3. Übungsblatt (WS 2022) – Musterlösung

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 19.12.	**12:00 Uhr**:	Upload der Abgabe über TUWEL
ab 9.01.	20:00 Uhr:	Korrektur und Feedback in TUWEL verfügbar

Weitere Fragen – TUWEL Forum

Sie können darüber hinaus das TUWEL Forum verwenden, sollten Sie inhaltliche oder organisatorische Fragen haben. **Posten Sie auf keinen Fall (partielle) Lösungen im Forum!**

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 ausführte 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
$[\#i, T_j, P_X, X{+}{=}d_1, X{-}{=}d_2, \#k]$,
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 d_1 vergrößert werden müsste und beim *Undo* X um d_2 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.

Hinweis: Sie können die vorgefertigte Tabelle ausfüllen, um die Lösung einzutragen.Alternativ können Sie auch gerne zwei eigene Tabellen mit (a) Ergebnissen bzw. (b) Log oder CLR-Einträgen anlegen. Denken Sie in jedem Falle daran Log / CLR-Eintrag korrekt zu unterscheiden.

Aufgabe 1 (Logging) [1 Punkt]

Betrachten Sie die in Abbildung 1 (auf Seite) angegebene Historie der drei Transaktionen T_1, T_2 und T_3.

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 $r_i(\Gamma, \gamma)$ eine Leseoperation (der Wert des Feldes Γ wird aus der Datenbasis in eine lokale Variable γ gelesen) und $w_i(\Gamma, \gamma)$ 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.

Nehmen Sie schlussendlich an, dass zu Beginn (Zeile 1) der relevante Datenbestand in der

Datenbank wie folgt aussieht:

$$A\colon 47 \qquad B\colon 11 \qquad C\colon 8 \qquad D\colon 15$$

(a) Geben Sie für jede Zeile der Historie, in welcher entweder der Wert eines Feldes, oder einer lokalen Variable 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 `abort`s die Zeilennummer 7). In der Abbildung ist die Lösung für Zeilen 1–2 bereits ausgefüllt.

(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 insbesonders 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 benützen 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_2.

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 LaTeXVorlage verwenden finden Sie dort bereits einen Vorschlag zur Formatierung. Sie können die Werte gerne in die vorgefertigte Tabelle neben der Abbildung einfügen in der bereits die Lösung für die Zeilen 1–2 eingetragen ist.)

	Angabe: History			Lösg. Aufg. (a)	Lösung Aufgabe (b)					
	T_1	T_2	T_3		LSN	TA	Page	Redo	U/PL	PL/U
1		BOT			[#1	T_2	BOT			#0]
2		$r_2(A, a_2)$		$a_2 = 47$				—		
3		$r_2(B, b_2)$		$b_2 = 11$				—		
4	BOT				[#2	T_1	BOT			#0]
5	$r_1(C, c_1)$			$c_1 = 8$				—		
6		$w_2(D, 2 \cdot a_2 + b_2)$		$D_{=2\cdot 47+11}^{=2\cdot a_2+b_2} = 105$	[#3	T_2	P_D	$D+=90$	$D-=90$	#1]
7		abort		$D = 15$	⟨#4	T_2	P_D	$D-=90$	#3	#1⟩
					⟨#5	T_2	BOT		#4	⟩
8	$r_1(D, d_1)$			$d_1 = 15$				—		
9			BOT		[#6	T_3	BOT			#0]
10			$r_3(A, a_3)$	$a_3 = 47$				—		
11	$w_1(A, c_1 + 2 \cdot d_1)$			$A_{=8+2\cdot 15}^{=c_1+2\cdot d_1} = 38$	[#7	T_1	P_A	$A-=9$	$A+=9$	#2]
12			$w_3(A, a_3 + 4)$	$A_{=47+4}^{=a_3+4} = 51$	[#8	T_3	P_A	$A+=13$	$A-=13$	#6]
13	$r_1(B, b_1)$			$b_1 = 11$				—		
14			$w_3(C, a_3 - 7)$	$C = a_3 - 7 = 40$	[#9	T_3	P_C	$C+=32$	$C-=32$	#8]
15	$w_1(D, c_1 - 40)$			$D_{=8-40}^{=c_1-40} = -32$	[#10	T_1	P_D	$D-=47$	$C+=47$	#7]
16			$r_3(B, b_3)$	$b_3 = 11$				—		
17			$w_3(D, 27)$	$D = 27$	[#11	T_3	P_D	$D+=59$	$D-=59$	#9]
18			$w_3(C, b_3 + a_3)$	$C_{=11+47}^{=b_3+a_3} = 58$	[#12	T_3	P_C	$C+=18$	$C-=18$	#11]
19			commit		[#13	T_3	commit			#12]
20	commit				[#14	T_1	commit			#10]

Abbildung 1: Historie für Aufgabe 1. U... Undo, PL... PrevLSN, PL, UN...UndoNextLSN. Klammern (für Log/CLR) nicht vergessen.

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, P_C und P_D dargestellt.

Log-Einträge

[#1, T_2, `BOT`, #0]
[#2, T_2, P_B, -4,+4, #1]
[#3, T_2, P_A, +5, -5, #2]
[#4, T_3, `BOT`,#0]
[#5, T_3, P_C, +9, -9, #4]
[#6, T_1, `BOT`, #0]
[#7, T_2, P_B, +7, -7, #3]
[#8, T_2, `COMMIT`, #7]
[#9, T_3, P_A, -5, +5, #5]
[#10, T_1, P_B, -14, +14, #6]
⟨#11, T_3, P_A, +5, #9, #5⟩
[#12, T_1, P_D, -2, +2, #10]
⟨#13, T_3, P_C, -9, #11, #4⟩
[#14, T_1, P_B, -8, +8, #12]
[#15, T_1, P_D, +6, -6, #14]

Seiten im Hintergrundspeicher

P_A	LSN: #3
$A = 27$	

P_B	LSN: #2
$B = 4$	

P_C	LSN: #5
$C = 29$	

P_D	LSN: #0
$D = 18$	

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: 27; B: -11; C: 20; D: 22

Nur die Redos ausführen, die eine höhere LSN haben, als die der jeweiligen Seite.
Lösungsweg:

LSN	#3	#9	#11	
A =	27	-5	+5	= 27

LSN	#2	#7	#10	# 14	
B =	4	+7	-14	-8	= -11

LSN	#5	#13	
C =	29	-9	=20

LSN	#0	#12	#15	
D =	18	-2	+6	=22

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

Lösung:

Log: ⟨`LSN, TA, PageID, Redo, PrevLSN, UndoNextLSN`⟩
⟨#16, T_1, P_D, D-=6, #15, #14⟩
⟨#17, T_1, P_B, B+=8, #16, #12⟩
⟨#18, T_1, P_D, D+=2, #17, #10⟩
⟨#19, T_1, P_B, B+=14, #18, #6⟩
⟨#20, T_1, `BOT`, #20⟩
⟨#21, T_3, `BOT`, #13⟩

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

Lösung:

A: 27; B: 11; C: 20; D: 18

Lösungsweg:

LSN	#15
A =	27

LSN	#15	#17	#21	
B =	-11	+8	+14	= 11

LSN	#15
C =	20

LSN	#15	#16	#19	
D =	22	-6	+2	= 18

Hinweis LSN oben sind nur als Hinweis auf den letzten Log-Eintrag gedacht (Abschluss Redo, nicht exakter Access LSN).

Aufgaben: Mehrbenutzersynchronisation

Aufgabe 3 (Eigenschaften von Transaktionen) [3 Punkte]

Gegeben ist die Menge $\mathcal{T}_a$ und $\mathcal{T}_b$ an Transaktionen sowie die dazugehörige Historie $\mathcal{H}_a$ und $\mathcal{H}_b$ mit den gegebenen Folge von Elementaroperationen.

(a) $\mathcal{T}_a = \{T_1, T_2, T_3, T_4\}$

$\mathcal{H}_a = b_1 \rightarrow w_1(B) \rightarrow b_3 \rightarrow r_1(D) \rightarrow r_3(B) \rightarrow b_2 \rightarrow r_1(A) \rightarrow r_1(C) \rightarrow r_2(A) \rightarrow b_4 \rightarrow w_1(D) \rightarrow c_1 \rightarrow w_3(B) \rightarrow w_2(D) \rightarrow w_2(A) \rightarrow w_4(D) \rightarrow w_3(C) \rightarrow c_3 \rightarrow w_4(B) \rightarrow r_2(C) \rightarrow c_4 \rightarrow c_2$.

(a1) Zeichnen Sie den Serialisierbarkeitsgraphen SG($\mathcal{H}$).

(a2) 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 $T_i \rightarrow T_j$ (sofern sie tatsächlich Teil des Graphen sind) *sämtliche* Konfliktoperationen an, welche verlangen dass diese Kanten Teil des Graph sind.

(a3) 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.

(a4) Geben Sie für die Historie $\mathcal{H}$ die Leseabhängigkeiten zwischen den Transaktionen an (d.h., geben Sie an 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.

(a5) Bestimmen Sie, welche der folgenden Eigenschaften die Historie $\mathcal{H}$ besitzt:

- Rücksetzbar
- Vermeidet kaskadierendes Rücksetzen
- Strikt

Begründen Sie jeweils Ihre Antwort.

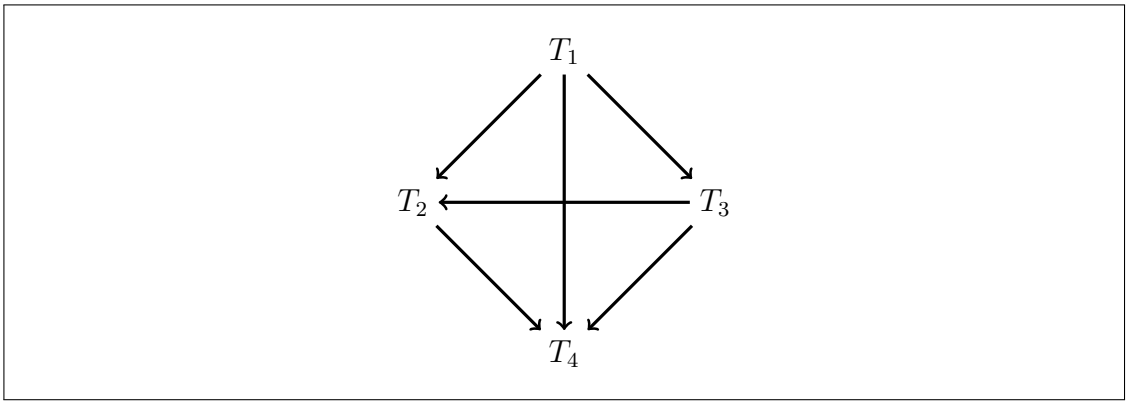
(b) Bestimmen Sie, ob die folgende Historie (1) konfliktserialisierbar ist und (2) welche der drei Eigenschaften (i) Rücksetzbar (ii), vermeidet kaskadierendes Rücksetzen, (iii) strikt die Historie erfüllt.

$\mathcal{T}_b = \{T_1, T_2, T_3, T_4\}$

$\mathcal{H}_b = b_3 \rightarrow r_3(A) \rightarrow b_1 \rightarrow b_2 \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow b_4 \rightarrow r_4(D) \rightarrow r_2(B) \rightarrow w_4(D) \rightarrow w_1(B) \rightarrow r_1(D) \rightarrow w_1(D) \rightarrow r_1(C) \rightarrow w_4(C) \rightarrow r_3(C) \rightarrow r_2(D) \rightarrow w_2(D) \rightarrow c_4 \rightarrow c_1 \rightarrow c_3 \rightarrow c_2$

Lösung:

(a1) **Serialisierbarkeitsgraph:**



(a2) **"Begründung" für die Kanten**
(Es sollten alle Konfliktoperationen angegeben werden.):

Hilfestellung (Reihenfolge der Zugriffe auf)
A: $r_1(A) \to r_2(A) \to w_2(A)$, **B:** $w_1(B) \to r_3(B) \to w_3(B) \to w_4(B)$,
C: $r_1(C) \to w_3(C) \to r_2(C)$, **D:** $r_1(D) \to w_1(D) \to w_2(D) \to w_4(D)$.

$T_1 \to T_3$
- $w_1(B) \to r_3(B)$
- $w_1(B) \to w_3(B)$
- $r_1(C) \to w_3(C)$

$T_1 \to T_4$
- $w_1(B) \to w_4(B)$
- $r_1(D) \to w_4(D)$
- $w_1(D) \to w_4(D)$

$T_3 \to T_2$
- $w_3(C) \to r_2(C)$

$T_1 \to T_2$
- $r_1(A) \to w_2(A)$
- $r_1(D) \to w_2(D)$
- $w_1(D) \to w_2(D)$

$T_2 \to T_4$
- $w_2(D) \to w_4(D)$

$T_3 \to T_4$
- $r_3(B) \to w_4(B)$
- $w_3(B) \to w_4(B)$

(a3) **Konflikt-Serialisierbarkeit:**

Ja, die Historie **ist** konfliktserialisierbar: Der Serialisierbarkeitsgraph enthält keine Zyklen.

Eine mögliche äquivalente serielle Ausführungsreihenfolge wäre

$$T_1 \text{ vor } T_3 \text{ vor } T_2 \text{ vor } T_4$$

(a4) **Lese-Abhängigkeiten:**

- T_2 liest von T_3: $(w_3(C), r_2(C))$
- T_3 liest von T_1: $(w_1(B), r_3(B))$

(a5) **Klassifikation der Historie:**

- **Rücksetzbar:**

  Eine Historie ist rücksetzbar wenn jede Transaktion von der gelesen wird ihr `COMMIT` *vor dem* `COMMIT` *der jeweils lesenden Transaktionen haben.*

  Ja, die Historie *ist* rücksetzbar.

  - Transaktion T_2 liest von T_3 und das `COMMIT` von T_2 kommt vor dem `COMMIT` von T_3.
  - Transaktion T_3 liest von T_1 und das `COMMIT` von T_3 kommt nach dem `COMMIT` von T_1.

- **Vermeidet Kaskadierendes Rücksetzen:**

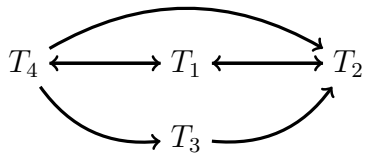
  Um kaskadierendes Rücksetzen zu vermeiden ist es notwendig dass bei jeder Leseoperation ein Wert gelesen wird, welcher von einer bereits erfolgreich abgeschlossenen (commited) Transaktion geschrieben wurde.

  Nein. Dies wird in der vorliegenden Historie verletzt. Ein solches Beispiel ist: $r_3(B)$ liest den Wert $w_1(B)$, obwohl T_1 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 $r_4(A)$ und $w_3(A)$ auch ein Beispiel dafür, dass die Historie nicht strikt ist.

(b)



Serialisierbarkeitsgraph (Nicht! Teil der Fragestellung):

(b1) **Konflikt-Serialisierbarkeit:**

Nein, Die Historie ist *nicht konfliktserialisierbar*. Es existiert (zumindest) ein Zyklus, siehe Serialisierbarkeitsgraph.

(b2) **Klassifikation der Historie:**

Leseabhängigkeiten (nicht gefordert)

- T_3 liest von T_4: $(w_4(C), r_3(C))$
- T_2 liest von T_1: $(r_2(D), w_1(D))$
- T_1 liest von T_4: $(w_4(D), r_1(D))$

(b2i) **Rücksetzbar:**

Ja, die Historie ist rücksetzbar. Es lesen T_3 von T_4, T_2 von T_1 und T_1 von T_4. Die `COMMIT` Reihenfolge ist: c_4,c_1,c_3, c_2. Damit ist die Bedingung für rücksetzbar Historien erfüllt.

(b2ii) **Vermeidet kaskadierendes Rücksetzen:**

Nein, die Historie vermeidet nicht kaskadierendes Rücksetzen. Beispielsweise liest T_3 von T_4, aber das `COMMIT` von T_4 ist erst nach dem Lesezugriff auf Datum C

(b2iii) **Strikte Historie:**

Nein. Die Historie ist nicht strikt. Das ergibt such zum einen daraus, dass die Historie kaskadierendes Rücksetzen nicht vermeidet und auch, beispielsweise dadurch, dass $w_3(A) \rightarrow w_2(A)$ gilt, aber dazwischen kein `COMMIT` von T_3 erfolgt ist.

Aufgabe 4 (Sperren und Deadlocks) [2.5 Punkte]

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

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

(a) Nehmen Sie an, ein DBMS erhält die angegebene Folge von Sperranforderungen für die Transaktionen T_1, T_2, T_3 und T_4, 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 $\texttt{grantS}_i(O)$ bzw. $\texttt{grantX}_i(O)$ um anzugeben, dass eine Lese- bzw. Schreibsperre auf dem Datenobjekt O gewährt wurde, verwenden Sie $\texttt{wait}(i)$ um anzuzeigen, dass eine Transaktion angehalten wurde um auf eine Sperre zu warten, verwenden Sie $\texttt{relS}_i(O)$ bzw. $\texttt{relX}_i(O)$ um anzugeben, dass eine Lese- bzw. Schreibsperre auf dem Datenobjekt O freigegeben wurde (als Reaktion auf ein $\texttt{rel}_i(O)$), und $\texttt{resume}(i)$ 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

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

von Sperranforderungen zweier Transaktionen T_1, T_2.

Wir erhalten die Liste

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

Lösung:

1:	$\texttt{lockX}_3(A)$
2:	$\texttt{grantX}_3(A)$
3:	$\texttt{lockS}_1(C)$
4:	$\texttt{grantS}_1(C)$
5:	$\texttt{lockX}_2(B)$
6:	$\texttt{grantX}_2(B)$
7:	$\texttt{lockX}_2(C)$
8:	$\texttt{wait}(2)$
9:	$\texttt{lockS}_4(A)$

10:	$\texttt{wait}(4)$
11:	$\texttt{lockX}_1(C)$
12:	$\texttt{grantX}_1(C)$
14:	$\texttt{lockS}_3(B)$
15:	$\texttt{wait}(3)$
16:	$\texttt{lockS}_1(D)$
17:	$\texttt{grantS}_1(D)$

(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:

	A	B	C	D
T_1			X	S
T_2		X	WX	
T_3	X	WS		
T_4	WS			

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

Lösung:

Wartegraph:

$$T_1 \longleftarrow T_2 \longleftarrow T_3 \longleftarrow T_4$$

(d) Geben Sie an, ob zum aktuellen Zeitpunkt ein Deadlock besteht. **Lösung: Nein**

(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_1 abgebrochen werden (d.h. alle Sperren von T_1 werden sofort freigegeben).

Lösung:
Nachdem derzeit kein Deadlock besteht, wird in Reihenfolge freigegeben.

$$\texttt{rel}_1(C) \rightarrow \texttt{relX}_1(C) \rightarrow \texttt{rel}_1(D) \rightarrow \texttt{relS}_1(D)$$

Auf Grund dieser Freigabe kann nun T_2 weiterlaufen, und wir erhalten:

$$\texttt{resume}(2) \rightarrow \texttt{grantX}_2(C)$$

Nun kann T_2 ihre Sperren wieder freigeben

$$\texttt{rel}_2(C) \rightarrow \texttt{relX}_2(C) \rightarrow \texttt{rel}_2(B) \rightarrow \texttt{relX}_2(B)$$

Dadurch kann Transaktion T_3 weiterlaufen

$$\texttt{resume}(3) \rightarrow \texttt{grantS}_3(B)$$

und anschließend alle Sperren freigeben

$$\texttt{rel}_3(B) \rightarrow \texttt{relS}_3(B) \rightarrow \texttt{rel}_3(A) \rightarrow \texttt{relX}_2(A).$$

Final kann T_4 weiterlaufen und Sperren freigeben:

$$\texttt{resume}(4) \rightarrow \texttt{grantS}_4(A) \rightarrow \texttt{rel}_4(A) \rightarrow \texttt{relS}_4(A).$$

(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).

$$\begin{array}{lccccccccccc} T_1\colon & r_1(D) & \to & r_1(A) & \to & r_1(B) & \to & r_1(A) & \to & r_1(C) & \to & c_1 \\ T_2\colon & r_2(A) & \to & r_2(C) & \to & w_2(A) & \to & r_2(D) & \to & w_2(B) & \to & c_2 \\ T_3\colon & r_3(D) & \to & w_3(D) & \to & r_3(B) & \to & r_3(C) & \to & w_3(B) & \to & c_3 \end{array}$$

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 $\texttt{lockS}_i(O)$ und $\texttt{lockX}_i(O)$ um das Anfordern einer Lese- bzw. Schreibsperre von Transaktion T_i auf das Objekt O anzuschreiben. Verwenden Sie bitte ebenso $\texttt{rel}_i(O)$ 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 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 so spät wie möglich beantragt, und so früh wie möglich wieder freigegeben.

  Hinweis: Diese Annahmen sind *zusätzlich* zum 2-Phasen Sperrprotokoll verstehen.

- *Verzahnung der Transaktionen:* Nehmen Sie an, dass jede Transaktion jeweils eine Operation (lesen, schreiben, commit) abarbeitet, und anschließend die nächste Transaktion ausgeführt wird. Beginnen Sie dabei mit der Transaktion 1. Im gegeben Fall wäre die Reihenfolge der Aktionen $r_1(A) \to r_2(C) \to r_3(B) \to r_1(B) \to \ldots$. 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. Dies 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 $\alpha_1, \alpha_2, \ldots$, und T_2 besteht aus Aktionen $\beta_1, \beta_2, \ldots$. Der normale Ablauf wäre $\alpha_1, \beta_1, \alpha_2, \beta_2, \ldots$. 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 $\alpha_3, \alpha_4, \alpha_5, \ldots$. Wird die Sperre nach Aktion α_5 wieder aufgegeben, so ist die weitere Abfolge $\alpha_5, \beta_3, \alpha_6, \beta_4, \ldots$.

Lösung:

#	T_1	T_2	T_3
1	$\texttt{lockS}_1(D)$		
2	$r_1(D)$		
3		$\texttt{lockS}_2(A)$	
4		$r_2(A)$	
5			$\texttt{lockS}_3(D)$
6			$r_3(D)$
7	$\texttt{lockS}_1(A)$		
8	$r_1(A)$		
9		$\texttt{lockS}_2(C)$	
10		$r_2(C)$	
11			$\texttt{lockX}_3(D)$
12	$\texttt{lockS}_1(B)$		
13	$r_1(B)$		
14		$\texttt{lockX}_2(A)$	
15	$r_1(A)$		
16	$\texttt{lockS}_1(C)$		
17	$r_1(C)$		
18	$\texttt{rel}_1(C)$		
19	$\texttt{rel}_1(A)$		
20	$\texttt{rel}_1(B)$		
21	$\texttt{rel}_1(D)$		
22	c_1		
23		$w_2(A)$	
24			$w_3(D)$
25		$\texttt{lockS}_2(D)$	
26			$\texttt{lockS}_3(B)$
27			$r_3(B)$
28			$\texttt{lockS}_3(C)$
29			$r_3(C)$
30			$\texttt{lockX}_3(B)$
31			$w_3(B)$
32			$\texttt{rel}_3(B)$
33			$\texttt{rel}_3(C)$
34			$\texttt{rel}_3(D)$
35			c_3
36		$r_2(D)$	
37		$\texttt{lockX}_2(B)$	
38		$w_2(B)$	
39		$\texttt{rel}_2(B)$	
40		$\texttt{rel}_2(D)$	
41		$\texttt{rel}_2(A)$	
42		$\texttt{rel}_2(C)$	
43		c_2	

Aufgabe 6 (Multi-Granularity Locking) [3 Punkte]

Betrachten Sie die Datenbasis-Hierarchie in Abbildung 3.

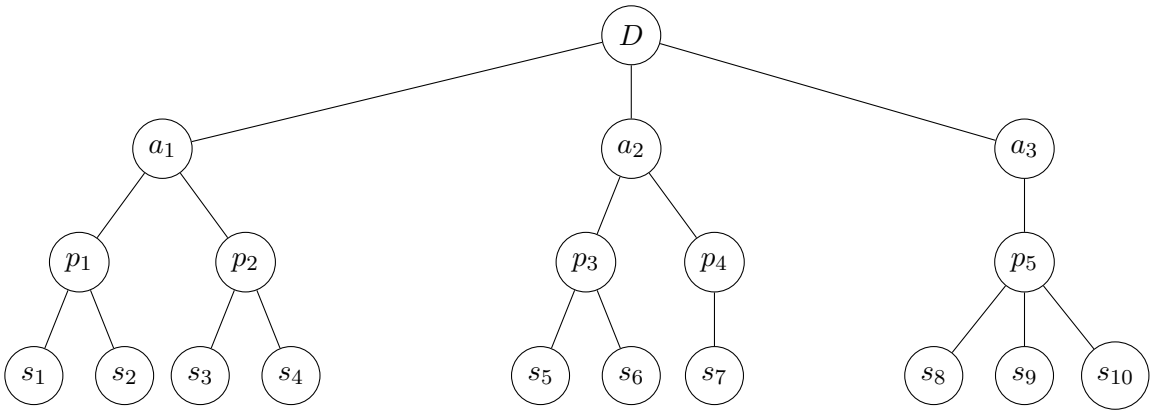


Abbildung 3: Datenbasis-Hierarchie zu Aufgabe 6

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) $\texttt{lockS}_1(p_4) \rightarrow \texttt{lockX}_2(s_9) \rightarrow \texttt{lockS}_3(p_2) \rightarrow \texttt{lockX}_1(p_3) \rightarrow \texttt{lockS}_3(s_4) \rightarrow \texttt{lockX}_2(s_6) \rightarrow \texttt{lockS}_4(a_1) \rightarrow \texttt{rel}_1(p_3)$

(b) $\texttt{lockX}_1(p_4) \rightarrow \texttt{lockS}_2(p_5) \rightarrow \texttt{lockX}_3(p_1) \rightarrow \texttt{lockS}_3(s_8) \rightarrow \texttt{lockX}_2(s_5) \rightarrow \texttt{lockX}_1(s_6) \rightarrow \texttt{lockX}_1(p_1) \rightarrow \texttt{lockS}_2(p_3) \rightarrow \texttt{lockS}_4(s_{10}) \rightarrow \texttt{rel}_3(s_8) \rightarrow \texttt{lockX}_3(s_7) \rightarrow \texttt{rel}_1(p_4)$

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

Bearbeiten Sie folgende Aufgabenstellungen für jede der beiden angegeben 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: $\texttt{lockIS}_i(O)$, $\texttt{lockIX}_i(O)$, $\texttt{lockS}_i(O)$ und $\texttt{lockX}_i(O)$ für das Anfordern einer `IS`-, `IX`-, `S`- bzw. `X`-Sperre von Transaktion T_i auf das Objekt O; $\texttt{relIS}_i(O)$, $\texttt{relIX}_i(O)$, $\texttt{relS}_i(O)$ und $\texttt{relX}_i(O)$ 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 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)

1:	$\texttt{lockIS}_1(D)$
2:	$\texttt{lockIS}_1(a_2)$
3:	$\texttt{lockS}_1(p_4)$
4:	$\texttt{lockIX}_2(D)$
5:	$\texttt{lockIX}_2(a_3)$
6:	$\texttt{lockIX}_2(p_5)$
7:	$\texttt{lockX}_2(s_9)$
8:	$\texttt{lockIS}_3(D)$
9:	$\texttt{lockIS}_3(a_1)$
10:	$\texttt{lockS}_3(p_2)$
11:	$\texttt{lockIX}_1(D)$
12:	$\texttt{lockIX}_1(a_2)$
13:	$\texttt{lockX}_1(p_3)$
14:	$\texttt{lockIS}_3(p_2)$
15:	$\texttt{lockS}_3(s_4)$
16:	$\texttt{lockIX}_2(D)$
17:	$\texttt{lockIX}_2(a_2)$
18:	$\texttt{lockIX}_2(p_3)$ (1)
19:	$\texttt{lockIS}_4(D)$
20:	$\texttt{lockS}_4(a_1)$
21:	$\texttt{relX}_1(p_3)$
22:	$\texttt{relX}_1(a_2)$

(b)

1:	$\texttt{lockIX}_1(D)$
2:	$\texttt{lockIX}_1(a_2)$
3:	$\texttt{lockX}_1(p_4)$
4:	$\texttt{lockIS}_2(D)$
5:	$\texttt{lockIS}_2(a_3)$
6:	$\texttt{lockS}_2(p_5)$
7:	$\texttt{lockIX}_3(D)$
8:	$\texttt{lockIX}_3(a_1)$
9:	$\texttt{lockX}_3(p_1)$
10:	$\texttt{lockIS}_3(a_3)$
11:	$\texttt{lockIS}_3(p_5)$
12:	$\texttt{lockS}_3(s_8)$
13:	$\texttt{lockIX}_2(D)$
14:	$\texttt{lockIX}_2(a_2)$
15:	$\texttt{lockIX}_2(p_3)$
16:	$\texttt{lockX}_2(s_5)$
17:	$\texttt{lockIX}_1(p_3)$
18:	$\texttt{lockX}_1(s_6)$
19:	$\texttt{lockIX}_1(a_1)$
20:	$\texttt{lockX}_1(p_1)$ (1)
21:	$\texttt{lockS}_2(p_3)$ (2)
22:	$\texttt{lockIS}_4(D)$
23:	$\texttt{lockIS}_4(a_3)$
24:	$\texttt{lockIS}_4(p_5)$
25:	$\texttt{lockS}_4(s_{10})$
26:	$\texttt{relS}_3(s_8)$
27:	$\texttt{relIS}_3(p_5)$
28:	$\texttt{relIS}_3(a_3)$
29:	$\texttt{lockIX}_3(a_2)$
30:	$\texttt{lockIX}_3(p_4)$ (3)

Anmerkung: Freigaben von IS dürfen auch ignoriert werden, solange noch ein IX gehalten wird. Wenn IX freigegeben wird, darf auch angenommen werden, dass implizit IS weiter gilt. Es kommt auf konsistente Handhabung an und darauf, dass Studierende sich im Idealfall Gedanken zu dem Case gemacht haben. Siehe Diskussionen im TUWEL Forum.

(a)
- Probleme bei Sperranforderungen:
  - (1): Die IX-Sperre kann wegen der Schreibsperre von T_2 auf p_3 nicht gewährt werden.
- Nein, es kommt zu keinem Deadlock. Die Transaktion T_2 wartet auf eine Sperre die T_1 hält, die Transaktion T_1 kann aber weiterlaufen.
- Die folgenden Freigaben sind nötig damit T_2 weiterlaufen kann: $\texttt{relX}_1(p_3) \rightarrow \texttt{relIX}_1(a_2) \rightarrow \texttt{relIX}_1(D)$. Anschließend erhält T_2 die gewünschte Schreibsperre, 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_3 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) [2 Punkte]

Gegeben ist die unten angeführte Historie dreier Transaktionen T_1, T_2, T_3, welche auf drei Datenobjektion A, B und C zugreifen.

Nehmen Sie an dass in einem zeitstempelbasierten Sperrverfahren die Anfangswerte für `readTS` und `writeTS` für alle drei Werte 0 sind.

Verwenden Sie die Regeln zeitstempel-basierender Synchronisation wie in der Vorlesung besprochen.

- Geben Sie die tatsächliche Historie der drei Transaktionen aus wie sie auf Grund der Synchronisation mittels Zeitstempel abgearbeitet werden.

  Nehmen Sie an, dass abgebrochene Transaktionen in der Reihenfolge wiederholt werden in der sie abgebrochen wurden, und dass eine Wiederholung beginnt sobald alle anderen Transaktionen abgeschlossen wurden.

- Geben Sie zu jeder Lese- oder Schreiboperation die Werte von `readTS`(A), `readTS`(b), `readTS`(C), `writeTS`(A), `writeTS`(b), `writeTS`(C) an. Als Zeitstempel für die Transaktionen nehmen Sie bitte die # ihres `BOT`.

#	T_1	T_2	T_3
1			`BOT`
2	`BOT`		
3		`BOT`	
4			$r_3(A)$
5	$r_1(C)$		
6	$w_1(A)$		
7		$r_2(C)$	
8			$w_3(C)$
9			`commit`
10	$r_1(B)$		
11		$w_2(C)$	
12		`commit`	
13	$w_1(C)$		
14	$w_1(A)$		
15	`commit`		

Lösung:

#	T_1	T_2	T_3	rTS(A)	wTS(A)	rTS(B)	wTS(B)	rTS(C)	wTS(C)
1			BOT	0	0	0	0	0	0
2	BOT			0	0	0	0	0	0
3		BOT		0	0	0	0	0	0
4			$r_3(A)$	1	0	0	0	0	0
5	$r_1(C)$			1	0	0	0	2	0
6	$w_1(A)$			1	2	0	0	2	0
7		$r_2(C)$		1	2	0	0	3	0
8			reset	1	2	0	0	3	0
9	$r_1(B)$			1	2	2	0	3	0
10		$w_2(C)$		1	2	2	0	3	3
11		commit		1	2	2	0	3	3
12	reset			1	0	2	0	3	3
13			BOT	1	0	2	0	3	3
14			$r_3(A)$	13	0	2	0	3	3
15			$w_3(C)$	13	0	2	0	3	13
16			commit	13	0	2	0	3	13
17	BOT			13	0	2	0	3	13
18	$r_1(C)$			13	0	2	0	17	13
19	$w_1(A)$			13	17	2	0	17	13
20	$r_1(B)$			13	17	17	0	17	13
21	$w_1(C)$			13	17	17	0	17	17
22	$w_1(A)$			13	17	17	0	17	17
23	commit			13	17	17	0	17	17

Aufgabe 8 (Transaktionen in SQL) [0.5 Punkte]

Für die folgende Aufgabe können Sie das Beispiel auf einer Datenbank bei sich laufen lassen, müssen dies aber nicht. Für die konzeptuelle Bearbeitung ist es nicht unbedingt notwendig.

Eine Anleitung zur Nutzung unseres Datenbankservers finden Sie hier
`https://tuwel.tuwien.ac.at/mod/page/view.php?id=1654198`

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

(Primärschlüssel sind unterstrichen, Fremdschlüssel sind kursiv geschrieben).

```
Tasks    (ticket_id, assigned_to, reviewed_by, description, effort)
done     (ticket_id: Tasks.ticket_id)
tested   (ticket_id: Tasks.ticket_id, when_tested)
```

Bestimmen Sie für jedes der unten aufgeführten Szenarien das jeweils *niedrigste* Isolation Level welches den geforderten Grad an Isolation bietet. **Beschreiben Sie** außerdem, ob die Transaktionen bei der angegeben 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 vollständingen Abfragen finden Sie in den zur Verfügung gestellten SQL Dateien.

(a) *Beschreibung:* Sophia und Emma stellen einen konzeptuellen Fehler einer bestehenden Aufgabe fest (FeatureXZ). Sie veranschlagen jeder 40 Stunden zur Lösung und müssen einen neuen Task anlegen. Allerdings auch bestehende Tasks updaten – wobei jeweils 20 Stunden für Implementierung und Test veranschlagt werden. Sophia übernimmt das aktualisieren der Implementierung. Emma das Testen. *Historie:*

Sophia	**Emma**
	`BEGIN;` `SET TRANSACTION ISOLATION LEVEL ____`
`BEGIN;` `SET TRANSACTION ISOLATION LEVEL ____`	

`UPDATE Tasks`
`SET effort=effort + 20`
`WHERE description like '%FeatureXZ%';`	

`DELETE FROM done WHERE ticket_id IN`
`(SELECT ticket_id FROM Tasks`
`WHERE description like '%FeatureXZ%');`	

	`UPDATE Tasks`
	`SET effort=effort + 20`
	`WHERE description like '%FeatureXZ%';`

	`DELETE FROM done WHERE ticket_id IN`
	`(SELECT ticket_id FROM Tasks`
	`WHERE description like '%FeatureXZ%');`

`INSERT INTO Tasks VALUES` `(9, 'Emma', 'Sophia', 'Concept', 40);`	
	`INSERT INTO Tasks VALUES` `(8, 'Sophia', 'Emma', 'Concept', 40);`
`COMMIT;`	
	`COMMIT;`

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 Einträg in der Tabelle `Tasks` aktualisieren, der Wert in diesem neuen Eintrag jedoch vom Inhalt der Tabelle `Tasks` abhängt. Das heißt, sowohl Sophie als auch Emma 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 Sohpie den Wert welchen Emma aktualisiert, noch umgekehrt. Im Fall einer seriellen Ausführung, hätte jedoch die Person, deren Transaktion als zweites ausgefürt 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 würde, 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 Tasks released 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 in den Tickets im Zuge dieser Transaktion sichtbar sind.

Historie:

ProjectOwner	**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 tested`	
`SELECT * FROM Tasks NATURAL JOIN tested` `WHERE ¬tested`	
`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 `¬tested` für die Bedingungen
`ticket_id NOT IN (SELECT ticket_id FROM approved)` bzw.
`ticket_id NOT IN (SELECT ticket_id FROM tested)` 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 paralelle Schreibzugriffe verändert.

Historie:

Accounting	**Parallele Anfragen**
`BEGIN;` `SET TRANSACTION ISOLATION LEVEL ____`	
`SELECT * FROM tasks` `  NATURAL JOIN done WHERE billed IS NULL`	
	`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(ticket_id) FROM tasks`
`  NATURAL JOIN done WHERE billed IS NULL`
`COMMIT;`	

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.

Im Paket der Übungsaufgaben finden Sie auch SQL Dateien, die Sie auf einem DBMS einfach ausprobieren können (die Dateien sind für Postgres geschrieben). 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 in 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 einen lokalen Docker-Container verwenden oder sich zu unserem System verbinden (siehe: `https://tuwel.tuwien.ac.at/mod/page/view.php?id=1654198`).

Anleitungen für einen PostgreSQL Docker Container finden Sie vielfältige im Netz, bspw. `https://unbiased-coder.com/run-postgres-mac-docker-2022/` für MacOS. Es gibt unzählige Entwicklungsumgebungen, u.a., DataGrip `https://www.jetbrains.com/de-de/datagrip` oder PyCharm `https://www.jetbrains.com/de-de/pycharm`. Nachdem Sie einen Docker Container erstellt haben, verbinden Sie sich und erstellen eine Datenbank mit "`CREATE DATABASE ex3_8;`". Vergessen Sie nach der Arbeit nicht Ihre Docker Container wieder zu beenden. Alternativ 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. Bitte beachten Sie, dass dieser Server ausschießlich mit ssh aus dem Terminal nutzbar ist. Wir haben im letzten Jahr sehr negative Erfahrungen mit Visual Studio Code (VSC) gesammelt. Bei der Nutzung von VSC am entfernten System werden Plugins ausgeführt, die das Arbeiten von mehreren Studierenden unmöglich macht.