AG Datenbanken und Informationssysteme

Wintersemester 2006 / 2007

Prof. Dr.-Ing. Dr. h. c. Theo Härder
Fachbereich Informatik
Technische Universität Kaiserslautern

http://wwwdvs.informatik.uni-kl.de

8. Übungsblatt

Für die Übung am Donnerstag, **21. Dezember 2006**,
von 15:30 bis 17:00 Uhr in 13/222.

Aufgabe 1: Serialisierbarkeit von Historien

Gegeben sei folgende Historie:

H = <$r_3(c)$, $r_3(b)$, $r_4(b)$, $r_2(a)$, $r_4(a)$, $w_3(b)$, $w_2(a)$, $r_1(a)$, $w_1(a)$, $r_1(b)$, c_1, $r_5(c)$, $r_2(b)$, c_2, $w_5(c)$, c_5, c_3, c_4>

a) Beschreiben Sie allgemein das Ziel der Synchronisation.

b) Welche Probleme können in DB-Systemen auftreten, wenn keine Synchronisation vorgenommen wird. Geben Sie je ein Beispiel an.

c) Zeichnen Sie den Serialisierbarkeitsgraphen SG(H) und stellen Sie dann anhand dessen die Serialisierbarkeit von H fest.

d) Ist H auch kommutativitätsbasiert reduzierbar? Begründen Sie ihre Antwort. Ordnen Sie die Operationen entsprechend der Kommutativitätsregeln um.

e) Ist H rücksetzbar? Wenn ja, dann begründen Sie Ihre Antwort. Ansonsten ordnen Sie die Operationen in H so um, dass die erzeugte Historie rücksetzbar wird. (Die Def. finden Sie ggf. in Ihrem alten GBIS Script)

f) Ermitteln Sie aus SG(H) alle möglichen seriellen Historien H_S^i.

g) Welche Einschränkungen der Klasse CSR kennen Sie? Beschreiben Sie diese mit ihren eigenen Worten.

Lösung:

a) Beschreiben Sie allgemein das Ziel der Synchronisation.

Ziel der Synchronisation ist es, die Anomalien die im Mehrbenutzerbetrieb auftreten können, zu vermeiden und somit viele Transaktionen (die ACID gewährleisten müssen) auf gemeinsamen Daten korrekt und verschränkt auszuführen.

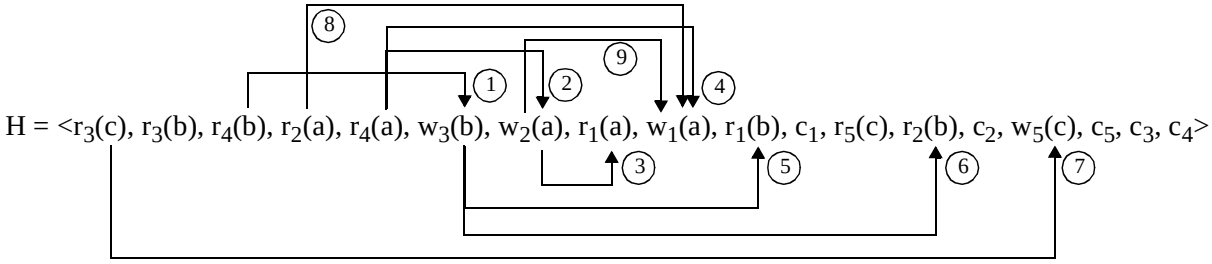
b) Welche Probleme können in DB-Systemen auftreten, wenn keine Synchronisation vorgenommen wird. Geben Sie je ein Beispiel an.

Es können Anomalien auftreten. Hierbei werden meist die folgenden Anomalien voneinander unterschieden:
- Dirty Read
- Lost Udate
- Inconsistent Read, Non-repeatable Read
- Phantom Problem

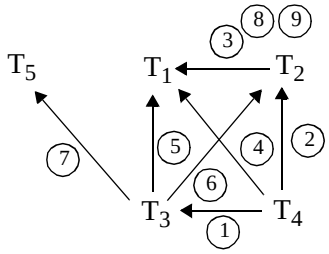
(Beispiele wurden an der Tafel kurz gezeigt - siehe Script)

Dies sind aber noch lang nicht alle Probleme, die durch den Mehrbenutzerbetrieb auftreten können:
- Integritätsverletzung durch Mehrbenutzer-Anomalie
- Instabilität von Cursorn

c) Zeichnen Sie den Serialisierbarkeitsgraphen SG(H) und stellen Sie dann anhand dessen die Serialisierbarkeit von H fest.



SG(H):



=> SG(H) ist azyklisch => H ist serialisierbar

d) Ist H auch kommutativitätsbasiert reduzierbar? Begründen Sie ihre Antwort. Ordnen Sie die Operationen entsprechend der Kommutativitätsregeln um.

Ja, weil jeder serialisierbare Historie auch kommutativitätsbasiert reduzierbar ist.
Die Umformung in einen der seriellen Abläufe:
$H_S^1 = T_4 \mid T_3 \mid T_5 \mid T_2 \mid T_1$
$H_S^2 = T_4 \mid T_3 \mid T_2 \mid T_5 \mid T_1$
$H_S^3 = T_4 \mid T_3 \mid T_2 \mid T_1 \mid T_5$
muss möglich sein.

Beispiel: $H_S^3 = T_4 \mid T_3 \mid T_2 \mid T_1 \mid T_5$
$H = <r_3(c),r_3(b),\mathbf{r_4(b)},r_2(a),r_4(a),w_3(b),w_2(a),r_1(a),w_1(a),r_1(b),c_1,r_5(c),r_2(b),c_2,w_5(c),c_5,c_3,c_4>$
Vertauschung von $\mathbf{r_4(b)}$ und $r_3(b)$ möglich wegen C1,
Vertauschung von $\mathbf{r_4(b)}$ und $r_3(c)$ möglich wegen C1 (siehe Script!)
In den nachfolgenden Schritten sind die entsprechenden Vertauschungen aufgrund der Regeln C1-C4 jeweils möglich, so dass sich ergibt:

$H = <r_4(b),r_3(c),r_3(b),r_2(a),\mathbf{r_4(a)},w_3(b),w_2(a),r_1(a),w_1(a),r_1(b),c_1,r_5(c),r_2(b),c_2,w_5(c),c_5,c_3,c_4>$
$H = <r_4(b),r_4(a),r_3(c),r_3(b),r_2(a),w_3(b),w_2(a),r_1(a),w_1(a),r_1(b),c_1,r_5(c),r_2(b),c_2,w_5(c),c_5,c_3,\mathbf{c_4}>$
$H = <r_4(b),r_4(a),c_4,r_3(c),r_3(b),r_2(a),\mathbf{w_3(b)},w_2(a),r_1(a),w_1(a),r_1(b),c_1,r_5(c),r_2(b),c_2,w_5(c),c_5,c_3>$
$H = <r_4(b),r_4(a),c_4,r_3(c),r_3(b),w_3(b),r_2(a),w_2(a),r_1(a),w_1(a),r_1(b),c_1,r_5(c),r_2(b),c_2,w_5(c),c_5,\mathbf{c_3}>$
$H = <r_4(b),r_4(a),c_4,r_3(c),r_3(b),w_3(b),c_3,r_2(a),w_2(a),r_1(a),w_1(a),r_1(b),c_1,r_5(c),\mathbf{r_2(b),c_2},w_5(c),c_5>$
$H = <r_4(b),r_4(a),c_4,r_3(c),r_3(b),w_3(b),c_3,r_2(a),w_2(a),r_2(b),c_2,r_1(a),w_1(a),r_1(b),c_1,r_5(c),w_5(c),c_5>$
Nun ist H seriell!

e) Ist H rücksetzbar? Wenn ja, dann begründen Sie Ihre Antwort. Ansonsten ordnen Sie die Operationen in H so um, dass die erzeugte Historie rücksetzbar wird. (Die Def. finden Sie ggf. in Ihrem alten GBIS Script)

Rücksetzbarkeit von H:
Def.: Eine Historie H heißt rücksetzbar, falls immer die schreibende TA (T_j) vor der lesenden TA (T_i) ihr Commit ausführt.
Im gegeben Fall ist H nicht rücksetzbar, weil:
- $w_3(b) <_H r_1(b)$ und $c_1 <_H c_3$
- $w_3(b) <_H r_2(b)$ und $c_2 <_H c_3$
- $w_2(a) <_H r_1(a)$ und $c_1 <_H c_2$

Damit H rücksetzbar wird, müssen ihre Commit-Operationen so umgeordnet werden, dass $c_3 <_H c_2 <_H c_1$ gilt. Also setze c_3 vor c_2 und c_1 nach c_2.

f) Ermittlung aller möglichen seriellen Historien H_S^i aus SG(H) mit topologischer Sortierung

Algorithmus für eine topologische Sortierung:
- Eingabe: Ein DAG G = (V, E) als Eingabeparameter
- Ausgabe:
  - Liefert wahr, falls G azyklisch ist, sonst ist G zyklisch.
  - Eine Knotenordnung ord []
- Algorithmus:

```
boolean TopologicalSort (DAG G, Vertex [] ord) {
    int i = 0;
    WHILE (G hat wenigstens einen Knoten v, der keine Eingangskanten hat) {
        ord [i] = v;
```

```
                Entferne Knoten v mit seinen ausgehenden Kanten aus G;
                i++;
        }
        IF (G ist leer bzw. enthält keine Knoten mehr)
                return (true);
        ELSE
                return (false);
}
```

Mögliche H_S^i :

$H_S^1 = T_4 \mid T_3 \mid T_5 \mid T_2 \mid T_1$
$H_S^2 = T_4 \mid T_3 \mid T_2 \mid T_5 \mid T_1$
$H_S^3 = T_4 \mid T_3 \mid T_2 \mid T_1 \mid T_5$

g) Welche Einschränkungen der Klasse CSR kennen Sie? Beschreiben Sie diese mit ihren eigenen Worten.

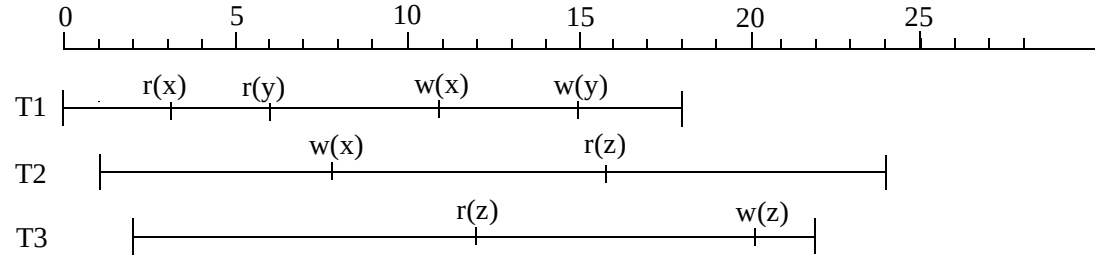
Wir unterscheiden die Klassen OCSR und COCSR:

OCSR: Ordnungserhaltende Konfliktserialisierbarkeit
Eine Transaktion j die komplett vor einer anderen Transaktion i abgeleaufen ist, heißt nur dann ordnungserhaltend konfliktserialisierbar, wenn ein äquivalenter serieller Ablauf existiert, der die Ornung j vor i = T_i vor T_j erhält.

COCSR: Einhaltung der Commit-Reihenfolge
Die Konfliktrelation zwischen den verschiedenen Operationen bestimmt deren Commit-Reihenfolge. Demnach ist eine Historie nur dann in COCSR, wenn ein serieller Ablauf gefunden werden kann, bei dem die Commit-Reihenfolge gleich der durch die Historie vorgegebenen entspricht.

Aufgabe 2: Vergleich verschiedener Scheduling-Algorithmen

Folgendes Bild zeigt den (hypothetischen) Ablauf dreier Transaktionen T1-T3, wenn sie ohne gegenseitige Beeinflussung (also kein Warten auf Sperren etc.) ablaufen könnten. Die zeitlichen Abstände zwischen den einzelnen Operationen **einer** Transaktion ergeben sich aus der "Denk"- oder Rechenzeit der Transaktionen. Diese Abstände verändern sich nicht, auch wenn eine Operation z.B. wegen eines Sperrkonfliktes für eine Weile blockiert wird oder nach einem Abort neu gestartet wird.



Ermitteln Sie basierend auf der so beschriebenen Eingabeschedule die Ausgabe, die folgende Scheduling-Algorithmen liefern würden:

h) SS2PL

i) TO (einfach, ein Zeitstempel)

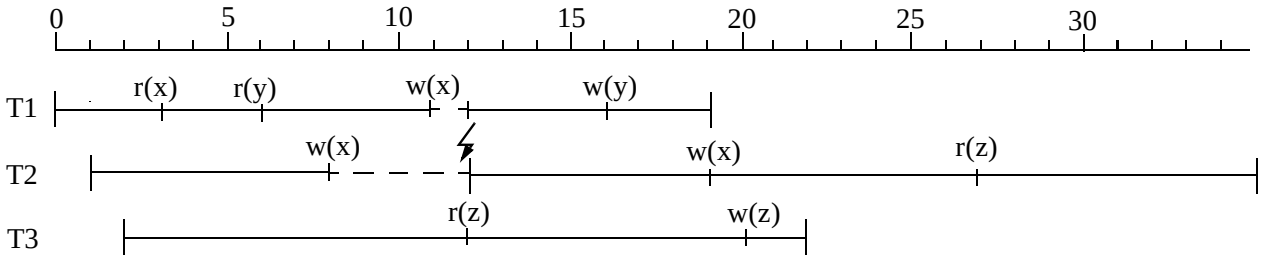
j) BTO (getrennte Lese- und Schreibzeitstempel)

k) SGT

Gehen Sie davon aus, dass beim Sperrverfahren die (erfolgreiche, nicht blockierende) Sperranforderung und die Durchführung der Operation eine Aktion sind. Wird eine Transaktion durch den Scheduler abgebrochen, wird sie direkt neu gestartet.

Lösung:

a) SS2PL

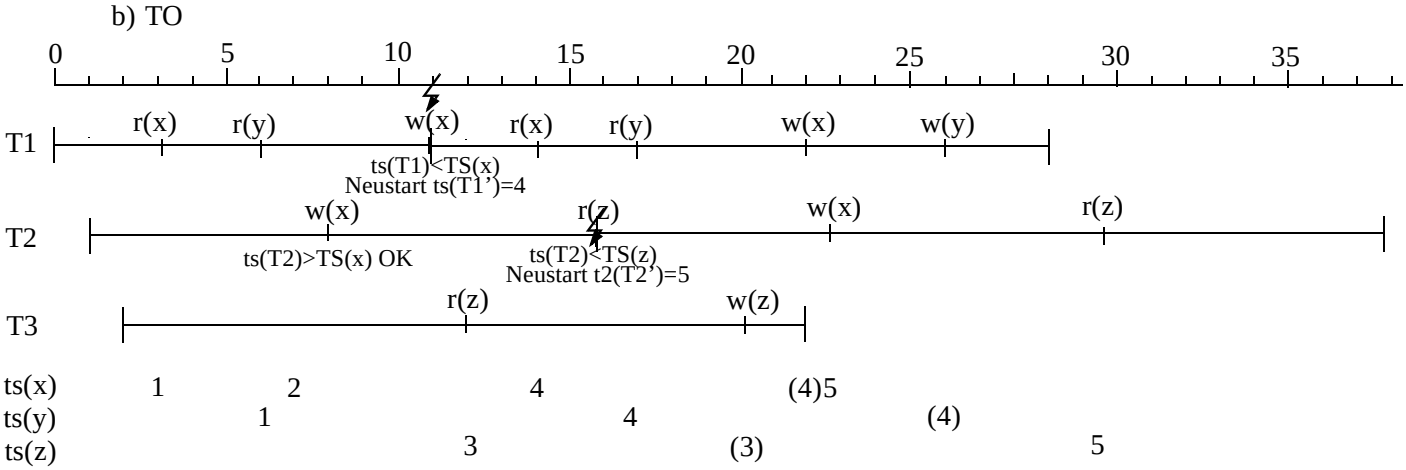


T2 muss auf Freigabe der R-Sperre von T1 warten. T1 fordert X-Sperre, muss auf Freigabe der Sperre von T2 warten => zyklische Wartebeziehung, (Konversions-)Deadlock.

Annahme: Deadlock wird nach einer Zeiteinheit bemerkt, Behebung durch Abort von T2 mit unmittelbar anschließendem Neustart.

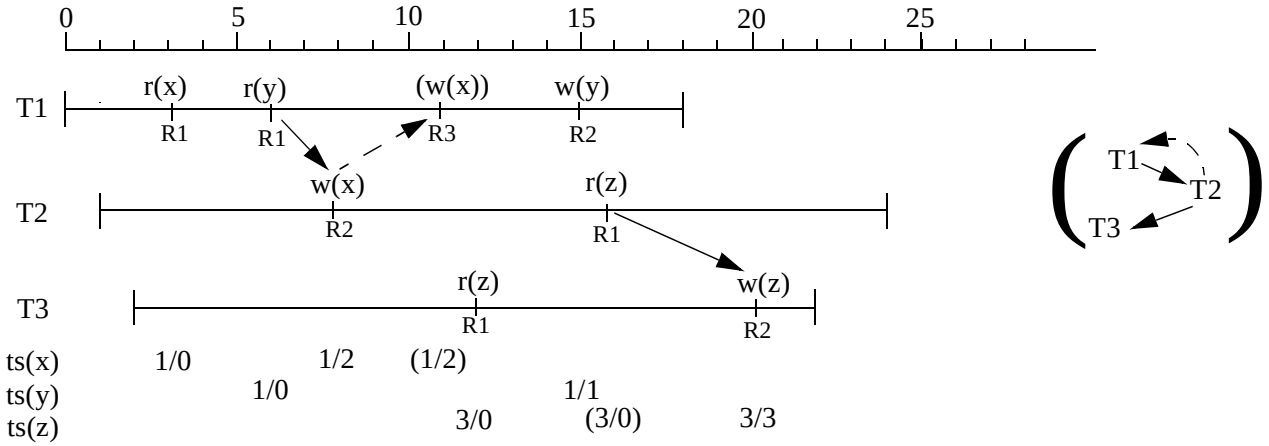
Äquivalente serielle Reihenfolgen: T1, T3, T2 und T3, T1, T2. Gesamtdauer der Schedule: 35 ZE

b) TO



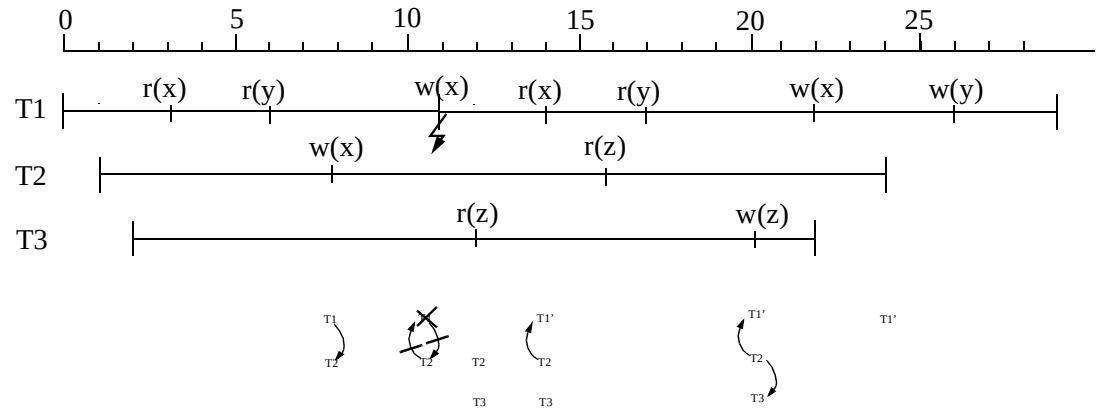
Verletzung der Zeitstempel-Regel zum Zeitpunkt 11 bei T1 (1<2), Neustart als T1' mit Zeitstempel 4. Verletzung der Zeitstempel-Regel zum Zeitpunkt 16 bei T2 (2<3), Neustart als T2' mit Zeitstempel 5.Kein echter Konflikt.
Äquivalente serielle Reihenfolgen: T3, T1, T2, und T1, T3, T2. Gesamtdauer der Schedule: 38 ZE

c) BTO



Hinweis: Die Anwendung der "Thomas' Write Rule" (R3) führt zu einer **scheinbar** nicht konfliktserialisierbaren Schedule: $r_1(y) <_S w_2(x)$ und $w_2(x) <_S w_1(x)$ => Zyklus im Serialisierbarkeitsgraphen (T1 vor T2 und zugleich T2 vor T1). Da R3 jedoch besagt, dass ein Schreiben nicht stattfindet (der blinde Schreibvorgang von T2 zum Zeitpunkt 8 wird also nicht überschrieben), ergibt sich jedoch eine äquivalente serielle Reihenfolge T1, T2, T3. Gesamtdauer der Schedule: 24 ZE.

d) SGT



Zyklus zwischen T1 und T2 durch $w_1(x)$ zum Zeitpunkt 11. Behebung durch sofortigen Abbruch und Neustart von T1 als T1'.

Äquivalente serielle Reihenfolgen sind T2, T1, T3 und T2, T3, T1. Gesamtdauer der Schedule: 29ZE

Aufgabe 3: Optimistische Synchronisation

In dieser Aufgabe werden Validierungsregeln für optimistische Verfahren zur Mehrbenutzerkontrolle betrachtet. Folgende Abkürzungen werden verwendet:

RS_i = Read-Set (T_i) = Menge der Objekte, die Transaktion T_i liest

WS_i = Write-Set (T_i) = Menge der Objekte, die Transaktion T_i schreibt

a) Konstruieren Sie einen Schedule, der zeigt, dass die folgenden Regeln der Validierungsphase nicht notwendigerweise Serialisierbarkeit gewährleisten.
Regeln: Eine Transaktion T_v wird validiert, wenn für alle T_i mit $i < v$ eine der folgenden beiden Bedingungen erfüllt ist:

(1) T_i beendet die Schreibphase, bevor T_v die Lesephase beginnt.

(2) $WS_i \cap RS_v = \varnothing$ und T_i beendet die Lesephase, bevor T_v die Schreibphase beginnt.

b) Beweisen Sie, dass die unter a) aufgeführten Validierungsregeln Serialisierbarkeit garantieren, sofern für alle Transaktionen T_i gilt:
$WS_i \subseteq RS_i$, d. h. jedes Objekt, das geschrieben wird, muss zuvor gelesen werden.

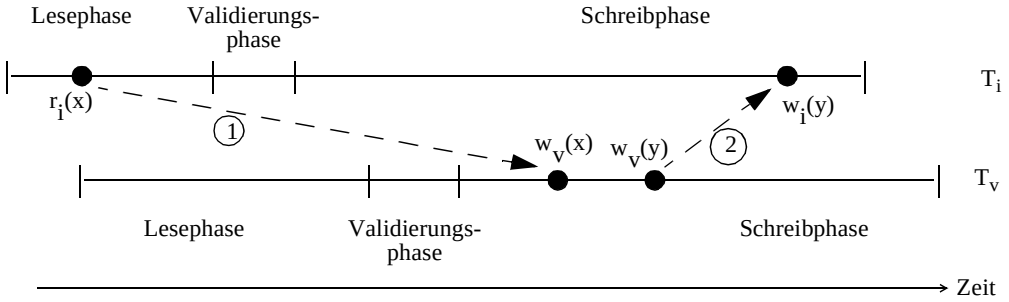
c) Warum ist es ungünstig, Transaktionen in der Reihenfolge ihres Starts zu validieren? Welche andere Reihenfolge erlaubt eine schnellere Abfertigung der Transaktion, d. h. einen höheren Durchsatz?

Lösung:

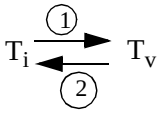
a) Konstruieren Sie einen Schedule, der zeigt, dass die folgenden Regeln der Validierungsphase nicht notwendigerweise Serialisierbarkeit gewährleisten.
Regeln: Eine Transaktion T_v wird validiert, wenn für alle T_i mit $i < v$ eine der folgenden beiden Bedingungen erfüllt ist:

(1) T_i beendet die Schreibphase, bevor T_v die Lesephase beginnt.

(2) $WS_i \cap RS_v = \varnothing$ und T_i beendet die Lesephase, bevor T_v die Schreibphase beginnt.



Der obenstehende Schedule erfüllt die angegebenen Regel 2. Der Schedule ist jedoch nicht serialisierbar, da der zugehörige Serialisierbarkeitsgraph einen Zyklus enthält:



b) Beweisen Sie, dass die unter a) aufgeführten Validierungsregeln Serialisierbarkeit garantieren, sofern für alle Transaktionen T_i gilt:

$WS_i \subseteq RS_i$, d. h. jedes Objekt, das geschrieben wird, muss zuvor gelesen werden.

Regel 1 ist Voraussetzung für Serialisierbarkeit..

Zu zeigen ist also lediglich, dass unter der Voraussetzung $WS_i \subseteq RS_i$ nur Transaktionen T_v mit Regel 2 validiert werden, die die Serialisierbarkeit nicht zerstören.

Daher zeigen wir im Folgenden, dass bei der Hinzunahme des Knotens T_v in den Abhängigkeitsgraph durch Regel 2 nur Kanten von T_i nach T_v möglich sind, nicht umgekehrt, und somit keine Zyklen entstehen können.

Es werden die drei möglichen Konflikttypen zwischen der validierten Transaktion T_v und einer früher gestarteten Transaktion T_i betrachtet:

1. Fall: $\exists$ Objekt x: $x \in RS_i \wedge x \in WS_v$
Da nach Regel 2 T_v die Schreibphase erst beginnt, nachdem T_i die Lesephase beendet hat, muss gelten: $r_i(x)$ vor $w_v(x)$, d. h. im Abhängigkeitsgraph kommt eine Kante von T_i nach T_v.

2. Fall: $\exists$ Objekt x: $x \in WS_i \wedge x \in RS_v$
Unmöglich wegen der Bedingung $WS_i \cap RS_v = \varnothing$ aus Regel 2

3. Fall: $\exists$ Objekt x: $x \in WS_i \wedge x \in WS_v$
Wegen der Forderung $WS_i \subseteq RS_i$ müsste dann auch $x \in RS_v$ gelten, was aber aufgrund der Validierungsbedingung $WS_i \cap RS_v = \varnothing$ nicht möglich ist (siehe 2. Fall).

Damit ist gezeigt, dass unter Voraussetzung $WS_i \subseteq RS_i$ die unter a) aufgeführten Validierungsregeln die Serialisierbarkeit garantieren.

c) Warum ist es ungünstig, Transaktionen in der Reihenfolge ihres Starts zu validieren? Welche andere Reihenfolge erlaubt eine schnellere Abfertigung der Transaktion, d. h. einen höheren Durchsatz?

Bei Validierung in BOT-Reihenfolge muss eine Transaktion T_v auf alle vor ihr gestarteten (unter Umständen sehr lange lesenden) Transaktionen T_i warten, da erst dann alle benötigten WS_i's feststehen.

Um diese Wartezeiten zu vermeiden, ist es günstiger, Transaktionsnummern erst bei Beendigung der Lesephase zu vergeben, so dass gegen Transaktionen, die erst in der Lesephase sind, nicht validiert werden muss.

Aufgabe 4: Sperrprotokolle verschiedener Konsistenzebenen

Mit LOCK (<name>, <modus>) wird für ein Objekt mit dem Namen <name> eine Sperre vom Typ <modus> angefordert. Durch UNLOCK <name> wird eine gewährte Sperre wieder zurückgegeben. Auf welchen Konsistenzebenen laufen die folgende Transaktionen? Welche der folgenden Protokolle sind zweiphasig in Bezug auf

a) Leser

b) Schreiber

c) Leser und Schreiber?

T_1	T_2	T_3	T_4	T_5
L_1:	LOCK (DB, IX)	LOCK (DB, IX)	LOCK (DB, IX)	LOCK (DB, IX)
READ NEXT IN R_1	LOCK (S_1, IX)	LOCK (S_1, IX)	LOCK (S_1, IX)	LOCK (S_1, IX)
IF !EOF GO TO L_1	LOCK (R_1, IX)	LOCK (R_1, IX)	LOCK (R_1, IX)	LOCK (R_1, IX)
LOCK (DB, X)	LOCK (t_{111}, R)	READ t_{111}	READ t_{111}	LOCK (t_{111}, R)
INSERT t_{11k}			READ t_{11i}	READ t_{111}
UNLOCK DB	READ t_{111}	READ t_{112}	UNLOCK R_1	
	UNLOCK t_{111}	READ t_{113}	LOCK (R_1, IX)	LOCK (t_{11i}, R)
		LOCK (t_{113}, X)	LOCK (t_{11i}, X)	READ t_{11i}
	LOCK (t_{11i}, X)	MODIFY t_{113}	MODIFY t_{11i}	LOCK (t_{11i}, X)
	MODIFY t_{11i}	READ t_{114}	UNLOCK t_{11i}	MODIFY t_{11i}
	LOCK (t_{11k}, R)		UNLOCK R_1	
	READ t_{11k}	LOCK (t_{11i}, X)	LOCK (R_1, R)	LOCK (t_{11k}, X)
	UNLOCK t_{11k}	MODIFY t_{11i}	READ t_{11k}	MODIFY t_{11k}
	LOCK (t_{11k}, X)		UNLOCK R_1	UNLOCK t_{11k}
	MODIFY t_{11k}	UNLOCK t_{11i}	LOCK (R_1, IX)	UNLOCK t_{11i}
	UNLOCK t_{11k}	UNLOCK t_{113}	LOCK (t_{11k}, X)	UNLOCK t_{111}
	UNLOCK t_{11i}	UNLOCK R_1	MODIFY t_{11k}	UNLOCK R_1
	UNLOCK R_1	UNLOCK S_1	UNLOCK t_{11k}	UNLOCK S_1
	UNLOCK S_1	UNLOCK DB	UNLOCK R_1	UNLOCK DB
	UNLOCK DB		UNLOCK S_1	
			UNLOCK DB	

Lösung:

Die Konsistenzebenen der verschiedenen Transaktionen sind:

Transaktion	Konsistenzebene	
T1	CL 1 oder CL 0	keine Lese-/ lange oder kurze Schreibsperren
T2	CL 2	kurze Lese-/ lange Schreibsperren
T3	CL 1	keine Lese- / lange Schreibsperren
T4	CL 0	nur kurze Sperren
T5	CL 3	lange Schreib-/ Lesesperren

Folgende Protokolle der Transaktionen sind zweiphasig im Bezug auf

Schreiber:	T1
	T2
	T3
	T5
Leser:	T5
Leser und Schreiber:	T5

T4 verletzt das Prinzip der Zweiphasigkeit für Schreiber und Leser.