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

Übung zur Vorlesung *Grundlagen: Datenbanken* im WS17/18

Harald Lang, Linnea Passing (gdb@in.tum.de)
http://www-db.in.tum.de/teaching/ws1718/grundlagen/

Blatt Nr. 14

Hausaufgabe 1

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 2

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

$$H_1 = w_1(x), r_2(y), w_3(y), w_2(x), w_3(z), c_3, w_1(z), c_2, c_1$$

richtig	falsch	Aussage
	✔	H_1 ist serialisierbar (SR)
✔		H_1 ist rücksetzbar (RC)
✔		H_1 vermeidet kaskadierendes Zurücksetzen (ACA)
	✔	H_1 ist strikt (ST)

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

$$H_2 = r_1(x), r_1(y), w_2(x), w_3(y), r_3(x), a_1, r_2(x), r_2(y), c_2, c_3$$

richtig	falsch	Aussage
	✔	H_2 ist serialisierbar (SR)
	✔	H_2 ist rücksetzbar (RC)
	✔	H_2 vermeidet kaskadierendes Zurücksetzen (ACA)
	✔	H_2 ist strikt (ST)

3. Gegeben die unvollständige Historie:

$$H_3 = w_1(x), w_1(y), r_2(x), r_2(y)$$

   a) Fügen Sie `commits` in H_3 so ein, dass die Historie RC aber nicht ACA erfüllt.

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

   b) Fügen Sie `commits` in das ursprüngliche H_3 so ein, dass die Historie ACA erfüllt.

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

Lösung:

Hausaufgabe 3

Bei der sperrbasierten Synchronisation hat jedes Datenobjekt eine zugehörige Sperre. Bevor eine Transaktion zugreifen darf, muss sie eine Sperre anfordern. Dabei unterscheiden wir zwei Sperrmodi: Lese- und Schreibsperre.

a) Erläutern Sie kurz die Unterschiede.

b) Geben Sie deren Verträglichkeiten an (wenn mehrere Transaktionen Sperren auf dem selben Datenobjekt anfordern).

Lösung:

a) Eine Lesesperre (auch S-Sperre, shared lock) für ein Datum wird angefordert bevor eine Transaktion das Datum lesen möchte. Ein Schreibvorgang erfordert eine entsprechende Schreibsperre (auch X-Sperre, exclusive lock).

Mehrere Transaktionen können gleichzeitig eine S-Sperre auf dem selben Datenobjekt besitzen, wohingegen maximal eine Transaktion eine X-Sperre für ein Datum besitzen kann.

b) Verträglichkeitsmatrix (auch Kompatibilitätsmatrix):

	gehaltene Sperre		
angeforderte Sperre	keine	S	X
S	$\surd$	$\surd$	–
X	$\surd$	–	–

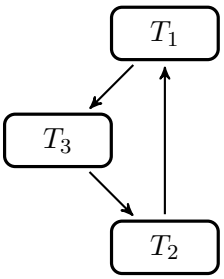
Hausaufgabe 4

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:



Hausaufgabe 5

Bei der Verklemmungsvermeidung durch Zeitstempel wird jeder Transaktion ein eindeutiger Zeitstempel (*time stamp, TS*) zugeordnet. Die TS werden (streng) monoton wachsend vergeben, so dass gilt: $TS(T_{alt}) < TS(T_{jung})$. Eine Sperranforderung auf ein bereits gesperrtes Datenobjekt führt nicht mehr zwangsläufig dazu, dass eine Transaktion in den Wartezustand übergeht. Stattdessen kann es, abhängig vom Alter der Transaktionen, zum Abbruch von einer der betroffenen Transaktionen führen.

a) Erläutern Sie die zwei Strategien *wound-wait* und *wait-die*.

Ein Nachteil der Zeitstempelmethode ist, dass es zu Transaktionsabbrüchen kommen kann, obwohl eine Verklemmung nie aufgetreten wäre.

b) Geben Sie für *wound-wait* und *wait-die* jeweils eine verklemmungsfreie Historie an, die dennoch zu einem Transaktionsabbruch führt.

Lösung:

a)
- *wound-wait*: Wenn eine ältere Transaktion T_{alt} eine Sperre anfordert, welche in Besitz einer jüngeren Transaktion T_{jung} ist, dann wird T_{jung} zugunsten von T_{alt} abgebrochen und zurückgesetzt.

  Fordert hingegen T_{jung} eine Sperre an, die bereits in Besitz von T_{alt} ist, wartet T_{jung} auf deren Freigabe.
- *wait-die*: Wenn T_{alt} eine Sperre anfordert, welche in Besitz von T_{jung} ist, wartet T_{alt} auf die Freigabe.

  Fordert hingegen T_{jung} eine Sperre an, die bereits in Besitz von T_{alt} ist, dann wird T_{jung} abgebrochen und zurückgesetzt.

b)
- *wound-wait*: Die Historie

$$H_1 = r_1(A), r_2(B), r_1(B), w_1(B), r_2(C), c_2, c_1$$

ist serialisierbar und verursacht keinen Deadlock. Dennoch wird die jüngere Transaktion T_2 abgebrochen wenn T_1 eine X-Sperre auf B anfordert:

Schritt	T_1	T_2	Bemerkung
1	lockS(A)		T_1 startet zuerst.
2	r(A)		
3		lockS(B)	
4		r(B)	
5	lockX(B)		T_2 wird bei *wound-wait* abgebrochen
6	...	...	

Anmerkung:

Die beiden Operationen in Schritt 1 und 2 signalisieren lediglich, dass T_1 vor T_2 startet. Das Datum A spielt ansonsten keine Rolle. Eine verzahnte Ausführung, die zu einem Abbruch von T_2 führt, könnte auch wie folgt skizziert werden:

Schritt	T_1	T_2	Bemerkung
1	BOT		T_1 startet zuerst.
2		BOT	T_2 startet als zweite.
3		lockS(B)	
4		r(B)	
5	lockX(B)		T_2 wird bei *wound-wait* abgebrochen
6	...	...	

Die Aufgabenstellung verlangt jedoch die Angabe einer Historie, und für diese sind aber lediglich die Elementaroperationen r, w, a, c definiert, nicht jedoch BOT (*begin of transaction*) und $lockS, lockX, unlock$.

- *wait-die*:

$$H_2 = w_1(A), r_2(A), r_1(B), r_2(C), c_2, c_1$$

Schritt	T_1	T_2	Bemerkung
1	lockX(A)		
2	w(A)		
3		lockS(A)	T_2 wird bei *wait-die* abgebrochen
4	...	...	

Hausaufgabe 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 $WriteSet(T_1) = \{[\text{House}, ...]\}$ und $WriteSet(T_2) = \{[\text{Brinkmann}, ...]\}$. 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.