

3. Übungsblatt (WS 2019) – Musterlösung

3.0 VU Datenmodellierung 2 / 6.0 VU Datenbanksysteme

Informationen zum Übungsblatt

Allgemeines

In diesem Übungsteil setzen Sie die theoretischen Kenntnisse im Bereich Transaktionsverwaltung, Recovery und Mehrbenutzersynchronisation praktisch um.

Lösen Sie die Beispiele **eigenständig** (auch bei der Prüfung und vermutlich auch in der Praxis sind Sie auf sich alleine gestellt)! Wir weisen Sie darauf hin, dass sämtliche abgeschriebene Lösungen mit 0 Punkten beurteilt werden (sowohl das “Original” als auch die “Kopie”).

Geben Sie ein einziges PDF Dokument ab (max. 5MB). Erstellen Sie Ihr Abgabedokument computerunterstützt. Wir akzeptieren keine PDF-Dateien mit handschriftlichen Inhalten.

Das Übungsblatt enthält 8 Aufgaben, auf welche Sie insgesamt 15 Punkte erhalten können.

Deadlines

bis 11.12. 12:00 Uhr: Upload der Abgabe über TUWEL
ab 07.01. 13:00 Uhr Korrektur und Feedback in TUWEL verfügbar

Tutorensprechstunden (freiwillig)

Rund eine Woche vor der Abgabedeadline bieten die TutorInnen Sprechstunden an. Falls Sie Probleme mit oder Fragen zum Stoff des Übungsblattes haben, es Verständnisprobleme mit den Beispielen oder technische Fragen gibt, kommen Sie bitte einfach vorbei. Die TutorInnen beantworten Ihnen gerne Ihre Fragen zum Stoff, oder helfen Ihnen bei Problemen weiter.

Ziel der Sprechstunden ist es, Ihnen beim **Verständnis des Stoffs** zu helfen, nicht, das Übungsblatt für Sie zu rechnen, oder die eigenen Lösungen vorab korrigiert zu bekommen.

Die Teilnahme ist vollkommen freiwillig — Termine und Orte der Tutorensprechstunden finden Sie in TUWEL.

Durchsprache der Übungsbeispiele (freiwillig)

In den Tagen nach Rückgabe der korrigierten Abgaben gibt es die Möglichkeit die Übungsbeispiele in kleineren Gruppen (max. 25 Personen) durchzusprechen. Jede dieser Gruppen wird von einer Assistentin/einem Assistenten geleitet. Der genaue Ablauf in einer Übungsgruppe kann variieren, und hängt auch von Ihren Wünschen und Fragen ab. Die grundsätzliche Idee ist es, die Beispiele durchzurechnen, und speziell auf Ihre Fragen und mögliche Unklarheiten einzugehen. Die (relativ) kleine Gruppengröße soll eine aktive Teilnahme ermöglichen. Daher ist es auch wichtig, dass Sie sich bereits im Vorfeld mit Ihrer korrigierten Abgabe auseinandersetzen, und Unklarheiten identifizieren. Trauen Sie sich, entsprechend Fragen zu stellen – keine Frage kann irgendeinen (negativen) Einfluss auf Ihre Note haben.

Die Teilnahme an so einer Gruppe ist absolut freiwillig. Um die Gruppengröße klein zu halten ist eine Anmeldung in TUWEL erforderlich. Termine und Orte finden Sie in TUWEL.

Weitere Fragen – TUWEL Forum

Sie können darüber hinaus das TUWEL Forum verwenden, sollten Sie inhaltliche oder organisatorische Fragen haben.

Aufgaben: Recovery

In diesem Abschnitt wird die Verwendung von Log-Einträgen zur Sicherstellung der Eigenschaften Atomicity und Durability von Transaktionen behandelt. Dabei kommt das in der Vorlesung vorgestellte Format für Log-Einträge zur Verwendung, welches hier noch einmal kurz zusammengefasst ist:

Log-Einträge für von einer Transaktion ausgeführte Aktionen haben das Format

[LSN, TA, PageID, Redo, Undo, PrevLSN],

wobei LSN die LogSequenceNumber des Eintrags angibt, TA die Transaktion welche die Aktion ausgeführt hat und PageID die veränderte Seite. Redo und Undo beinhalten die Redo/Undo Information und PrevLSN die LSN des vorhergehenden Log-Eintrags der selben Transaktion.

Die Redo- und Undo-Informationen werden logisch protokolliert, d.h. *relativ* zum Datenbestand mittels Addition bzw. Subtraktion.

Ein Beispiel für einen solchen Log-Eintrag wäre

[#i, T_j, P_X, X+=d₁, X-=d₂, #k],

welcher besagt dass laut *i*-tem Logeintrag die Transaktion T_j auf ein Feld X auf der Seite P_X schreibend zugreift, so dass beim Redo X um d₁ vergrößert werden müsste und beim Undo X um d₂ verkleinert werden müsste. Außerdem hat der vorangegangene Logeintrag dieser Transaktion die Nummer k.

Für die Log-Einträge für den Beginn (BOT – Begin of Transaction) und das Commit von Transaktionen werden nur die LSN, die TA, der Operationsnamen (BOT bzw. commit), sowie die PrevLSN angegeben. D.h. die entsprechenden Log-Einträge haben das Format

[LSN, TA, (BOT|Commit), PrevLSN].

Compensation Log Records (CLRs) folgen dem Format

⟨LSN, TA, PageID, Redo, PrevLSN, UndoNextLSN⟩,

wobei UndoNextLSN die LSN des nächsten Log-Eintrags der Transaktion angibt, welcher rückgängig gemacht werden soll. Äquivalent zu den Standard Log-Einträgen kann auch für BOT-CLRs die verkürzte Schreibweise

⟨LSN, TA, BOT, PrevLSN⟩

verwendet werden.

Aufgabe 1 (Logging)

[1 Punkt]

Betrachten Sie die in Abbildung 1 angegebene Historie der drei Transaktionen T₁, T₂ und T₃. Dabei bezeichnen A, B, C und D Felder in der Datenbank, während a_i, b_i, c_i und d_i lokale Variablen der einzelnen Transaktionen darstellen. Weiters bezeichnet r_i(Γ, γ) eine Leseoperation (der Wert des Feldes Γ wird aus der Datenbasis in eine lokale Variable γ gelesen) und w_i(Γ, γ) eine Schreiboperation (der Wert γ wird in das Feld Γ in der Datenbasis geschrieben). Mit COMMIT wird der erfolgreiche Abschluss einer Transaktion gekennzeichnet. ROLLBACK bezeichnet den Abbruch einer Transaktion. Nehmen Sie dabei an, dass das Zurücksetzen der Transaktion vollständig und erfolgreich abgeschlossen wird bevor die nächste Aktion laut Liste ausgeführt wird (d.h. das ROLLBACK in Schritt 21 wird abgeschlossen bevor der Lesevorgang in Schritt 22 erfolgt).

Nehmen Sie schlussendlich an, dass zu Beginn (Zeile 1) der relevante Datenbestand in der Datenbank wie folgt aussieht:

A: 10 B: 57 C: 31 D: 43

	T_1	T_2	T_3
1		BOT	
2	BOT		
3		$r_2(A, a_2)$	
4	$w_1(B, 0 + 25)$		
5			BOT
6		$r_2(B, b_2)$	
7			$r_3(C, c_3)$
8		$w_2(A, a_2 - 5)$	
9			$w_3(B, 0 + 18)$
10	$r_1(C, c_1)$		
11		$w_2(D, a_2 + b_2)$	
12			$w_3(C, c_3 - 5)$
13	$w_1(C, c_1 + 15)$		
14	$r_1(D, d_1)$		
15			$r_3(D, d_3)$
16		$r_2(A, a_{22})$	
17			$r_3(B, b_3)$
18	COMMIT		
19			$w_3(D, c_3 - b_3)$
20		$w_2(A, a_2 - a_{22})$	
21			ROLLBACK
22		$r_2(D, d_2)$	
23		$w_2(A, d_2 - 5)$	
24		COMMIT	

Abbildung 1: Historie für Aufgabe 1.

- (a) Geben Sie für jede Zeile der Historie, in welcher entweder der Wert eines Feldes, oder einer lokalen Variabel geändert wird, den Wert des entsprechenden Feldes/Variablen *nach* der Operation an. Geben Sie jeweils die dazugehörige Zeilennummer der Historie an (verwenden Sie für Änderungen auf Grund des ROLLBACKs die Zeilennummer 21).

Lösung:

	Wert	14	$d_1 = 35$
3	$a_2 = 10$	15	$d_3 = 35$
4	$B = 25$	16	$a_{22} = 5$
6	$b_2 = 25$	17	$b_3 = 18$
7	$c_3 = 31$	19	$D = 13$
8	$A = 5$	20	$A = 5$
9	$B = 18$	21	$D = 35$
10	$c_1 = 31$	21	$C = 51$
11	$D = 35$	21	$B = 25$
12	$C = 26$	22	$d_2 = 35$
13	$C = 46$	23	$A = 30$

- (b) Geben Sie eine Liste der entsprechenden Log-Einträge zu dieser Historie – in der Reihenfolge in welcher diese Einträge angelegt werden – an. Verwenden Sie dabei das am Beginn dieses Abschnitts beschriebene Format. Denken Sie insbesondere daran, dass die *Redo*- und *Undo* Informationen mittels Addition und Subtraktion angegeben werden sollen. Nehmen Sie an, dass jedes Feld Γ auf der Seite P_Γ gespeichert wird. Zur besseren Lesbarkeit benutzen Sie außerdem bitte die Schreibweise $\#i$ für die LSN bzw. PrevLSN. Sollte es zu einem Eintrag keinen vorangegangenen Eintrag geben so verwenden Sie bitte 0 als Wert. Inkludieren Sie ebenfalls die Log-Einträge für das Zurücksetzen von T_3 .

Hinweis: Formatieren Sie die Log-Einträge bitte auf eine übersichtliche Art und Weise, z.B. in einer Liste (ein Eintrag pro Zeile) oder einer Tabelle (ein Eintrag pro Zeile). Schreiben Sie die Log-Einträge bitte *nicht* als normalen Fließtext hintereinander. Wir behalten es uns vor für unlesbare Formatierungen 0 Punkte zu vergeben. (Falls Sie die L^AT_EX-Vorlage verwenden finden Sie dort bereits einen Vorschlag zur Formatierung.)

Lösung:

	Log: [LSN, TA, PageID, Redo, Undo, PrevLSN] bzw. ⟨LSN, TA, PageID, Redo, PrevLSN, UndoNextLSN⟩
1	[#1, T_2 , BOT, #0]
2	[#2, T_1 , BOT, #0]
4	[#3, T_1 , P_B , B-=-32, B+=32, #2]
5	[#4, T_3 , BOT, #0]
8	[#5, T_2 , P_A , A-=-5, A+=5, #1]
9	[#6, T_3 , P_B , B-=-7, B+=7, #4]
11	[#7, T_2 , P_D , D-=-8, D+=8, #5]
12	[#8, T_3 , P_C , C-=-5, C+=5, #6]
13	[#9, T_1 , P_C , C+=20, C-=20, #3]
18	[#10, T_1 , COMMIT, #9]
19	[#11, T_3 , P_D , D-=-22, D+=22, #8]
20	[#12, T_2 , P_A , A+=0, A-=0, #7]
21	⟨#13, T_3 , P_D , D+=22, #11, #8⟩
21	⟨#14, T_3 , P_C , C+=5, #13, #6⟩
21	⟨#15, T_3 , P_B , B+=7, #14, #4⟩
21	⟨#16, T_3 , BOT, #15⟩
23	[#17, T_2 , P_A , A+=25, A-=25, #12]
24	[#18, T_2 , COMMIT, #17]

Aufgabe 2 (Recovery)

[1.5 Punkte]

Nehmen Sie an, nach einem System-Absturz finden Sie die in Abbildung 2 dargestellte Situation vor. Die linke Seite der Abbildung beschreibt den Inhalt der Logdatei, also der gesicherten Log-Einträge. Auf der rechten Seite der Abbildung ist der Inhalt der Seiten P_A , P_B und P_D dargestellt.

Log-Einträge**Seiten im Hintergrundspeicher**

[#1, T_2 , BOT, #0]
 [#2, T_1 , BOT, #0]
 [#3, T_3 , BOT, #0]
 [#4, T_1 , P_B , B-=45, B+=45, #2]
 [#5, T_2 , P_A , A+=1, A-=1, #1]
 [#6, T_3 , P_D , D+=111, D-=111, #3]
 [#7, T_4 , BOT, #0]
 [#8, T_1 , P_B , B+=51, B-=51, #4]
 <#9, T_1 , P_B , B-=51, #8, #4>
 [#10, T_2 , P_B , C+=50, C-=50, #5]
 [#11, T_2 , P_B , C+=157, C-=157, #10]
 <#12, T_2 , P_B , C-=157, #11, #10>
 <#13, T_1 , P_B , B+=45, #9, #2>
 [#14, T_4 , P_D , D+=5, D-=5, #7]
 <#15, T_2 , P_B , C-=50, #12, #5>
 [#16, T_3 , P_B , C-=206, C+=206, #6]
 <#17, T_1 , BOT, #13>
 [#18, T_4 , P_B , B+=183, B-=183, #14]
 [#19, T_5 , BOT, #0]
 [#20, T_5 , P_A , A+=5, A-=5, #19]
 [#21, T_5 , COMMIT, #20]

P_A	LSN: #5
A = 50	
P_B	LSN: #13
B = 70	C = 111
P_D	LSN: #6
D = 66	

Abbildung 2: Angabe zu Aufgabe 2: Der Inhalt des Log-Archivs (links) sowie der Datenbankseiten (rechts) nach einem Absturz.

Führen Sie an Hand dieser Informationen einen Wiederanlauf (Recovery) der Datenbank durch.

- (a) Bestimmen Sie die Werte für A , B , C und D nach der *Redo*-Phase.

Lösung:

A: 55; B: 253; C: -145; D: 71

- (b) Erzeugen Sie die Compensation Log Records (CLRs), welche während des Wiederanlaufs geschrieben werden.

Lösung:

Log:

LSN	TA	PageID	Redo	PrevLSN	UndoNextLSN
-----	----	--------	------	---------	-------------

#22	T ₄	P _B	B-=183	#18	#14
-----	----------------	----------------	--------	-----	-----

#23	T ₃	P _B	C+=206	#16	#6
-----	----------------	----------------	--------	-----	----

#24	T ₄	P _D	D-=5	#22	#7
-----	----------------	----------------	------	-----	----

#25	T ₄	BOT		#24	
-----	----------------	-----	--	-----	--

#26	T ₃	P _D	D-=111	#23	#3
-----	----------------	----------------	--------	-----	----

#27	T ₂	P _A	A-=1	#15	#1
-----	----------------	----------------	------	-----	----

#28	T ₃	BOT		#26	
-----	----------------	-----	--	-----	--

#29	T ₂	BOT		#27	
-----	----------------	-----	--	-----	--

(c) Geben Sie die Werte von A , B , C und D nach dem Wiederanlauf an.

Lösung:

A: 54; B: 70 C: 61 D: -45

Aufgaben: Mehrbenutzersynchronisation

Aufgabe 3 (Eigenschaften von Transaktionen)

[2.6 Punkte]

Gegeben ist die Menge \mathcal{T} an Transaktionen sowie die dazugehörige Historie \mathcal{H} , welche durch ihre Folge von Elementaroperationen gegeben ist:

$$\mathcal{T} = \{T_1, T_2, T_3, T_4, T_5, T_6, T_7\}$$

$$\begin{aligned} \mathcal{H} = & b_1 \rightarrow r_1(A) \rightarrow b_2 \rightarrow r_2(B) \rightarrow r_2(A) \rightarrow w_2(B) \rightarrow b_3 \rightarrow w_3(C) \rightarrow b_7 \rightarrow w_1(A) \rightarrow r_7(C) \\ & \rightarrow b_6 \rightarrow b_4 \rightarrow r_6(A) \rightarrow w_4(A) \rightarrow r_7(A) \rightarrow b_5 \rightarrow w_6(A) \rightarrow r_3(A) \rightarrow r_7(B) \rightarrow r_6(B) \rightarrow r_7(C) \\ & \rightarrow w_5(D) \rightarrow w_6(C) \rightarrow a_6 \rightarrow a_1 \rightarrow r_5(B) \rightarrow r_2(C) \rightarrow r_5(C) \rightarrow c_2 \rightarrow r_3(C) \rightarrow w_5(C) \rightarrow c_3 \\ & \rightarrow w_5(B) \rightarrow c_7 \rightarrow c_5 \rightarrow w_4(D) \rightarrow c_4. \end{aligned}$$

- (a) Zeichnen Sie den Serialisierbarkeitsgraphen $\text{SG}(\mathcal{H})$.
- (b) Geben Sie für jede Kante im Serialisierbarkeitsgraphen mindestens ein Paar $p_i \rightarrow p_j$ von Operationen an, welches begründet warum diese Kante Teil des Graphen ist.
Geben Sie für die Kanten $T_2 \rightarrow T_5$, $T_3 \rightarrow T_6$ und $T_4 \rightarrow T_7$ (sofern sie tatsächlich Teil des Graphen sind) *sämtliche* Konfliktoperationen an, welche verlangen dass diese Kanten Teil des Graph sind.
- (c) Falls die Historie konfliktserialisierbar ist, geben Sie *eine* mögliche konfliktäquivalente serielle Reihenfolge an. Andernfalls geben Sie eine (möglichst kleine) Menge an Transaktionen an welche man abbrechen müsste damit die Historie konfliktserialisierbar wird. Geben Sie anschließend eine mögliche konfliktäquivalente serielle Reihenfolge an.
- (d) Geben Sie für die Historie \mathcal{H} die Leseabhängigkeiten zwischen den Transaktionen an (d.h., geben Sie welche Transaktionen von welchen Transaktionen lesen). Geben Sie zu jeder Leseabhängigkeit mindestens ein Paar $(w_i(X), r_j(X))$ von Operationen an, welches die Leseabhängigkeit belegt.
- (e) Bestimmen Sie, welche der folgenden Eigenschaften die Historie \mathcal{H} besitzt:
- Rücksetzbar
 - Vermeidet kaskadierendes Rücksetzen
 - Strikt

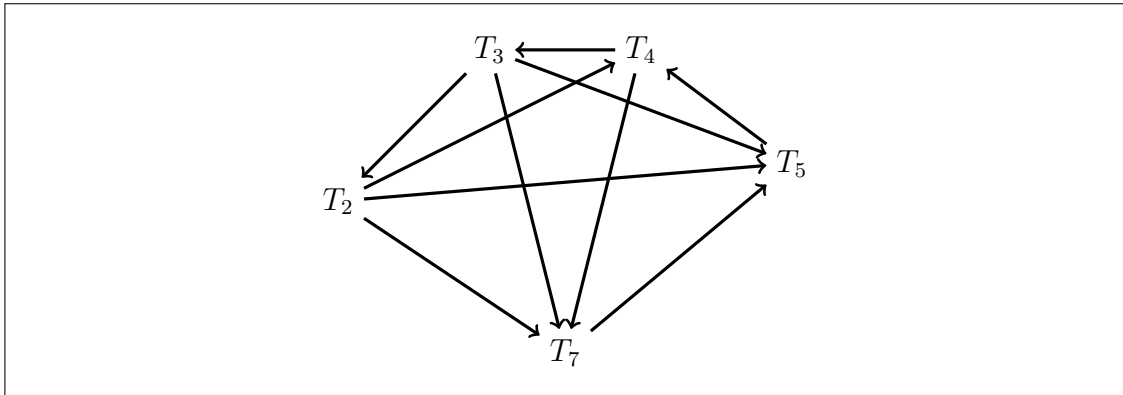
Begründen Sie jeweils Ihre Antwort.

- (f) Bestimmen Sie, welche der drei Eigenschaften aus Aufgabe (e) (Rücksetzbar, vermeidet kaskadierendes Rücksetzen und strikt) die folgende Historie bestehend aus vier Transaktionen T_1 , T_2 , T_3 und T_4 hat:

$$\begin{aligned} & b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b_4 \rightarrow r_3(A) \rightarrow w_2(A) \rightarrow w_4(C) \rightarrow r_1(B) \rightarrow r_2(A) \rightarrow w_4(C) \rightarrow r_2(A) \rightarrow \\ & a_4 \rightarrow w_3(C) \rightarrow w_2(B) \rightarrow c_2 \rightarrow r_1(A) \rightarrow w_3(B) \rightarrow r_3(A) \rightarrow w_1(C) \rightarrow c_3 \rightarrow r_1(C) \rightarrow \\ & r_1(B) \rightarrow c_1 \end{aligned}$$

Lösung:

(a) Serialisierbarkeitsgraph:



(b) “Begründung” für die Kanten:

$T_2 \rightarrow T_4$

- $r_2(A) \rightarrow w_4(A)$

$T_2 \rightarrow T_5$

- $r_2(B) \rightarrow w_5(B)$
- $r_2(C) \rightarrow w_5(C)$
- $w_2(B) \rightarrow r_5(B)$
- $w_2(B) \rightarrow w_5(B)$

$T_2 \rightarrow T_7$

- $w_2(B) \rightarrow r_7(B)$

$T_3 \rightarrow T_2$

- $w_3(C) \rightarrow r_2(C)$

$T_3 \rightarrow T_5$

- $r_3(C) \rightarrow w_5(C)$
- $w_3(C) \rightarrow r_5(C)$
- $w_3(C) \rightarrow w_5(C)$

$T_3 \rightarrow T_7$

- $w_3(C) \rightarrow r_7(C)$
- $w_3(C) \rightarrow r_7(C)$

$T_4 \rightarrow T_3$

- $w_4(A) \rightarrow r_3(A)$

$T_4 \rightarrow T_7$

- $w_4(A) \rightarrow r_7(A)$

$T_5 \rightarrow T_4$

- $w_5(D) \rightarrow w_4(D)$

$T_7 \rightarrow T_5$

- $r_7(B) \rightarrow w_5(B)$
- $r_7(C) \rightarrow w_5(C)$
- $r_7(C) \rightarrow w_5(C)$

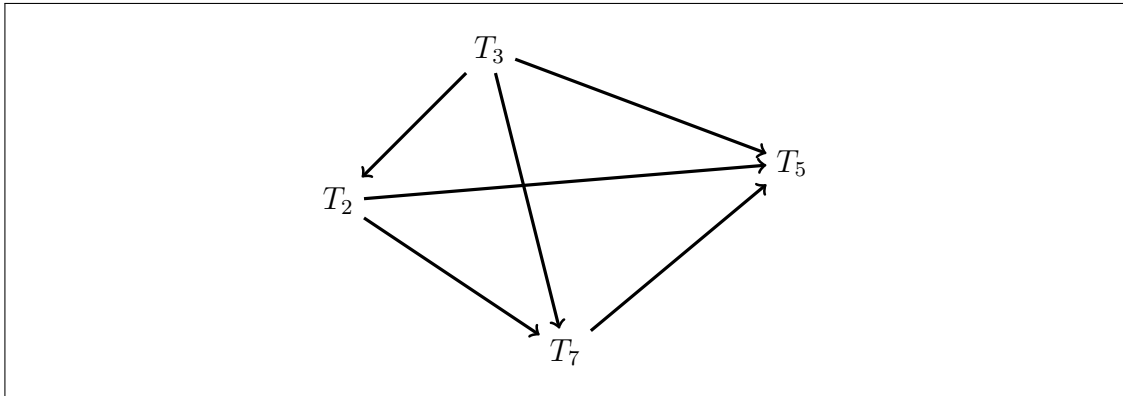
(c) Konflikt-Serialisierbarkeit:

Nein, die Historie **ist nicht** konfliktserialisierbar: Der Serialisierbarkeitsgraph enthält sogar mehrere Zyklen, z.B.:

- $T_3 \rightarrow T_2 \rightarrow T_7 \rightarrow T_5 \rightarrow T_4 \rightarrow T_3$
- $T_3 \rightarrow T_2 \rightarrow T_4 \rightarrow T_3$
- $T_3 \rightarrow T_2 \rightarrow T_5 \rightarrow T_4 \rightarrow T_3$

- $T_3 \rightarrow T_7 \rightarrow T_5 \rightarrow T_4 \rightarrow T_3$
- $T_3 \rightarrow T_5 \rightarrow T_4 \rightarrow T_3$
- $T_4 \rightarrow T_7 \rightarrow T_5 \rightarrow T_4$

Nachdem T_4 in allen Zyklen vor kommt, reicht es die Transaktion T_4 abzurechnen, um eine konfliktserialisierbare Historie zu erhalten. Der resultierende Serialisierbarkeitsgraph ist



(Dieser Serialisierbarkeitsgraph wird in der Angabe nicht gefordert!)

Die Historie der verbleibenden Transaktionen ist nun konfliktserialisierbar, eine mögliche äquivalente serielle Ausführungsreihenfolge wäre

$$T_3 \text{ vor } T_2 \text{ vor } T_7 \text{ vor } T_5$$

(d) **Lese-Abhängigkeiten:**

- T_2 liest von T_3 : $(w_3(C), r_2(C))$ ($w_6(C)$ wurde zurückgesetzt)
- T_3 liest von T_6 : $(w_6(A), r_3(A))$
- T_5 liest von T_2 : $(w_2(B), r_5(B))$
- T_5 liest von T_3 : $(w_3(C), r_5(C))$ ($w_6(C)$ wurde zurückgesetzt)
- T_6 liest von T_1 : $(w_1(A), r_6(A))$
- T_6 liest von T_2 : $(w_2(B), r_6(B))$
- T_7 liest von T_2 : $(w_2(B), r_7(B))$
- T_7 liest von T_3 : $(w_3(C), r_7(C))$, $(w_3(C), r_7(C))$
- T_7 liest von T_4 : $(w_4(A), r_7(A))$

(e) **Klassifikation der Historie:**

- **Rücksetzbar:**

Nein, die Historie *ist nicht* rücksetzbar. Eine Historie ist rücksetzbar wenn keine Transaktion ihr COMMIT vor einer Transaktion durchführt, von welcher sie gelesen hat. Mit Ausnahme von T_6 , welche von T_2 liest und abbricht (daher nie ein COMMIT ausführt), T_7 welche nach T_2 und T_3 , von welchen sie liest, ihr COMMIT hat (aber noch vor T_4 , von welcher sie ebenfalls liest) und T_5 welche nach T_2 und T_3 , von denen sie liest, ihr COMMIT hat, committen alle lesenden Transaktionen vor den Transaktionen von denen sie lesen.

- **Vermeidet Kaskadierendes Rücksetzen:**

Nein. Dies ergibt sich zum Einen sofort daraus dass die Historie nicht rücksetzbar ist. Zum Anderen lässt sich auch ein konkretes Gegenbeispiel angeben. Um kaskadierendes Rücksetzen zu vermeiden ist es notwendig dass bei jeder Leseoperation ein Wert gelesen wird, welcher von einer bereits erfolgreich abgeschlossenen (committed) Transaktion geschrieben wurde. Dies wird in der vorliegenden Historie in vielen Fällen verletzt. Ein solches Beispiel ist gleich am Ende der ersten Zeile $r_7(C)$ liest den Wert von $w_3(C)$, obwohl T_3 zu diesem Zeitpunkt noch aktiv ist.

- **Strikte Historie:**

Nein. Dies ergibt sich zum Einen wiederum daraus, dass die Historie kein kaskadierendes Rücksetzen vermeidet. Des Weiteren ist das obige Gegenbeispiel $r_7(C)$ und $w_3(C)$ auch ein Beispiel dafür, dass die Historie nicht strikt ist. Ein weiteres Beispiel welches die zusätzliche Bedingung an strikte Historien verletzt ist die Operation $w_5(C)$ am Ende der dritten Zeile, welche die Operation $w_3(C)$ überschreibt, ohne dass T_3 zuerst beendet wurde.

(f) **Eigenschaften der zweiten Historie:**

- **Rücksetzbar:**

Ja, die Historie ist rücksetzbar. Es lesen T_1 und T_3 von T_2 , und das COMMIT von T_2 findet vor den COMMITs von T_1 und T_3 statt. Außerdem liest T_1 von T_3 , committed aber wiederum erst nach T_3 .

- **Vermeidet Kaskadierendes Rücksetzen:**

Ja, die Historie vermeidet kaskadierendes Rücksetzen. Jeder Lesezugriff auf ein Datum welches von einer anderen Transaktion geschrieben wurde erfolgt erst, nachdem die schreibenden Transaktion committed wurde.

- **Strikte Historie:**

Nein. Die Historie ist nicht strikt, da T_1 den Wert von C überschreibt, bevor T_3 (die Transaktion welche diesen Wert davor geschrieben hatte) committed. D.h. wir haben $w_3(C)$ vor $w_1(C)$, ohne dass vor dem $w_1(C)$ ein c_3 stattgefunden hätte.

Aufgabe 4 (Sperranforderungen)

[2.5 Punkte]

Gegeben ist die untenstehende Folge von Sperranforderungen, wobei „ $\text{lockS}_i(O)$ “ (bzw. „ $\text{lockX}_i(O)$ “) bedeutet, dass eine Transaktion T_i eine Lesesperre (bzw. eine Schreibsperre) auf das Datenobjekt O anfordert, und „ $\text{rel}_i(O)$ “ bedeutet dass eine Transaktion T_i sämtliche Sperren auf das Datenobjekt O aufgibt:

$\text{lockX}_1(C) \rightarrow \text{lockS}_3(D) \rightarrow \text{lockS}_2(D) \rightarrow \text{lockX}_4(A) \rightarrow \text{lockX}_4(D) \rightarrow \text{lockS}_1(A) \rightarrow \text{lockX}_2(B) \rightarrow \text{lockS}_4(B) \rightarrow \text{rel}_2(D) \rightarrow \text{lockS}_3(C) \rightarrow \text{lockX}_3(B)$.

- (a) Nehmen Sie an, ein DBMS erhält die angegebene Folge von Sperranforderungen und arbeitet sie in der genannten Reihenfolge ab, wobei Transaktionen, welchen eine gewünschte Sperre nicht gewährt wird, angehalten werden. (D.h. nachfolgende Sperranforderungen der selben Transaktion werden ignoriert und aufgeschoben bis die Transaktion wieder aktiv wird.)

Geben Sie an, in welcher Reihenfolge das DBMS die Sperranforderungen abarbeitet, und geben Sie unmittelbar nach einer Sperranforderung an ob es die gewünschte Sperre gewährt oder die entsprechende Transaktion auf die Sperre warten muss. Verwenden Sie $\text{grantS}_i(O)$ bzw. $\text{grantX}_i(O)$ um anzugeben, dass eine Lese- bzw. Schreibsperre auf dem Datenobjekt O gewährt wurde, verwenden Sie $\text{wait}(i)$ um anzuzeigen, dass eine Transaktion angehalten wurde um auf eine Sperre zu warten, verwenden Sie $\text{relS}_i(O)$ bzw. $\text{relX}_i(O)$ um anzugeben, dass eine Lese- bzw. Schreibsperre auf dem Datenobjekt O freigegeben wurde (als Reaktion auf ein $\text{rel}_i(O)$), und $\text{resume}(i)$ um anzuzeigen dass die Blockierung einer Transaktion wieder aufgehoben wurde, weil das gewünschte Feld nun verfügbar ist.

Nehmen Sie dabei an, dass wenn eine blockierte Transaktion durch die Freigabe einer Sperre wieder aktiv wird diese Transaktion zuerst alle “übersprungenen” Aktionen nachholt, bis sie entweder wieder blockiert oder es keine weiteren ausgelassenen Aktionen dieser Transaktion gibt. Erst danach soll mit dem Abarbeiten der Aktionen bei der ursprünglichen Freigabe fortgefahren werden.

Beispiel: Nehmen Sie die Folge

$\text{lockS}_1(A) \rightarrow \text{lockS}_2(A) \rightarrow \text{lockX}_1(A) \rightarrow \text{lockX}_2(B) \rightarrow \text{lockS}_1(B)$

von Sperranforderungen zweier Transaktionen T_1, T_2 .

Wir erhalten die Liste

1:	$\text{lockS}_1(A)$
2:	$\text{grantS}_1(A)$
3:	$\text{lockS}_2(A)$
4:	$\text{grantS}_2(A)$
5:	$\text{lockX}_1(A)$
6:	$\text{wait}(1)$
7:	$\text{lockX}_2(B)$
8:	$\text{grantX}_2(B)$

Lösung:

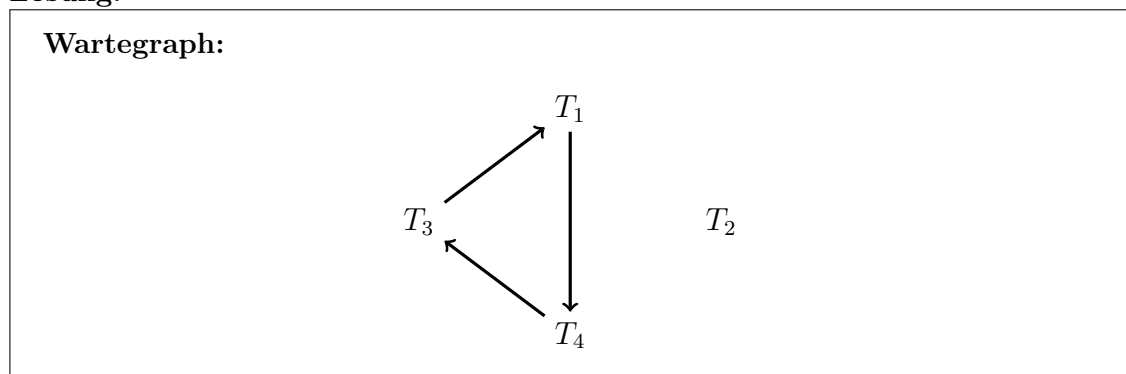
1:	lockX ₁ (C)	10:	wait(4)
2:	grantX ₁ (C)	11:	lockS ₁ (A)
3:	lockS ₃ (D)	12:	wait(1)
4:	grantS ₃ (D)	13:	lockX ₂ (B)
5:	lockS ₂ (D)	14:	grantX ₂ (B)
6:	grantS ₂ (D)	15:	rel ₂ (D)
7:	lockX ₄ (A)	16:	relS ₂ (D)
8:	grantX ₄ (A)	17:	lockS ₃ (C)
9:	lockX ₄ (D)	18:	wait(3)

- (b) Skizzieren Sie bitte die aktuelle Situation der gehaltenen Sperren bzw. angehaltener Transaktionen. Geben Sie dazu eine wie weiter unten dargestellte Tabelle an. Tragen Sie in ein Feld ein *X* (bzw. ein *S*) ein, wenn die entsprechende Transaktion eine Schreibsperre (bzw. eine Lesesperre) auf dieses Datenobjekt besitzt. Tragen Sie für jede blockierte Transaktion bitte zusätzlich für jene Sperranforderung auf Grund welcher die Transaktion nun blockiert ist ein *WS* (*wait shared*) bzw. *WX* (*wait exclusive*) in das entsprechende Feld ein.

Lösung:

	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>
<i>T</i> ₁	WS		X	
<i>T</i> ₂		X		
<i>T</i> ₃			WS	S
<i>T</i> ₄	X			WX

- (c) Geben Sie den der aktuellen Situation entsprechenden Wartegraphen an.

Lösung:

- (d) Geben Sie an, ob zum aktuellen Zeitpunkt ein Deadlock besteht. **Lösung: Ja**
- (e) Geben Sie eine mögliche Sequenz an Freigaben an, mit welcher sämtliche gehaltenen Sperren wieder freigegeben werden können. Sollte eine blockierte Transaktion durch die Freigabe einer Sperre weiterlaufen können, so müssen von dieser Transaktion zuerst alle ausstehenden Sperranfragen und Freigaben aus der Angabe abgearbeitet werden, bevor Sie zusätzliche Freigaben definieren können. Sollte derzeit ein Deadlock bestehen, so soll Transaktion T_4 abgebrochen werden (d.h. alle Sperren von T_4 werden sofort freigegeben).

Lösung:

Nachdem derzeit ein Deadlock besteht wird zuerst T_4 abgebrochen, was zu folgender Freigabe führt:

$$\text{rel}_4(A) \rightarrow \text{relX}_4(A)$$

Auf Grund dieser Freigabe kann nun T_1 weiterlaufen, und wir erhalten:

$$\text{resume}(1) \rightarrow \text{grantS}_1(A)$$

Nun kann T_1 ihre Sperren wieder freigeben

$$\text{rel}_1(A) \rightarrow \text{relS}_1(A) \rightarrow \text{rel}_1(C) \rightarrow \text{relX}_1(C)$$

Dadurch kann Transaktion T_3 weiterlaufen

$$\text{resume}(3) \rightarrow \text{grantS}_3(C) \rightarrow \text{lockX}_3(B) \rightarrow \text{wait}(3)$$

Nun kann T_2 ihre Sperre aufgeben

$$\text{rel}_2(B) \rightarrow \text{relX}_2(B)$$

Was nun dazu führt dass T_3 die letzte Sperre bekommt,

$$\text{resume}(3) \rightarrow \text{grantX}_3(B)$$

und anschließend ebenfalls alle Sperren freigeben kann.

$$\text{rel}_3(D) \rightarrow \text{relS}_3(D) \rightarrow \text{rel}_3(C) \rightarrow \text{relS}_3(C) \rightarrow \text{rel}_3(B) \rightarrow \text{relX}_3(B)$$

- (f) Betrachten Sie noch einmal die gegebene Sequenz an Sperranforderungen und Freigaben. Widerspricht diese dem Zwei Phasen Sperrprotokoll? Wie sieht es bei der von Ihnen in Aufgabe (e) erstellten Sequenz aus?

Lösung:

Nein. In beiden Fällen fordert keine Transaktion mehr eine Sperre an, nachdem von ihr bereits eine Sperre wieder freigegeben wurde. Dies entspricht den Vorgaben des 2PL.

Aufgabe 5 (Zwei-Phasen-Sperrprotokoll)

[1.5 Punkte]

Betrachten Sie die folgenden drei Transaktionen T_1 , T_2 und T_3 , für welche jeweils eine Folge von Elementaroperationen gegeben ist ($r_i(O)$ und $w_i(O)$ bezeichnen eine Lese- bzw. Schreiboperation von T_i auf O , und c_i bezeichnet das commit von T_i).

$$\begin{array}{l} T_1: \quad r_1(A) \rightarrow r_1(B) \rightarrow w_1(A) \rightarrow w_1(C) \rightarrow r_1(B) \rightarrow c_1 \\ T_2: \quad r_2(A) \rightarrow r_2(D) \rightarrow w_2(D) \rightarrow r_2(A) \rightarrow w_2(E) \rightarrow c_2 \\ T_3: \quad w_3(C) \rightarrow r_3(C) \rightarrow r_3(B) \rightarrow w_3(D) \rightarrow r_3(E) \rightarrow c_3 \end{array}$$

Nehmen Sie an, zur Synchronisation dieser Transaktionen wird das (“normale”) *2-Phasen Sperrprotokoll* verwendet. Geben Sie die dadurch entstehende Historie (bestehend aus Sperranforderungen, Lese- und Schreiboperationen, Freigaben sowie commits) an.

Treffen Sie dabei folgende Annahmen:

- *Notation:* Verwenden Sie bitte die Notation $\text{lockS}_i(O)$ und $\text{lockX}_i(O)$ um das Anfordern einer Lese- bzw. Schreibsperre von Transaktion T_i auf das Objekt O anzuschreiben. Verwenden Sie bitte ebenso $\text{rel}_i(O)$ für die Freigabe sämtlicher Sperren von T_i auf O . (*Hinweis:* Sie brauchen die Information ob eine Sperre gewährt wird oder eine Transaktion blockiert wird nicht explizit angeben. Das ergibt sich daraus, ob auf eine Sperranforderung einer Transaktion eine entsprechende Operation dieser Transaktion folgt, oder nicht).
- *Sperranforderungen und Freigaben:* Geben Sie zu jeder Operation (lesen, schreiben, commit) die benötigten Sperranforderungen an (sofern diese von der Transaktion noch nicht gehalten werden). Nehmen Sie dabei an, dass Sperren so sparsam wie möglich beantragt werden. D.h.
 - Es werden nur Sperren beantragt die auch tatsächlich benötigt werden.
 - Sperren werden so kurz wie möglich gehalten, d.h. sie werden spät wie möglich beantragt, und so früh wie möglich wieder freigegeben.
- *Verzahnung der Transaktionen:* Nehmen Sie an, dass jede Transaktion jeweils eine Operation (lesen, schreiben, commit) abarbeitet, und anschließend die nächste Transaktion ausgeführt wird. D.h. im gegebenen Fall wäre die Reihenfolge der Aktionen $r_1(A) \rightarrow r_2(A) \rightarrow w_3(C) \rightarrow r_1(B) \rightarrow \dots$. Sperranforderungen und Freigaben zählen hierbei nicht als Operationen, d.h. vor und nach jeder Operation (lesen, schreiben, commit) darf die Transaktion eine beliebige Anzahl von Sperranforderungen und Freigaben ausführen, bevor die nächste Transaktion an die Reihe kommt.

Von dieser Reihenfolge soll nur abgewichen werden, wenn eine Transaktion blockiert ist. Dann wird die Transaktion für diese Runde übersprungen. Die passiert so lange, bis die Blockierung wieder aufgehoben wird, dann nimmt die Transaktion wieder ganz normal am Ablauf teil – jedoch weiterhin mit nur einer einzigen Operation (lesen, schreiben, commit), bevor wieder die anderen Transaktionen an der Reihe sind.

Das folgende Beispiel soll den Ablauf an Hand von zwei Transaktionen demonstrieren. Nehmen wir an, T_1 besteht aus Aktionen $\alpha_1, \alpha_2, \dots$, und T_2 besteht aus Aktionen β_1, β_2, \dots . Der normale Ablauf wäre $\alpha_1, \beta_1, \alpha_2, \beta_2, \dots$. Angenommen T_2 bräuchte für die Aktion β_3 eine Sperre, welche T_1 hält. D.h. T_2 blockiert. Dann wäre der weitere Ablauf $\alpha_3, \alpha_4, \alpha_5, \dots$. Wird die Sperre nach Aktion α_5 wieder aufgegeben, so ist die weitere Abfolge $\alpha_5, \beta_3, \alpha_6, \beta_4, \dots$.

Lösung:

#	T_1	T_2	T_3
1	lockS ₁ (A)		
2	r ₁ (A)		
3		lockS ₂ (A)	
4		r ₂ (A)	
5			lockX ₃ (C)
6			w ₃ (C)
7	lockS ₁ (B)		
8	r ₁ (B)		
9		lockS ₂ (D)	
10		r ₂ (D)	
11			r ₃ (C)
12	lockX ₁ (A)		
13		lockX ₂ (D)	
14		w ₂ (D)	
15			lockS ₃ (B)
16			r ₃ (B)
17		r ₂ (A)	
18			lockX ₃ (D)
19		lockX ₂ (E)	
20		w ₂ (E)	
21		rel ₂ (E)	
22		rel ₂ (A)	
23		rel ₂ (D)	
24			w ₃ (D)
25	w ₁ (A)		
26		c ₂	
27			lockS ₃ (E)
28			r ₃ (E)
29			rel ₃ (E)
30			rel ₃ (C)
31			rel ₃ (D)
32			rel ₃ (B)
33	lockX ₁ (C)		
34	w ₁ (C)		
35	rel ₁ (C)		
36	rel ₁ (A)		
37			c ₃
38	r ₁ (B)		
39	rel ₁ (B)		
40	c ₁		

Aufgabe 6 (Multi-Granularity Locking)

[2.4 Punkte]

Betrachten Sie die Datenbasis-Hierarchie in Abbildung 3.

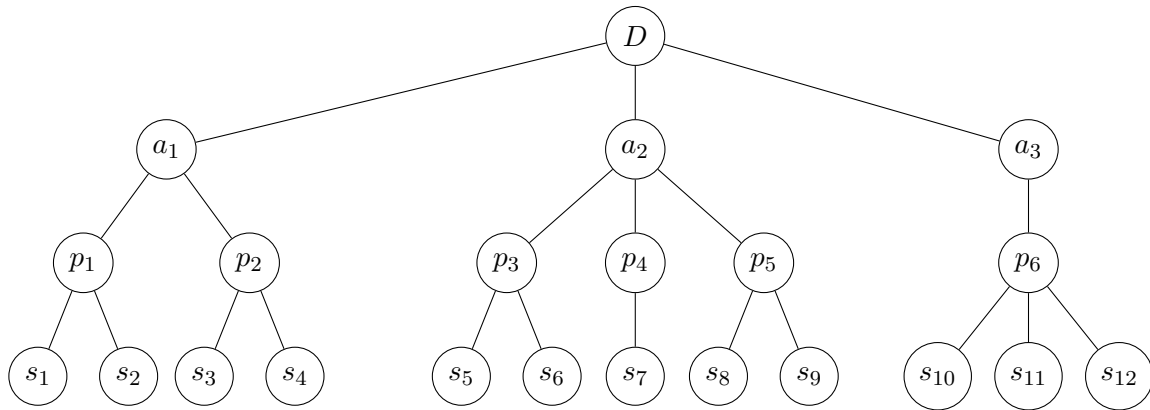


Abbildung 3: Datenbasis-Hierarchie zu Aufgabe 6

Betrachten Sie die folgenden Sequenzen von Sperranforderungen bzw. Freigaben der vier Transaktionen T_1 , T_2 , T_3 und T_4 auf die in Abbildung 3 dargestellten Ressourcen.

- (a) $\text{lockS}_1(p_2) \rightarrow \text{lockX}_2(s_9) \rightarrow \text{lockS}_3(p_6) \rightarrow \text{lockX}_1(p_3) \rightarrow \text{lockS}_3(s_4) \rightarrow \text{lockX}_2(s_3) \rightarrow \text{lockS}_4(a_1) \rightarrow \text{rel}_1(p_2)$
- (b) $\text{lockX}_1(p_4) \rightarrow \text{lockS}_2(p_5) \rightarrow \text{lockX}_3(p_1) \rightarrow \text{lockS}_3(s_8) \rightarrow \text{lockX}_2(s_5) \rightarrow \text{lockX}_1(s_6) \rightarrow \text{lockX}_1(p_1) \rightarrow \text{lockS}_2(p_3) \rightarrow \text{lockS}_4(s_{10}) \rightarrow \text{rel}_3(s_8) \text{ lockX}_3(s_7) \rightarrow \text{rel}_1(p_4)$

Dabei bedeutet $\text{lockS}_i(O)$ dass die Transaktion T_i eine Lesesperre (Shared lock) auf das Object O anfordert, $\text{lockX}_i(O)$ dass die Transaktion T_i eine Schreibsperre (eXclusive lock) auf das Object O anfordert, und $\text{rel}_i(O)$ dass Transaktion T_i alle für das Objekt O gehaltene Sperren freigibt.

Bearbeiten Sie folgende Aufgabenstellungen für jede der beiden angegebenen Sequenzen:

- Geben Sie an, wie vorgegangen werden muss um die Sperranforderungen bzw. Freigaben entsprechend dem Protokoll des Multi Granularity Lockings (MGL) zu verarbeiten: Geben Sie die Sequenz der benötigten Sperranforderungen aus, bzw. im Fall einer Freigabe, geben Sie an welche weiteren Sperren freigegeben werden können. *Hinweis:* Achten Sie sowohl beim Anfordern der Sperren als auch bei der Freigabe auf die richtige Ordnung. Verwenden Sie bitte die folgende Notation: $\text{lockIS}_i(O)$, $\text{lockIX}_i(O)$, $\text{lockS}_i(O)$ und $\text{lockX}_i(O)$ für das Anfordern einer IS-, IX-, S- bzw. X-Sperre von Transaktion T_i auf das Objekt O ; $\text{relIS}_i(O)$, $\text{relIX}_i(O)$, $\text{relS}_i(O)$ und $\text{relX}_i(O)$ für die Freigabe einer IS-, IS-, S- bzw. X-Sperre von Transaktion T_i auf das Objekt O . Achten Sie bitte außerdem darauf, dass nur Sperren angefordert werden, welche die Transaktionen nicht bereits besitzen.
- Markieren Sie Sperren, welche nicht gewährt werden können. Nehmen Sie an, dass die entsprechende Transaktion in so einem Fall angehalten wird, d.h. es werden keine weiteren Aktionen dieser Transaktion durchgeführt bis die benötigte Sperre auf Grund einer

Freigabe der anderen Transaktion gewährt werden kann. (Sperranforderungen bzw. Freigaben angehaltener Transaktionen werden bis dahin einfach übersprungen.) Kann eine zuvor angehaltene Transaktion auf Grund einer Freigabe weiterlaufen, so nehmen Sie an dass alle “übersprungenen” Aktionen ausgeführt werden bevor die Abarbeitung der Sequenz nach der Freigabeaktion weitergeführt wird.

- Geben Sie zu jeder nicht gewährten Sperranforderung an, warum diese Sperre verweigert wurde.
- Entsteht während der Abarbeitung der Sequenz ein Deadlock? Falls ja, warum?
- Falls es zu keinem Deadlock kommt, am Ende der Sequenz jedoch eine Transaktion blockiert ist: Geben Sie eine minimale Menge an Sperren an, welche Transaktionen freigeben müssen, damit die blockierten Transaktionen weiterlaufen können. Beachten Sie dabei, dass Transaktionen nur Schreib- und Lesesperren explizit freigeben können; geben Sie aber auch an welche IX- und IS- Sperren dadurch implizit freigegeben werden können. (Achten Sie dabei auf die richtige Reihenfolge.)

Führen Sie anschließend die blockierte Transaktion weiter. Sollte es dabei zu weiteren Blockierungen kommen, geben Sie jeweils wiederum eine Menge minimale Menge an Freigaben an damit die Transaktion weiterlaufen kann.

Hinweis:

- Sollte der Fall eintreten, dass eine Transaktion eine Sperre auf einem Knoten erhält, welche sie bereits für einen oder mehrere Kindknoten hält, so können Sie davon ausgehen dass die entsprechenden Sperren automatisch in sämtlichen betroffenen Kindknoten entfernt werden (dies brauchen Sie nicht angeben).

Lösung:

(a)		(b)	
1:	lockIS ₁ (D)	1:	lockIX ₁ (D)
2:	lockIS ₁ (a ₁)	2:	lockIX ₁ (a ₂)
3:	lockS ₁ (p ₂)	3:	lockX ₁ (p ₄)
4:	lockIX ₂ (D)	4:	lockIS ₂ (D)
5:	lockIX ₂ (a ₂)	5:	lockIS ₂ (a ₂)
6:	lockIX ₂ (p ₅)	6:	lockS ₂ (p ₅)
7:	lockX ₂ (s ₉)	7:	lockIX ₃ (D)
8:	lockIS ₃ (D)	8:	lockIX ₃ (a ₁)
9:	lockIS ₃ (a ₃)	9:	lockX ₃ (p ₁)
10:	lockS ₃ (p ₆)	10:	lockIS ₃ (a ₂)
11:	lockIX ₁ (D)	11:	lockIS ₃ (p ₅)
12:	lockIX ₁ (a ₂)	12:	lockS ₃ (s ₈)
13:	lockX ₁ (p ₃)	13:	lockIX ₂ (D)
14:	lockIS ₃ (a ₁)	14:	lockIX ₂ (a ₂)
15:	lockIS ₃ (p ₂)	15:	lockIX ₂ (p ₃)
16:	lockS ₃ (s ₄)	16:	lockX ₂ (s ₅)
17:	lockIX ₂ (a ₁)	17:	lockIX ₁ (p ₃)
18:	lockIX ₂ (p ₂) (1)	18:	lockX ₁ (s ₆)
19:	lockIS ₄ (D)	19:	lockIX ₁ (a ₁)
20:	lockS ₄ (a ₁) (2)	20:	lockX ₁ (p ₁) (1)
21:	relS ₁ (p ₂)	21:	lockS ₂ (p ₃) (2)
22:	relIS ₁ (a ₁)	22:	lockIS ₄ (D)
23:	lockIX ₂ (p ₂)	23:	lockIS ₄ (a ₃)
24:	lockX ₂ (s ₃)	24:	lockIS ₄ (p ₆)
		25:	lockS ₄ (s ₁₀)
		26:	relS ₃ (s ₈)
		27:	relIS ₃ (p ₅)
		28:	relIS ₃ (a ₂)
		29:	lockIX ₃ (a ₂)
		30:	lockIX ₃ (p ₄) (3)

- (a)
- Probleme bei Sperranforderungen:
 - (1): Die IX-Sperre kann wegen der Lesesperre von T_1 auf p_2 nicht gewährt werden.
 - (2): Die Lesesperre kann wegen der IX-Sperre von T_2 auf a_1 nicht gewährt werden.
 - Nein, es kommt zu keinem Deadlock. Die Transaktion T_4 wartet auf eine Sperre die T_2 hält, die Transaktion T_2 kann aber weiterlaufen.
 - Die folgenden Freigaben sind nötig damit T_4 weiterlaufen kann: $\text{relX}_2(s_3) \rightarrow \text{relIX}_2(p_2) \rightarrow \text{relIX}_2(a_1)$. Anschließend erhält T_4 die gewünschte Lesesperre, und jede der Transaktionen hat alle Einträge der Sequenz abgearbeitet.
- (b)
- Probleme bei Sperranforderungen:

- (1): Die Schreibsperre für T_1 auf p_1 kann wegen der Schreibsperre von T_3 nicht gewährt werden.
- (2): Die Lesesperre für T_2 auf p_3 kann wegen der IX-Sperre von T_1 nicht gewährt werden.
- (3): Die IX-Sperre von T_3 auf p_4 kann wegen der Schreibsperre von T_1 nicht gewährt werden"
- Ja, es kommt zu einem Deadlock, da T_1 auf eine Sperre wartet, welche von T_3 gehalten wird, und T_3 auf eine Sperre wartet welche von T_1 gehalten wird. Da jedoch beide Transaktionen blockiert sind, wird keine der beiden ausgeführt werden, und somit kann keine der beiden Sperren freigegeben werden.

Aufgabe 7 (Zeitstempelbasiertes Sperrverfahren)

[3 Punkte]

Gegeben ist die folgende Sequenz von Elementaroperationen von vier Transaktionen T_1 , T_2 , T_3 und T_4 , welche auf drei Datenobjekte A , B und C zugreifen.

$\text{BOT}_2 \rightarrow \text{BOT}_3 \rightarrow w_2(A) \rightarrow w_3(A) \rightarrow \text{BOT}_1 \rightarrow r_2(A) \rightarrow r_2(B) \rightarrow w_1(B) \rightarrow \text{BOT}_4 \rightarrow r_2(B) \rightarrow \text{res?} \rightarrow w_4(C) \rightarrow r_1(A) \rightarrow r_2(C) \rightarrow w_4(B) \rightarrow c_2 \rightarrow c_3 \rightarrow \text{res?} \rightarrow c_1 \rightarrow c_4$

Dabei bezeichnet BOT_i den Beginn der Transaktion T_i , $r_i(X)$ eine Leseoperation (Transaktion T_i liest Datenobjekt X), $w_i(X)$ eine Schreiboperation (Transaktion T_i schreibt Datenobjekt X) und c_i den erfolgreichen Abschluss (commit) von Transaktion T_i . Einträge res? bedeuten, dass zu diesem Zeitpunkt eine Transaktion neu gestartet werden soll, falls es zu dieser Zeit eine Transaktion gibt welche zurückgesetzt aber noch nicht neu gestartet wurde (falls es mehrere solcher Transaktionen gibt, starten Sie jene neu, deren Zurücksetzen am weitesten zurück liegt). Anschließend an den Neustart führen Sie bitte alle Operationen der Transaktion bis zum aktuellen Zeitpunkt aus.

- (a) Wenden Sie die Regeln der Zeitstempel-Synchronisationsmethode auf den gegebenen Operationen an, um eine (nach dieser Methode) gültige Historie zu erhalten. Verwenden Sie jene in der Vorlesung vorgestellte Variante, welche nicht notwendigerweise rücksetzbare Historien erzeugt. D.h. Schreiboperationen werden sofort durchgeführt, und der Zugriff auf entsprechende Felder wird für andere Transaktionen ausschließlich über die Lese- und Schreibzeitstempel geregelt (d.h. die Transaktionen werden entweder abgebrochen, oder erhalten Zugriff, Transaktionen werden nicht blockiert).

Sie brauchen sich im Falle eines Zurücksetzens nicht darum kümmern, ob bereits ausgeführte Aktionen anderer Transaktionen von dem Zurücksetzen betroffen sind: es wird nur die aktuelle Transaktionen zurückgesetzt, und kein kaskadierendes Zurücksetzen durchgeführt.

Im Falle eines Zurücksetzens bleiben sowohl der Lese- als auch der Schreibzeitstempel unverändert.

Geben Sie die daraus entstehende Historie bitte in Form einer Tabelle mit den folgenden Spalten an:

#	Aktion	rTS(A)	wTS(A)	rTS(B)	wTS(B)	rTS(C)	wTS(C)
---	--------	--------	--------	--------	--------	--------	--------

Die Spalte # soll eine fortlaufende Nummer beinhalten. Für die Spalte **Aktion** verwenden Sie bitte die weiter oben beschriebene (und in der Angabe verwendete) Schreibweise für

BOT-Einträge, COMMIT-Einträge, sowie Lese- und Schreibeinträge. Wird eine Transaktion zurückgesetzt, so verwenden Sie bitte reset_i für den Eintrag. In den restlichen Spalten geben Sie bitte die Werte der Lese- und Schreibzeitstempel jeweils nach der Ausführung der entsprechenden Aktion an.

Lösung:

#	T	rTS(A)	wTS(A)	rTS(B)	wTS(B)	rTS(C)	wTS(C)
1	BOT ₂	0	0	0	0	0	0
2	BOT ₃	0	0	0	0	0	0
3	$w_2(A)$	0	1	0	0	0	0
4	$w_3(A)$	0	2	0	0	0	0
5	BOT ₁	0	2	0	0	0	0
6	$\text{reset}_2 [r_2(A)]$	0	2	0	0	0	0
7	$w_1(B)$	0	2	0	5	0	0
8	BOT ₄	0	2	0	5	0	0
9	BOT ₂	0	2	0	5	0	0
10	$w_2(A)$	0	9	0	5	0	0
11	$r_2(A)$	9	9	0	5	0	0
12	$r_2(B)$	9	9	9	5	0	0
13	$r_2(B)$	9	9	9	5	0	0
14	$w_4(C)$	9	9	9	5	0	8
15	$\text{reset}_1 [r_1(A)]$	9	9	9	5	0	8
16	$r_2(C)$	9	9	9	5	9	8
17	$\text{reset}_4 [w_4(B)]$	9	9	9	5	9	8
18	c_2	9	9	9	5	9	8
19	c_3	9	9	9	5	9	8
20	BOT ₁	9	9	9	5	9	8
21	$w_1(B)$	9	9	9	20	9	8
22	$r_1(A)$	20	9	9	20	9	8
23	c_1	20	9	9	20	9	8

(b) Ist die erzeugte Historie rücksetzbar?

Lösung:

Nein. Obwohl das verwendete Verfahren nicht garantiert, dass die erzeugten Historien rücksetzbar sind, könnte es ja in konkreten Fällen trotzdem sein, dass die resultierenden Historien rücksetzbar sind. Dies ist hier jedoch nicht der Fall: In Schritt 16 liest T_2 das Datum C , welches zuvor von T_4 geschrieben wurde. Rücksetzbarkeit verlangt nun, dass T_2 nicht vor T_4 committed. Genau das passiert aber in Schritt 18: T_2 macht ihr commit, obwohl T_4 zu diesem Zeitpunkt noch nicht committed wurde.

(c) Verwenden Sie nun die in der Vorlesung vorgestellte Variante der Zeitstempel-Synchronisationsmethode welche *strikte Historien* erzeugt. (Verwenden Sie die Methode mittels *dirty bit*.) Im Falle eines Zurücksetzens sollen diesmal – falls anwendbar – auch die Schreibzeitstempel zurückgesetzt werden. Lesezeitstempel bleiben wiederum unverändert.

Um in der Tat alle Deadlocks auszuschließen ist eine leichte Erweiterung des Algorithmus gegenüber der in der Vorlesung vorgestellten Version nötig: Wann immer die Zeitstempel eines Datensatzes geändert werden, so soll die dazugehörige Warteschlange überprüft werden, ob dort Transaktionen auf den Zugriff warten, welche auf Grund der neuen Zeitstempel nicht auf den Datensatz zugreifen dürfen. Ist dies der Fall sollen die entsprechenden wartenden Transaktionen abgebrochen werden.

Geben Sie die resultierende Historie wiederum in einer wie in Aufgabe (a) beschriebenen Tabelle aus, jedoch mit den folgenden beiden zusätzlichen Informationen:

- Geben Sie für jedes der Felder A , B und C nun zusätzlich noch aus, ob das dirty bit gesetzt ist oder nicht. D.h. verwenden Sie eine Tabelle mit den Spalten:

#	Aktion	rTS(A)	wTS(A)	d(A)	rTS(B)	wTS(B)	d(B)	rTS(C)	wTS(C)	d(C)
---	--------	------------	------------	----------	------------	------------	----------	------------	------------	----------

- Wenn eine Transaktion T_i blockiert, ergänzen Sie die Historie mit einem Eintrag block_i .

Lösung:

#	T	rTS(A)	wTS(A)	d(A)	rTS(B)	wTS(B)	d(B)	rTS(C)	wTS(C)	d(C)
1	BOT ₂	0	0		0	0		0	0	
2	BOT ₃	0	0		0	0		0	0	
3	$w_2(A)$	0	1	✓(2)	0	0		0	0	
4	block ₃ [$w_3(A)$]	0	1	✓(2)	0	0		0	0	
5	BOT ₁	0	1	✓(2)	0	0		0	0	
6	$r_2(A)$	1	1	✓(2)	0	0		0	0	
7	$r_2(B)$	1	1	✓(2)	1	0		0	0	
8	$w_1(B)$	1	1	✓(2)	1	5	✓(1)	0	0	
9	BOT ₄	1	1	✓(2)	1	5	✓(1)	0	0	
10	reset ₂ [$r_2(B)$]	1	0		1	5	✓(1)	0	0	
11	(cont. ₃) $w_3(A)$	1	2	✓(3)	1	5	✓(1)	0	0	
12	BOT ₂	1	2	✓(3)	1	5	✓(1)	0	0	
13	block ₂ [$w_2(A)$]	1	2	✓(3)	1	5	✓(1)	0	0	
14	$w_4(C)$	1	2	✓(3)	1	5	✓(1)	0	9	✓(4)
15	block ₁ [$r_1(A)$]	1	2	✓(3)	1	5	✓(1)	0	9	✓(4)
16	block ₄ [$w_4(B)$]	1	2	✓(3)	1	5	✓(1)	0	9	✓(4)
17	c_3	1	2		1	5	✓(1)	0	9	✓(4)
18	(cont. ₂) $w_2(A)$	1	12	✓(2)	1	5	✓(1)	0	9	✓(4)
19	reset ₁	1	12	✓(2)	1	0		0	9	✓(4)
20	$r_2(A)$	12	12	✓(2)	1	0		0	9	✓(4)
21	$r_2(B)$	12	12	✓(2)	12	0	✓(2)	0	9	✓(4)
22	$r_2(B)$	12	12	✓(2)	12	0	✓(2)	0	9	✓(4)
23	block ₂ [$r_2(C)$]	12	12	✓(2)	12	0	✓(2)	0	9	✓(4)
24	(cont. ₄)	12	12	✓(2)	12	0	✓(2)	0	9	✓(4)
..	reset ₄ [$w_4(B)$]	12	12	✓(2)	12	0	✓(2)	0	0	
25	(cont. ₂) $r_2(C)$	12	12	✓(2)	12	0	✓(2)	12	0	✓(2)
26	c_2	12	12		12	0		12	0	
27	b_1	12	12		12	0		12	0	
28	$w_1(B)$	12	12		12	27	✓(1)	12	0	
29	$r_1(A)$	27	12	✓(1)	12	27	✓(1)	12	0	
30	c_1	27	12		12	27		12	0	

(d) Ist die erzeugte Historie rücksetzbar?

Lösung:

Ja. Die verwendete Synchronisationsmethode garantiert die Erzeugung strikter Historien. Da jede strikte Historie auch rücksetzbar ist, ist das Ergebnis rücksetzbar.

Verwenden Sie zum Lösen der Aufgabe bitte die folgenden Annahmen bzw. Konventionen:

- Nehmen Sie an, dass die Anfangswerte für die Werte `readTS` und `writeTS` aller drei Felder A , B und C jeweils 0 sind.
- Als Zeitstempel für die Transaktionen nehmen Sie bitte die # ihres des jeweiligen BOT-Eintrags.

- Wenn Sie am Ende der Historie angekommen sind, und nicht alle rückgestzten Transaktionen neu gestartet wurden (kein passender *res?* Eintrag), dann wird diese Transaktion einfach nicht erneut durchgeführt.

Aufgabe 8 (Transaktionen in SQL)

[0.5 Punkte]

Betrachten Sie das folgende Relationenschema eines Unternehmens, in welchem durchgeführte Arbeiten und deren Verrechnung gespeichert werden:

(Primärschlüssel sind unterstrichen, Fremdschlüssel sind kursiv geschrieben).

Tasks (task_id, by_person, for_person, task_description, amount)
 paid (*task_id*: Tasks.task_id)
 approved (*task_id*: Tasks.task_id, approval)

Bestimmen Sie für jedes der unten aufgeführten Szenarien das jeweils *niedrigste* Isolation Level welches das geforderte Grad an Isolation bietet. Beschreiben Sie außerdem, ob die Transaktionen bei der angegebenen Verzahnung in dem jeweiligen Isolation Level wie gewünscht ablaufen können, oder ob es zu Problemen kommt. Aus Platzgründen wird unten nur eine Skizze der Abfragen angezeigt. Die vollständigen Abfragen finden Sie in den zur Verfügung gestellten SQL Dateien.

- (a) *Beschreibung*: Alice und Bob legen jeweils eine neue Tätigkeit an, welche sie der jeweils anderen Person in Rechnung stellen. Dabei ziehen sie von den eigentlichen Kosten jenen Wert ab, den sie der anderen Person noch schulden. Für das Anlegen neuer Aufgaben gibt es die strikte Bestimmung der Firma, dass Tätigkeiten nicht parallel angelegt werden dürfen, sondern immer eine klare Reihenfolge der Aktionen nachvollziehbar sein muss.

Historie:

Alice	Bob
	BEGIN; SET TRANSACTION ISOLATION LEVEL _____
BEGIN; SET TRANSACTION ISOLATION LEVEL _____	
SELECT sum(amount) FROM Tasks('Bob', 'Alice');	
INSERT INTO paid (SELECT task_id FROM Tasks('Bob', 'Alice'));	
INSERT INTO Tasks VALUES (9, 'Alice', 'Bob', 'work', 211.20);	
	SELECT sum(amount) FROM Tasks('Alice', 'Bob');
	INSERT INTO paid (SELECT task_id FROM Tasks('Alice', 'Bob'));
	INSERT INTO Tasks VALUES (8, 'Bob', 'Alice', 'work', 211.20);
COMMIT;	
	COMMIT;

wobei Tasks(a,b) für folgende Zeichenkette steht:

```
Tasks WHERE by_person=a AND for_person=b AND task_id NOT IN (SELECT task_id
FROM paid)
```


Lösung:Isolation Level: **SERIALIZABLE**

(Datenbank: PostgreSQL 11.5) Bei diesem Isolation Level kommt es zu einem Fehler wenn die Transaktion in Szenario1-b versucht zu committen.

Das Problem ist, dass die beiden Transaktionen nicht serialisierbar sind. Der Grund dafür ist, dass beide Transaktionen einen neuen Eintrag in die Tabelle **paid** einfügen, der Wert in diesem neuen Eintrag jedoch vom Inhalt der Tabelle **paid** abhängt. Das heißt, sowohl Bob als auch Alice würden einen anderen Wert einfügen, wenn Sie das Tupel welche die jeweils andere Person erstellt hat sehen könnten. Da die Transaktionen parallel ablaufen, sieht jedoch weder Bob den Wert welchen Alice einfügt, noch umgekehrt. Im Fall einer seriellen Ausführung, hätte jedoch die Person, deren Transaktion als zweites ausgeführt wird, den von der anderen Person eingefügten Wert sehen müssen. Daher handelt es sich um keine serialisierbare Transaktion, was vom DBMS erkannt wird, und die Transaktion welche als zweites abschließt, und daher den von der anderen Transaktion eingefügten Wert berücksichtigen, wird abgebrochen.

Die Fehlermeldung lautet:

```
psql:Szenario1-b-muster.sql:23: ERROR: could not serialize access due to read/write dependencies among transactions
```

```
DETAIL: Reason code: Canceled on identification as a pivot, during commit attempt.
```

- (b) *Beschreibung:* Bevor Arbeiten einem Kunden in Rechnung gestellt werden können, muss die entsprechende Tätigkeit erst von entsprechenden Vorarbeiter_innen genehmigt werden. Die Genehmigungen werden im Rahmen von Transaktionen in der Datenbank eingetragen. Diese Transaktionen sollen jedoch die Möglichkeiten Transaktionen parallel auf der Datenbank auszuführen so wenig wie möglich beschränken. Darüber hinaus ist es kein Problem, wenn (abgeschlossene) parallele Änderungen an der Auftragslage im Zuge dieser Transaktion sichtbar sind.

Historie:

Supervisor	Parallele Anfragen
BEGIN;	
SET TRANSACTION ISOLATION LEVEL _____	
SELECT * FROM Tasks WHERE ¬approved	
	INSERT INTO Tasks VALUES (12, ...);
	INSERT INTO Tasks VALUES (13, ...);
SELECT * FROM Tasks NATURAL JOIN paid	
SELECT * FROM Tasks NATURAL JOIN paid WHERE ¬paid	
INSERT INTO approved VALUES (5, ...);	
SELECT * FROM Tasks WHERE ¬approved	
	INSERT INTO Tasks VALUES (10, ...);
	INSERT INTO Tasks VALUES (11, ...);
SELECT * FROM Tasks WHERE ¬approved	
COMMIT;	

wobei ¬approved und ¬paid für die Bedingungen
 task_id NOT IN (SELECT task_id FROM approved) bzw.
 task_id NOT IN (SELECT task_id FROM paid) stehen.

Lösung:

Isolation Level: READ COMMITTED

Die Transaktion kann wie gewünscht durchgeführt werden.

- (c) *Beschreibung:* Die Buchhaltung möchte Rechnungen an jene Kunden mit einer bestätigten aber unbezahlten Leistung ausstellen. Für diese Tätigkeit ist es wichtig, während der gesamten Dauer mit einem einheitlichen Datenbestand zu arbeiten, d.h. einem Datenbestand welcher sich während der Rechnungsausstellung nicht durch parallele Schreibzugriffe verändert.

Historie:

Supervisor	Parallele Anfragen
BEGIN;	
SET TRANSACTION ISOLATION LEVEL _____	
SELECT * FROM open	
	INSERT INTO Tasks VALUES (12, ...);
	INSERT INTO Tasks VALUES (13, ...);
SELECT sum(amount) FROM open	
	INSERT INTO Tasks VALUES (10, ...);
	INSERT INTO Tasks VALUES (11, ...);
SELECT count(task_id) FROM open	
COMMIT;	

wobei `open` für die Bedingung

`tasks NATURAL JOIN approved WHERE task_id NOT IN (SELECT task_id FROM paid)` steht.

Lösung:

Das korrekte Isolation Level kann hier selbst unter den “üblichen” DBMS variieren. Prinzipiell ist ein Isolation Level zu wählen, bei dem Phantom Reads ausgeschlossen sind.

Entsprechend dem SQL-Standard ist dies nur im Isolation Level `SERIALIZABLE` gewährleistet.

Wie in der Vorlesung erwähnt können jedoch z.B. in PostgreSQL bereits im Isolation Level `REPEATABLE READ` das Phantomproblem nicht mehr auftreten. In diesem Fall würde also diese Wahl bereits ausreichen, um das gewünschte Verhalten zu erlangen.

Die Transaktionen sollten auf jeden Fall wie gewünscht durchgeführt werden können.

Hinweis: Sie können die Szenarien mit Hilfe der SQL Dateien auch einfach auf einem DBMS ausprobieren (die Dateien sind für Postgres geschrieben, können aber, evtl. mit kleineren Änderungen, auch auf anderen DBMS ausprobiert werden). Auf Postgres, gehen Sie dabei wie folgt vor:

1. Öffnen Sie mittels `psql` eine Datenbankkonsole.
2. Öffnen Sie in einem anderen Terminal mittels `psql` eine weitere Datenbankkonsole.
3. Laden Sie einem Terminal das Szenario mittels `\i Szenario1-a.sql` (wobei Sie “1” durch die gewünschte Zahl ersetzen)

4. Laden Sie im anderen Terminal das parallele Szenario mittels `iSzenario1-b.sql` (wobei Sie "1" durch die gewünschte Zahl ersetzen)
5. Die Ausführung der SQL-Befehle sollte nun in beiden Konsolen mit der Meldung **Press Enter to continue** (i) unterbrochen sein.
6. Drücken Sie jeweils in der Konsole in der `i` den kleineren Wert hat [Enter].

Sollte sich das Ergebnis bei einem gewählten Isolation Level von Ihrer Erwartung unterscheiden, so können Sie dies gerne dokumentieren. In dem Fall geben Sie bitte an, auf welchem DBMS Sie die Transaktionen ausgeführt haben.

Sie können das Beispiel entweder auf einer Datenbank bei sich laufen lassen, es steht Ihnen aber auch unser Übungsserver `bordo` zur Verfügung. Zu diesem können Sie sich per `ssh` verbinden und haben anschließend mittels `psql` Zugriff auf eine PostgreSQL Datenbank. Weitere Informationen darüber, wie Sie sich am Server und der Datenbank einloggen können finden Sie in TUWEL.

Lösung: