie an, welche Nebenläufigkeitsprobleme mit dem jeweiligen Level vermieden werden. Füllen Sie folgende Tabelle aus, die zeigt, welche Probleme durch die jeweiligen Isolation Level verhindert (✓) werden:

Lösung: Nicht abgesicherte Nebenläufigkeit kann zu folgenden Problemen führen:

- lost update
- dirty read
- non-repeatable read
- phantom problem

		lost update	dirty read	non-repeatable read	phantom problem
isolation level	read un-commited	✓			
	read commited	✓	✓		
	repeatable read	✓	✓	✓	
	serializable	✓	✓	✓	✓

b) Warum kann zwischen den Konsistenzstufen gewählt werden?

Lösung: Die Isolation Level erlauben einen Kompromiss zwischen Performanz und Genauigkeit zu schließen. Je mehr Genauigkeit/Sicherheit, desto langsamer wird die Ausführung. Zusätzlich kann dem Datenbanksystem mitgeteilt werden, ob die Transaktion *read only* ist. Sind im System ausschließlich read-only-Transaktionen aktiv, dann können diese völlig uneingeschränkt parallel laufen, da Konfliktoperationen ausgeschlossen sind.

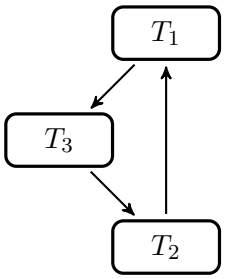
Hausaufgabe 5

Ein inhärentes Problem der sperrbasierten Synchronisationsmethoden ist das Auftreten von Verklemmungen (Deadlocks). Zur Erkennung von Verklemmungen wurde der Wartegraph eingeführt. Dabei wird eine Kante $T_i \rightarrow T$ eingefügt, wenn T_i auf die Freigabe einer Sperre durch T wartet.

Skizzieren Sie einen Ablauf von Transaktionen, bei dem ein Deadlock auftritt, der einen Zyklus mit einer Länge von mindestens 3 Kanten im Wartegraphen erzeugt.

Schritt	T_1	T_2	T_3	Bemerkung
1.	BOT			
2.		BOT		
3.			BOT	
4.	lockX(A)			
5.		lockX(B)		
6.			lockX(C)	
7.	write(A)			
8.		write(B)		
9.			write(C)	
10.	lockS(C)			Will C lesen.
11.		lockS(A)		Will A lesen.
12.			lockS(B)	Will B lesen.

Der Wartegraph sieht dann wie folgt aus:



Gruppenaufgabe 6

Gegeben die Relation „Aerzte“, die den Bereitschaftsstatus von Ärzten modelliert

Name	Vorname	...	Bereit
House	Gregory	...	ja
Green	Mark	...	nein
Brinkmann	Klaus	...	ja

sowie die folgende Transaktion in Pseudocode:

```
dienstende(arzt_name)
  select count(*) into anzahl_bereit from aerzte where bereit='ja'
  if anzahl_bereit > 1 then
    update aerzte set bereit='nein' where name=arzt_name
```

Die Transaktion soll sicherstellen, dass immer mindestens ein Arzt bereit ist.

Betrachten Sie einen Ablauf, bei dem zwei zur Zeit bereite Ärzte zum gleichen Zeitpunkt entscheiden, ihren Status auf „nein “, d.h. nicht bereit zu ändern:

T_1: execute dienstende('House')
T_2: execute dienstende('Brinkmann')

Gehen Sie beispielsweise davon aus, dass das DBMS versucht, die Transaktion jeweils abwechselnd zeilenweise abzuarbeiten.

Diskutieren Sie:

a) Was kann bei Snapshot Isolation passieren?

b) Warum ist dies bei optimistischer Synchronisation nicht möglich?

c) Wie verhält sich die Zeitstempel-basierte Synchronisation?

d) Wie verhält sich das strenge 2PL?

Lösung:

a) **Snapshot Isolation:** Hier wird defakto die Standardanomalie von Snapshot Isolation gezeigt. Es ist ein Constraint für die Ausprägung der Datenbank gegeben (hier: Es muss immer mindestens ein Arzt bereit sein; ein anderes traditionelles Beispiel wäre die Summe des Geldes auf der Welt ist konstant oder ähnliches), jedoch kann dieser bei Snapshot Isolation verletzt werden.

Im konkreten Fall wird lediglich geprüft, ob sich die *WriteSets* der parallel laufenden Transaktionen überlappen. Dies ist nicht der Fall, weswegen beide Transaktionen bei Snapshot Isolation erfolgreich sind.

b) **Optimistische Synchronisation:** Bei der (klassischen) optimistischen Synchronisation kann diese Anomalie hingegen nicht auftreten. Hier wird in der Validierungsphase geprüft, ob sich das *ReadSet* mit dem *WriteSet* einer anderen Transaktion überlappt. D.h. das System würde bemerken, dass die Transaktion Daten gelesen und verarbeitet hat, die sich inzwischen geändert haben, was zu einem Transaktionsabbruch führt.

Im konkreten Fall wären alle Tupel der Relation "Ärzte" im *ReadSet* von T_1 sowie von T_2 enthalten. Das $WriteSet(T_1) = \{$[House, ...]$\}$ und $WriteSet(T_2) = \{$[Brinkmann, ...]$\}$. Die *Read-* und *WriteSets* der beiden parallel laufenden Transaktionen sind nicht disjunkt:

$$WriteSet(T_1) \cap ReadSet(T_2) \neq \emptyset$$

$$WriteSet(T_2) \cap ReadSet(T_1) \neq \emptyset$$

In diesem Fall "gewinnt" also die Transaktion, welche die Validierungsphase zuerst erreicht.

Anmerkung: In der Praxis sind die *WriteSets* sehr viel kleiner als die *ReadSets*. Die Validierung ist bei Snapshot Isolation also mit deutlich geringerem Aufwand verbunden.

Lösung:

c) **Zeitstempelbasierte Synchronisation**: Bei zeitstempelbasierter Synchronisation erhält jede Transaktion zu Beginn einen eindeutigen (streng) monoton steigenden Zeitstempel und jedes Datum hat einen Lese- sowie einen Schreib-zeitstempel ($readTS$ u. $writeTS$). Beim Zugriff auf ein Datum wird wie folgt verfahren:

- Wenn Transaktion T ein Datum A lesen möchte:
  - falls $TS(T) < writeTS(A)$, dann wurde A bereits von einer jüngeren Transaktion überschrieben, deshalb muss T zurückgesetzt werden.
  - andernfalls, wenn $TS(T) \geq writeTS(A)$, kann A gelesen werden und es wird gesetzt: $readTS(A) := max(TS(T), readTS(A))$.
- Wenn T ein Datum A schreiben möchte:

- falls $TS(T) < readTS(A)$, dann wurde A bereits von einer jüngeren Transaktion gelesen. Der zu schreibende Wert kann von der jüngeren Transaktion nicht mehr berücksichtigt werden. Deshalb muss T zurückgesetzt werden.
- falls $TS(T) < writeTS(A)$, dann wurde A von einer jüngeren Transaktion geschrieben. Die ältere TA würde den Wert der jüngeren überschreiben, was nicht zulässig ist. T wird zurückgesetzt.
- andernfalls darf T schreiben. Dabei wird der $writeTS(A) := TS(T)$ gesetzt.

Angenommen $TS(T_1) = 1$ und $TS(T_2) = 2$, dann haben im obigen Beispiel alle Tupel einen $readTS = 2$ nachdem die Ärzte im Status 'bereit' gezählt wurden. Möchte dann T_1 das Tupel [House,...] ändern, wird T_1 zurückgesetzt, da $TS(T_1) < readTS([House, ...])$. Nur T_2 kommt zum Abschluss.

d) **2PL**: Die Anomalie kann nicht auftreten. Bei abwechselnder zeilenweiser Ausführung würden in diesem Fall beide Transaktionen zunächst Shared Locks auf alle Bereitschaftsfelder erwerben. Danach würden beide versuchen, das Lock für ihr Bereitschaftsfeld auf ein Exclusive Lock zu eskalieren. Es entsteht ein Deadlock mit Zykluslänge 2. Im Zuge der Deadlock-Behandlung wird dann eine der Transaktionen abgebrochen.