Dr. rer. nat. Sven Groppe

Übungen zur Vorlesung

Mobile und Verteilte Datenbanken

WS 2009/2010

Blatt 4

Lösung

Aufgabe 1:

Bestimmen Sie zu den folgenden Transaktions-Schedules, ob diese (konflikt-) serialisierbar sind. Falls ja, geben Sie ein äquivalentes serielles Schedule an. Falls nein, begründen Sie dies:

a)

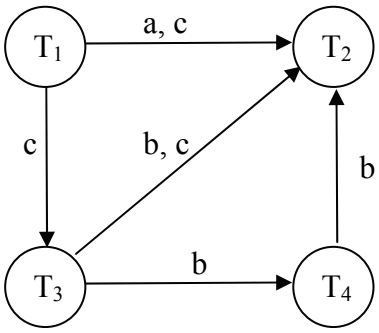
T_1	T_2	T_3	T_4
		read(b)	
		write(b)	
			write(b)
	read(b)		
read(a)			
read(c)			
write(a)			
write(c)			
		read(a)	
		write(c)	
	read(a)		
	write(c)		

b)

T_1	T_2	T_3
read(a)		
read(b)		
write(a)		
		read(a)
	read(b)	
		write(c)
	read(c)	
	write(b)	
	read(a)	
		write(a)
	write(c)	
	write(a)	

Lösung:

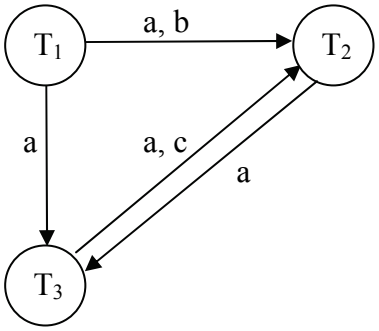
Der Abhängigkeitsgraph des Schedules a) sieht wie folgt aus:



Das Schedule a) ist serialisierbar, weil der Abhängigkeitsgraph keinen Zyklus enthält.

Das Schedule a) ist äquivalent dem seriellen Schedule $T_1 \rightarrow T_3 \rightarrow T_4 \rightarrow T_2$.

Der Abhängigkeitsgraph des Schedules b) sieht wie folgt aus:



Das Schedule b) ist nicht serialisierbar, weil der Abhängigkeitsgraph den Zyklus $T_2 \rightarrow T_3 \rightarrow T_2$ enthält.

Aufgabe 2:

Sowohl die Methodik der Validierung als auch die Methodik der Benutzung von Locks verhindern teilweise Parallelität, die nach dem Konzept der Serialisierbarkeit möglich wäre. Zeigen Sie an einem möglichst einfachen Beispiel, dass es parallele Abläufe von Transaktionen gibt, bei denen die 2-Phasigkeit unnötig Parallelität verhindert. Anders formuliert: Schreiben Sie einen möglichst einfachen Beispielablauf zweier 2-phasiger Transaktionen auf, bei dem eine Transaktion (unnötig) auf die andere wartet, weil die andere Transaktion 2-phasig sperrt, der aber auch serialisierbar wäre, wenn die andere Transaktion beim Sperren nicht 2-phasig vorginge.

Lösung:

Das folgende Schedule ist serialisierbar ($T_1 \rightarrow T_2$), beim 2-phasigen Sperren muss aber die Transaktion T_1 (mit ihrem lock(a)) auf unlock(a) der Transaktion T_2 warten:

T_1	T_2
	read(a)
read(a)	
	write(a)
read(b)	
	write(b)

Aufgabe 3:

Angenommen eine Transaktion T_1 macht zuerst ein *read(x)*, dann ein *read(y)*, dann ein *write(x)* und schließlich ein *write(y)*. Transaktion T_2 macht ein *read(y)* und dann ein *write(x)*. Geben Sie parallele Abläufe der beiden Transaktionen an, die serialisierbar sind und

a) wiederherstellbar (recoverable) aber nicht kaskadenfrei (cascadeless)

Lösung:

T1	**T2**
read(x)	
read(y)	
write(x)	
write(y)	
	read(y)
	write(x)
abort	

Die Transaktion **T2 macht ein dirty read** (fett in dem obigen Plan), d.h., T2 liest Daten von T1 vor T1 bestätigt (committed) worden ist. ⇒ **Der Plan ist nicht kaskadenfrei.**

Der Plan ist wiederherstellbar, weil in dem Plan **nur T2 von T1 liest und T1 als erste commit/abort ausführt.**

Der Plan ist serialisierbar (äquivalent dem sequentiellen Plan T1→T2).

b) kaskadenfrei aber nicht strikt

Lösung:

T1	**T2**
read(x)	

read(y)	
write(x)	
	read(y)
	write(x)
write(y)	
abort	

Der Plan ist kaskadenfrei, weil **keine Transaktion ein dirty read macht** (d.h., keine der zwei Transaktionen unbestätigte Daten liest).

Der Plan ist nicht strikt, weil die Transaktion **T2 ein dirty write macht** (d.h., unbestätigte Daten einer anderen Transaktion überschreibt).

Der Plan ist serialisierbar, **äquivalent dem seriellen Plan T2** (dank dem *abort* der Transaktion T1 muss in dem Konflikt-Graph nur die Transaktion T2 betrachtet werden).

c) strikt

Lösung:

T1	**T2**
read(x)	
read(y)	
write(x)	
write(y)	
commit	
	read(y)
	write(x)
	commit

Der obige Plan ist ein serieller Plan T1→T2.

Alle serielle Pläne sind auch serialisierbar.

Alle serielle Pläne sind auch strikt.

Aufgabe 4:

Sperrende Transaktionen: um Deadlock-frei zu sperren können die Relationen R_1, R_2, ..., R_n global angeordnet werden und benötigte Sperren in der Reihenfolge dieser Ordnung

angefordert werden. Man könnte durch einen Widerspruchbeweis zeigen, dass diese Anordnung der Betriebsmittel zyklische Wartegraphen verhindert, also Deadlock-Freiheit garantiert.

a) Wenn man zwischen Readlock(A) und Writelock(A) unterscheiden will und Deadlock-freies Sperren erweitern will auf Readlock- und Writelock-Operationen, in welcher Reihenfolge muss jede Transaktion dann ihre Sperranforderungen anordnen, um Deadlocks zu vermeiden?

Lösung:

Man muss die üblichen Sperr-Regel um eine zusätzliche Regel erweitern: *Eine Transaktion darf nie ein Writelock(A) anfordern, nachdem sie Readlock(A) angefordert hat.* Mit anderen Wörtern: *Jede Transaktion darf die Sperren nur in einer **strikt** wachsenden Reihenfolge anfordern (unabhängig davon, ob das Lese- oder Schreibsperren sind)*.

Ohne diese Regel landen die folgenden zwei Transaktionen in Deadlock:

T1	**T2**
Readlock(X)	
	Readlock(X)
Writelock(X)	
	Writelock(X)

b) Ergänzen Sie folgende Transaktion um Readlock- und Writelock-Befehle (Unlock braucht nicht extra eingetragen zu werden):

read(R_2);

read(R_1);

write(R_3);

read(R_4);

write(R_4);

Lösung:

readlock(R_1);

readlock(R_2);

read(R_2);

read(R_1);

writelock(R_3);

write(R_3);

writelock(R_4);

read(R_4);

write(R_4);

Aufgabe 5:

Sowohl die Methodik der Validierung als auch die Methodik der Benutzung von Locks verhindern teilweise Parallelität, die nach dem Konzept der Serialisierbarkeit möglich wäre.
Geben Sie ein möglichst einfaches Beispiel für einen parallelen Ablauf zweier Transaktionen an, der serialisierbar wäre, bei dem aber trotzdem in der Validierungsphase eine Transaktion zurückgesetzt wird.

Lösung:

T1	**T2**
start	
	start
	write(a)
	validierung
	schreiben
read(a)	
validierung	

Bei der Validierung von T1 wird die Transaktion T1 abgebrochen, weil T1 von T2 gelesen hat und T2 ihre Validierung/Schreibphase schon angefangen hat. Der Ablauf ist aber serialisierbar. In diesem konkreten Fall ist also der Abbruch der Transaktion T1 nicht notwendig.