3. Übungsblatt (WS 2021) – 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 10.12.	**12:00 Uhr**:	Upload der Abgabe über TUWEL
ab 10.01.	13:00 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!**

Änderungen im Ablauf bzgl. COVID-19

Wegen der andauerenden Sondersituation werden heuer keine Sprechstunden zum Übungsblatt stattfinen. Bitte wenden Sie sich stattdessen verstärkt an das TUWEL Forum wenn Sie Probleme damit haben die Angaben zu verstehen oder technische Schwierigkeiten haben.

Wenn möglich empfehlen wir Ihnen auch das Forum zur Diskussion mit Ihren Komillitonen zu Nutzen. Ein gemeinsames Analysieren von Problemen hilft erfahrungsgemäß allen Beteiligten dabei den Stoff besser zu verstehen.

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.

Aufgabe 1 (Logging) [1 Punkt]

Betrachten Sie die in Abbildung 1 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 (d.h. das `ROLLBACK` in Schritt 20 wird abgeschlossen bevor der Schreibvorgang in Schritt 22 erfolgt).

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

A: 314 B: 15 C: 92 D: 65

(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*

	T_1	T_2	T_3
1	`BOT`		
2	$r_1(C, c_1)$		
3		`BOT`	
4			`BOT`
5	$r_1(B, b_1)$		
6		$r_2(B, b_2)$	
7	$w_1(B, c_1 - 10)$		
8			$r_3(C, c_3)$
9		$r_2(D, d_2)$	
10	$r_1(A, a_1)$		
11			$w_3(D, c_3 - 10)$
12			$r_3(D, d_3)$
13		$w_2(D, d_2 + b_2)$	
14	$w_1(B, -3 \cdot b_1)$		
15	$r_1(B, b_1)$		
16	$r_1(C, c_1)$		
17	$w_1(C, c_1 - 11)$		
18	$w_1(A, a_1 + b_1 + 2)$		
19		$w_2(B, b_2 + 5)$	
20		`ROLLBACK`	
21	`COMMIT`		
22			$w_3(B, d_3)$
23			`COMMIT`

Abbildung 1: Historie für Aufgabe 1.

der Operation an. Geben Sie jeweils die dazugehörige Zeilennummer der Historie an (verwenden Sie für Änderungen auf Grund des `ROLLBACK`s die Zeilennummer 20).

Lösung:

	Werte (a)
1	
2	$c_1 = 92$
3	
4	
5	$b_1 = 15$
6	$b_2 = 15$
7	$B = c_1 - 10$ $= 82$
8	$c_3 = 92$
9	$d_2 = 65$
10	$a_1 = 314$
11	$D = c_3 - 10$ $= 92 - 10$ $= 82$
12	$d_3 = D = 82$
13	$D = d_2 + b_2$ $= 15 + 65$ $= 80$
14	$B = -3 \cdot b_1$ $= -3 \cdot 15$ $= -45$
15	$b_1 = -45$
16	$c_1 = C = 92$
17	$C = c_1 - 11 = 81$
18	$A = a_1 + b_1 + 2$ $= 314 - 45 + 2$ $= 271$
19	$B = b_2 + 5$ $= 20$
20	$B = -45$ $D = 82$
21	
22	$B = 82$
23	

(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.)

Lösung:

	Log (b) [LSN,TA,PageID,Redo,Undo,PrevLSN] bzw. ⟨LSN,TA,PageID,Redo,PrevLSN,UndoNextLSN⟩
1	[#1,T_1,BOT,#0]
2	
3	[#2,T_2,BOT,#0]
4	[#3,T_3,BOT,#0]
5	
6	
7	[#4,T_1,P_B,$B+$ =67,$B-$ = 67,#1]
8	
9	
10	
11	[#5,T_3,P_D,$D+ = 17$,$D- = 17$,#3]
12	
13	[#6,T_2,P_D,$D- = 2$,$D+ = 2$,#2]
14	[#7,T_1,P_B,$B- = 127$,$B+ = 127$,#4]
15	
16	
17	[#8,T_1,P_C,$C- = 11$,$C+ = 11$,#7]
18	[#9,T_1,P_A,$A- = 43$,$A+ = 43$,#8]
19	[#10,T_2,P_B,$B+ = 65$,$B- = 65$,#6]
20	<#11,T_2,P_B,$B- = 65$,#10,#6> <#12,T_2,P_D,$D+ = 2$,#11,#2> <#13,T_2,BOT,#12,#0>
21	[#14,T_1,COMMIT,#9]
22	[#15,T_3,P_B,$B+ = 127$,$B- = 127$,#5]
23	[#16,T_3,COMMIT,#15]

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_C und P_D dargestellt.

Log-Einträge

[#1, T_1, `BOT`, #0]
[#2, T_2, `BOT`, #0]
[#3, T_2, P_A, A+=12, A-=12, #2]
[#4, T_1, P_B, C+=21, C-=21, #1]
[#5, T_3, `BOT`, #0]
[#6, T_3, P_A, A+=4, A-=4, #5]
[#7, T_3, P_A, A-=7, A+=7, #6]
[#8, T_1, P_B, B+=14, B-=14, #4]
[#9, T_1, P_B, C+=50, C-=50, #8]
⟨#10, T_3, P_A, A+=7, #7, #6⟩
⟨#11, T_3, P_A, A-=4, #10, #5⟩
⟨#12, T_1, P_B, C-=50, #9, #8⟩
⟨#13, T_1, P_B, B-=14, #12, #4⟩
[#14, T_2, P_B, C-=42, C+=42, #3]
⟨#15, T_3, `BOT`, #11⟩
[#16, T_4, `BOT`, #0]
[#17, T_4, P_D, D+=15, D-=15, #16]
[#18, T_4, `COMMIT`, #17]

Seiten im Hintergrundspeicher

P_A	LSN: #3
$A = 50$	

P_B	LSN: #9
$B = 8$	$C = 8$

P_D	LSN: #0
$D = 37$	

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: 50; B: -6; C: -84; D: 52

Beispielhaft für A. Nur die Redos ausführen, die eine höhere LSN haben, als die der jeweiligen Seite, d.h., $50 + 4 - 7 + 7 - 4 = 50$.

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

Lösung:

Log: ⟨`LSN, TA, PageID, Redo, PrevLSN, UndoNextLSN`⟩
⟨#19, T_2, P_B, C+=42, #14, #3⟩
⟨#20, T_1, P_B, C-=21, #13, #1⟩
⟨#21, T_2, P_A, A-=12, #19, #2⟩
⟨#22, T_2, `BOT`, #21⟩
⟨#23, T_1, `BOT`, #20⟩

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

Lösung:

A: 38; B: -6; C: -63; D: 52

Aufgaben: Mehrbenutzersynchronisation

Aufgabe 3 (Eigenschaften von Transaktionen) [2.6 Punkte]

Gegeben ist die Menge $\mathcal{T}_1$ und $\mathcal{T}_2$ an Transaktionen sowie die dazugehörige Historie $\mathcal{H}_1$ und $\mathcal{H}_2$ mit den gegebenen Folge von Elementaroperationen.

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

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

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

(a2) Geben Sie für jede Kante im Serialisierbarkeitsgraphen mindestens ein Paar $p_i \to p_j$ von Operationen an, welches begründet warum diese Kante Teil des Graphen ist.

Geben Sie für die Kanten $T_1 \to T_3$, $T_1 \to T_4$, $T_1 \to T_2$, $T_2 \to T_1$, $T_2 \to T_3$, $T_2 \to T_4$, und $T_3 \to T_4$ (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.

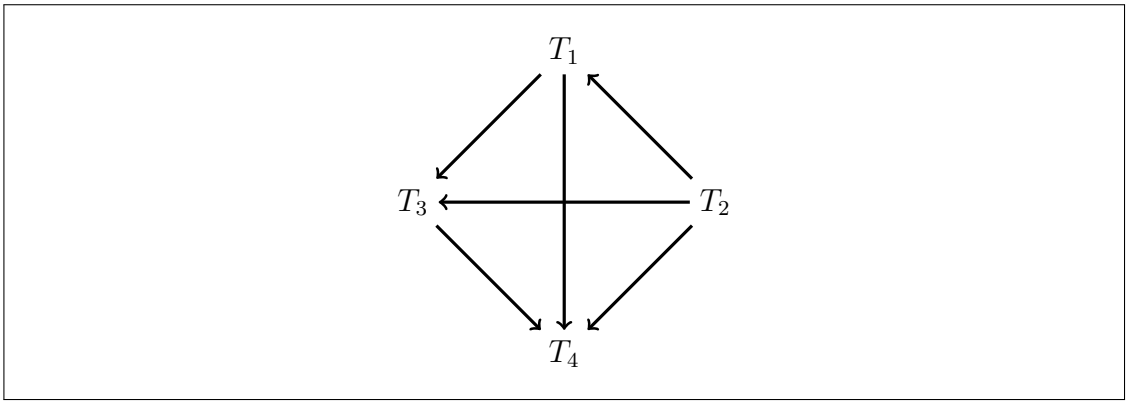
(b) Bestimmen Sie, ob die folgende Historie (1) konfliktserialisierbar ist und (2) welche der drei Eigenschaften (i) Rücksetzbar (ii), vermeidet kaskadierendes Rücksetzen, (iii) strikt die Historie erfüllt.

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

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

Lösung:

(a1) **Serialisierbarkeitsgraph:**



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

Hilfestellung:
Reihenfolge für A Zugriffe: $w_2(A) \rightarrow r_3(A) \rightarrow w_3(A) \rightarrow r_4(A) \rightarrow w_4(A)$

Reihenfolge für B Zugriffe: $r_1(B) \rightarrow w_3(B)$

Reihenfolge für C Zugriffe: $r_2(C) \rightarrow w_1(C) \rightarrow r_3(C)$

Reihenfolge für D Zugriffe: $r_2(D) \rightarrow w_1(D) \rightarrow w_4(D)$

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

$T_1 \rightarrow T_4$
- $w_1(D) \rightarrow w_4(D)$

$T_2 \rightarrow T_1$
- $r_2(C) \rightarrow w_1(C)$
- $r_2(D) \rightarrow w_1(D)$

$T_2 \rightarrow T_3$
- $w_2(A) \rightarrow w_3(A)$
- $w_2(A) \rightarrow r_3(A)$

$T_2 \rightarrow T_4$
- $w_2(A) \rightarrow r_4(A)$
- $w_2(A) \rightarrow w_4(A)$
- $r_2(D) \rightarrow w_4(D)$

$T_3 \rightarrow T_4$
- $r_3(A) \rightarrow w_4(A)$
- $w_3(A) \rightarrow r_4(A)$
- $w_3(A) \rightarrow w_4(A)$

(a3) **Konflikt-Serialisierbarkeit:**

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

Eine mögliche äquivalente serielle Ausführungsreihenfolge wäre

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

(a4) **Lese-Abhängigkeiten:**

- T_3 liest von T_1: $(w_1(C), r_3(C))$
- T_3 liest von T_2: $(w_2(A), r_3(A))$
- T_4 liest von T_3: $(w_3(A), r_4(A))$

(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_3 liest von T_1 sowie T_2 und das `COMMIT` von T_3 kommt nach dem `COMMIT` von T_1 sowie nach T_2.
  - Transaktion T_4 liest von T_2 sowie T_3 und das `COMMIT` von T_4 kommt nach dem `COMMIT` von T_1 sowie T_3.

- **Vermeidet Kaskadierendes Rücksetzen:**

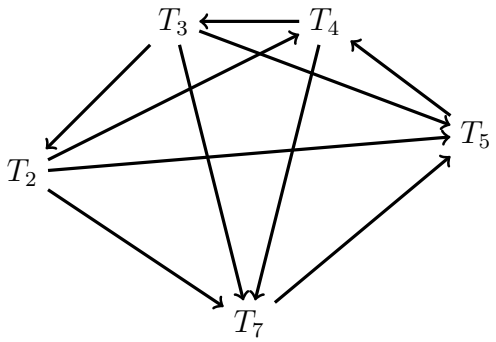
  Um kaskadierendes Rücksetzen zu vermeiden ist es notwendig dass bei jeder Leseoperation ein Wert gelesen wird, welcher von einer bereits erfolgreich abgeschlossenen (commited) Transaktion geschrieben wurde.

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

(b)



Serialisierbarkeitsgraph (Nicht! Teil der Fragestellung):

- "Begründung" für die Kanten **(nicht Teil der Fragestellung)**:

$T_2 \to T_4$
- $r_2(A) \to w_4(A)$

$T_2 \to T_5$
- $r_2(B) \to w_5(B)$
- $r_2(C) \to w_5(C)$
- $w_2(B) \to r_5(B)$
- $w_2(B) \to w_5(B)$

$T_2 \to T_7$
- $w_2(B) \to r_7(B)$

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

$T_3 \to T_5$
- $r_3(C) \to w_5(C)$
- $w_3(C) \to r_5(C)$
- $w_3(C) \to w_5(C)$

$T_3 \to T_7$
- $w_3(C) \to r_7(C)$
- $w_3(C) \to r_7(C)$

$T_4 \to T_3$
- $w_4(A) \to r_3(A)$

$T_4 \to T_7$
- $w_4(A) \to r_7(A)$

$T_5 \to T_4$
- $w_5(D) \to w_4(D)$

$T_7 \to T_5$
- $r_7(B) \to w_5(B)$
- $r_7(C) \to w_5(C)$
- $r_7(C) \to w_5(C)$

(b1) **Konflikt-Serialisierbarkeit:**

Nein, die Historie **ist nicht** konfliktserialisierbar: Der Serialisierbarkeitsgraph enthält sogar mehrere Zyklen, z.B.:
- $T_3 \to T_2 \to T_7 \to T_5 \to T_4 \to T_3$
- $T_3 \to T_2 \to T_4 \to T_3$
- $T_3 \to T_2 \to T_5 \to T_4 \to T_3$
- $T_3 \to T_7 \to T_5 \to T_4 \to T_3$
- $T_3 \to T_5 \to T_4 \to T_3$
- $T_4 \to T_7 \to T_5 \to T_4$

- Lese-Abhängigkeiten **(Nicht Teil der Fragestellung)**:
  - 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: $(w_4(A), r_7(A))$

(b2) **Klassifikation der Historie:**

(b2i) **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.

(b2ii) **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 (commited) 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.

(b2iii) **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.

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:

$Seq = \langle \texttt{lockX}_1(A) \rightarrow \texttt{lockS}_2(C) \rightarrow \texttt{lockX}_3(D) \rightarrow \texttt{lockS}_3(A) \rightarrow \texttt{lockS}_2(B) \rightarrow \texttt{lockS}_1(B) \rightarrow \texttt{rel}_1(A) \rightarrow \texttt{lockX}_3(B) \rightarrow \texttt{lockS}_3(C) \rightarrow \texttt{lockX}_2(D) \rangle$.

(a) Nehmen Sie an, ein DBMS erhält die angegebene Folge von Sperranforderungen für die Transaktionen T_1, T_2 und T_3, 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:	$\texttt{lockX}_1(A)$
2:	$\texttt{grantX}_1(A)$
3:	$\texttt{lockS}_2(C)$
4:	$\texttt{grantS}_2(C)$
5:	$\texttt{lockX}_3(D)$
6:	$\texttt{grantX}_3(D)$
7:	$\texttt{lockS}_3(A)$
8:	$\texttt{wait}(3)$
9:	$\texttt{lockS}_2(B)$

10:	$\texttt{grantS}_2(B)$
11:	$\texttt{lockS}_1(B)$
12:	$\texttt{grantS}_1(B)$
13:	$\texttt{relX}_1(A)$
14:	$\texttt{resume}(3)$
15:	$\texttt{grantS}_3(A)$
16:	$\texttt{lockX}_3(B)$
17:	$\texttt{wait}(3)$
18:	$\texttt{lockX}_2(D)$
19:	$\texttt{wait}(2)$

(b) Skizzieren Sie bitte die aktuelle Situation der gehaltenen Sperren bzw. angehaltener Transaktionen (am Ende der Sperranforderungen oben, also nach $\ldots \rightarrow \texttt{lockX}_2(D)$). 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.

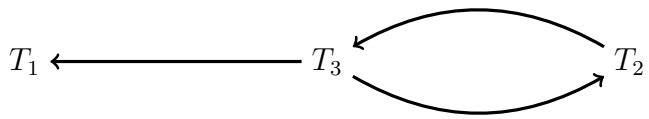
Lösung:

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

(c) Geben Sie den der aktuellen Situation entsprechenden Wartegraphen an. Frühere (bereits aufgelöste) Wartezustände sollten nicht mit einbezogen werden.

Lösung:

Wartegraph:



(d) Geben Sie an, ob zum aktuellen Zeitpunkt (am Ende der Sperranforderungen oben, also nach $\ldots \rightarrow \texttt{lockX}_2(D)$) 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_2 abgebrochen werden (d.h. alle Sperren von T_2 werden sofort freigegeben).

Lösung:
Nachdem derzeit ein Deadlock besteht wird zuerst T_2 abgebrochen, was zu folgender Freigabe führt:

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

Wenn T_3 fortgesetzt wird, muss zunächst T_1 die Sperren freigeben

$$\texttt{rel}_1(B) \rightarrow \texttt{relS}_1(B)$$

Anschließend nun T_3 weiterlaufen, und wir erhalten:

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

Nun kann T_3 ihre Sperren wieder freigeben

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

(f) Betrachten Sie noch einmal die gegebene Sequenz *Seq* an Sperranforderungen und Freigaben. Widerspricht diese dem Zwei Phasen Sperrprotokoll?

(Bedingung 5 bezüglich des Transaktionsendes darf für die Aufgabe ignoriert werden.)

Lösung:

1) Jedes von einer Transaktion benutzte Objekt muss vor Verwendung entsprechend gesperrt werden. **ok**
2) Eine Transaktion fordert keine Sperre an, die sie schon besitzt. **ok**
3) Sperren werden nach der Kompatibilitätsmatrix gewährt. Kann eine Sperre nicht gewährt werden, wartet die Transaktion bis die Sperre gewährt werden kann. **ok**
4) Nachdem eine Transaktion eine Sperre freigegeben hat darf sie keine weiteren Sperren mehr anfordern. **ok**
5) Am Transaktionsende (EOT) müssen alle Sperren freigegeben worden sein. **ok**
   Kann ignoriert werden.

Nein. Insbesondere Bedingung (4) —keine Transaktion fordert mehr eine Sperre an, nachdem von ihr bereits eine Sperre wieder freigegeben wurde— ist erfüllt. 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).

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

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 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. 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 $\alpha_1, \alpha_2, \ldots$, 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	$\texttt{lockS}_1(A)$		
2	$r_1(A)$		
3		$\texttt{lockS}_2(C)$	
4		$r_2(C)$	
5			$\texttt{lockS}_3(B)$
6			$r_3(B)$
7	$\texttt{lockS}_1(B)$		
8	$r_1(B)$		
9		$\texttt{lockX}_2(A)$	
10			$\texttt{lockX}_3(E)$
11			$w_3(E)$
12	$\texttt{lockS}_1(E)$		
13			$\texttt{lockS}_3(A)$
14			$r_3(A)$
15			$r_3(E)$
16			$\texttt{lockX}_3(D)$
17			$w_3(D)$
18			$\texttt{rel}_3(D)$
19			$\texttt{rel}_3(E)$
20			$\texttt{rel}_3(A)$
21			$\texttt{rel}_3(B)$
22	$r_1(E)$		
23			c_3
24	$\texttt{lockS}_1(D)$		
25	$r_1(D)$		
26	$\texttt{rel}_1(D)$		
27	$\texttt{rel}_1(E)$		
28	$\texttt{rel}_1(B)$		
29	$r_1(A)$		
30	$\texttt{rel}_1(A)$		
31		$w_2(A)$	
32	c_1		
33		$\texttt{lockX}_2(C)$	
34		$w_2(C)$	
35		$\texttt{lockS}_2(E)$	
36		$r_2(E)$	
37		$\texttt{rel}_2(E)$	
38		$\texttt{rel}_2(C)$	
39		$w_2(A)$	
40		$\texttt{rel}_2(A)$	
41		c_2	

Aufgabe 6 (Multi-Granularity Locking) [2.4 Punkte]

Betrachten Sie die Datenbasis-Hierarchie in Abbildung 3.

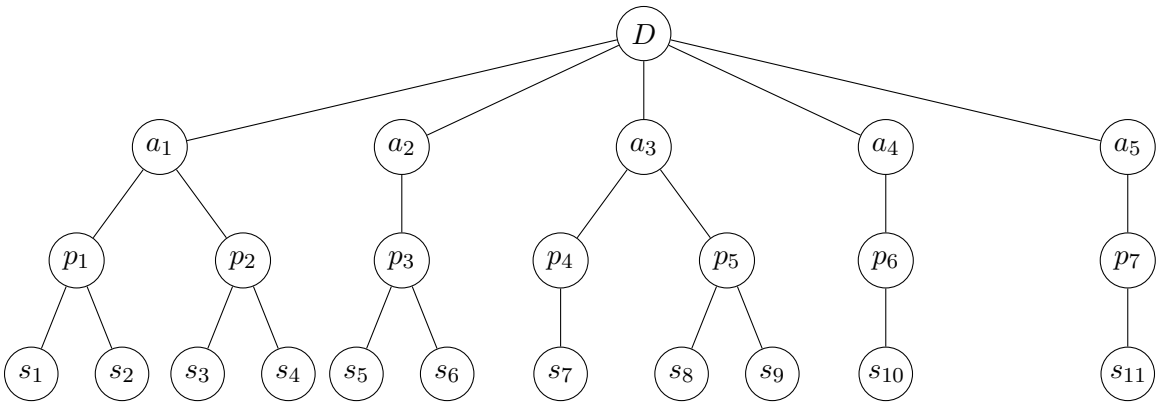


Abbildung 3: Datenbasis-Hierarchie zu Aufgabe 6

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) $\texttt{lockS}_3(p_6) \rightarrow \texttt{lockX}_1(s_{11}) \rightarrow \texttt{lockS}_4(p_2) \rightarrow \texttt{lockX}_2(s_2) \rightarrow \texttt{lockS}_2(p_3) \rightarrow \texttt{lockS}_1(p_5) \rightarrow \texttt{lockX}_3(s_8) \rightarrow \texttt{rel}_2(s_2)$

(b) $\texttt{lockS}_3(s_7) \rightarrow \texttt{lockX}_1(s_4) \rightarrow \texttt{lockX}_2(p_7) \rightarrow \texttt{lockS}_4(s_5) \rightarrow \texttt{lockX}_3(p_6) \rightarrow \texttt{lockS}_2(s_2) \rightarrow \texttt{lockX}_4(s_6) \rightarrow \texttt{lockS}_1(p_5) \rightarrow \texttt{lockS}_3(s_9) \rightarrow \texttt{lockX}_4(s_8) \rightarrow \texttt{lockS}_3(s_{11}) \rightarrow \texttt{rel}_1(p_5) \rightarrow \texttt{lockX}_2(p_2) \rightarrow \texttt{rel}_3(p_6) \rightarrow \texttt{rel}_2(p_7) \rightarrow \texttt{lockS}_3(a_3) \rightarrow \texttt{rel}_4(s_5) \rightarrow \texttt{rel}_2(s_2) \rightarrow \texttt{lockS}_1(s_{10})$

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

1:	$\texttt{lockIS}_3(D)$
2:	$\texttt{lockIS}_3(a_4)$
3:	$\texttt{lockS}_3(p_6)$
4:	$\texttt{lockIX}_1(D)$
5:	$\texttt{lockIX}_1(a_5)$
6:	$\texttt{lockIX}_1(p_7)$
7:	$\texttt{lockX}_1(s_{11})$
8:	$\texttt{lockIS}_4(D)$
9:	$\texttt{lockIS}_4(a_1)$
10:	$\texttt{lockS}_4(p_2)$
11:	$\texttt{lockIX}_2(D)$
12:	$\texttt{lockIX}_2(a_1)$
13:	$\texttt{lockIX}_2(p_1)$
14:	$\texttt{lockX}_2(s_2)$
15:	$\texttt{lockIS}_2(a_2)$
16:	$\texttt{lockS}_2(p_3)$
17:	$\texttt{lockIS}_1(a_3)$
18:	$\texttt{lockS}_1(p_5)$
19:	$\texttt{lockIX}_3(D)$
20:	$\texttt{lockIX}_3(a_3)$
21:	$\texttt{lockIX}_3(p_5)$ (1)
22:	$\texttt{relX}_2(s_2)$
23:	$\texttt{relIX}_2(p_1)$
24:	$\texttt{relIX}_2(a_1)$
25:	$\texttt{relIX}_2(D)$

(b)

1:	$\texttt{lockIS}_3(D)$	22:	$\texttt{lockIX}_4(D)$
2:	$\texttt{lockIS}_3(a_3)$	23:	$\texttt{lockIX}_4(a_2)$
3:	$\texttt{lockIS}_3(p_4)$	24:	$\texttt{lockIX}_4(p_3)$
4:	$\texttt{lockS}_3(s_7)$	25:	$\texttt{lockX}_4(s_6)$
5:	$\texttt{lockIX}_1(D)$	26:	$\texttt{lockIS}_1(a_3)$
6:	$\texttt{lockIX}_1(a_1)$	27:	$\texttt{lockS}_1(p_5)$
7:	$\texttt{lockIX}_1(p_2)$	28:	$\texttt{lockIS}_3(p_5)$
8:	$\texttt{lockX}_1(s_4)$	29:	$\texttt{lockS}_3(s_9)$
9:	$\texttt{lockIX}_2(D)$	30:	$\texttt{lockIX}_4(a_3)$
10:	$\texttt{lockIX}_2(a_5)$	31:	$\texttt{lockIX}_4(p_5)$ (1)
11:	$\texttt{lockX}_2(p_7)$	32:	$\texttt{lockIS}_3(a_5)$
12:	$\texttt{lockIS}_4(D)$	33:	$\texttt{lockIS}_3(p_7)$ (2)
13:	$\texttt{lockIS}_4(a_2)$	34:	$\texttt{relS}_1(p_5)$
14:	$\texttt{lockIS}_4(p_3)$	35:	$\texttt{relIS}_1(a_3)$
15:	$\texttt{lockS}_4(s_5)$	36:	$\texttt{lockIX}_4(p_5)$
16:	$\texttt{lockIX}_3(D)$	37:	$\texttt{lockX}_4(s_8)$
17:	$\texttt{lockIX}_3(a_4)$	38:	$\texttt{lockIX}_2(a_1)$
18:	$\texttt{lockX}_3(p_6)$	39:	$\texttt{lockX}_2(p_2)$ (3)
19:	$\texttt{lockIS}_2(a_1)$	40:	$\texttt{relS}_4(s_5)$
20:	$\texttt{lockIS}_2(p_1)$	41:	$\texttt{lockIS}_1(a_4)$
21:	$\texttt{lockS}_2(s_2)$	42:	$\texttt{lockIS}_1(p_6)$ (4)

(a)
- Probleme bei Sperranforderungen:
  - (1): IX-Sperre von T_3 auf p_5 nicht möglich wegen der Lesesperre von T_1.
- Nein, es kommt zu keinem Deadlock. Die Transaktion T_3 wartet auf eine Sperre die T_1 hält, die Transaktion T_1 kann aber weiterlaufen.
- Die folgenden Freigaben sind nötig damit T_3 weiterlaufen kann: $\texttt{relS}_1(p_5) \rightarrow \texttt{relIS}_1(a_3) \rightarrow \texttt{relIS}_1(D)$. Anschließend erhält T_3 die gewünschte Schreibsperre, und jede der Transaktionen hat alle Einträge der Sequenz abgearbeitet.

(b)
- Probleme bei Sperranforderungen:
  - (1): IX-Sperre für T_4 auf p_5 nicht möglich wegen Lesesperre von T_1.
  - (2): IS-Sperre für T_3 auf p_7 nicht möglich wegen der Schreibsperre von T_2.
  - (3): Schreibsperre von T_2 auf p_2 nicht möglich wegen der IX-Sperre von T_1.
  - (4): IS-Sperre von T_1 auf p_6 nicht möglich wegen der Schreibsperre von T_3 .
- 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 von T_2, und T_2 auf eine Sperre von T_1 wartet . Dies führt zu einer zyklischen Abhängigkeit zwischen den drei genannten Transaktionen, und keine der betroffenen Sperren kann 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, C und D zugreifen.

$\texttt{BOT}_1 \rightarrow w_1(A) \rightarrow \texttt{BOT}_2 \rightarrow r_2(C) \rightarrow \texttt{BOT}_3 \rightarrow w_3(C) \rightarrow r_1(D) \rightarrow w_2(D) \rightarrow w_1(D) \rightarrow \texttt{BOT}_4 \rightarrow res? \rightarrow w_4(B) \rightarrow r_4(C) \rightarrow w_1(B) \rightarrow c_4 \rightarrow c_3 \rightarrow c_1 \rightarrow r_2(D) \rightarrow res? \rightarrow c_2$

Dabei bezeichnet $\texttt{BOT}_i$ 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 Rücksetzen 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 gegeben 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 Rücksetzen 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)	rTS(D)	wTS(D)

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 $\texttt{reset}_i$ 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:

#	Aktion	rTS(A)	wTS(A)	rTS(B)	wTS(B)	rTS(C)	wTS(C)	rTS(D)	wTS(D)
1:	$\texttt{BOT}_1$	0	0	0	0	0	0	0	0
2:	$w_1(A)$	0	1	0	0	0	0	0	0
3:	$\texttt{BOT}_2$	0	1	0	0	0	0	0	0
4:	$r_2(C)$	0	1	0	0	3	0	0	0
5:	$\texttt{BOT}_3$	0	1	0	0	3	0	0	0
6:	$w_3(C)$	0	1	0	0	3	5	0	0
7:	$r_1(D)$	0	1	0	0	3	5	1	0
8:	$w_2(D)$	0	1	0	0	3	5	1	3
9:	$\texttt{reset}_1\ [w_1(D)]$	0	1	0	0	3	5	1	3
10:	$\texttt{BOT}_4$	0	1	0	0	3	5	1	3
11:	$\texttt{BOT}_1$	0	1	0	0	3	5	1	3
12:	$w_1(A)$	0	11	0	0	3	5	1	3
13:	$r_1(D)$	0	11	0	0	3	5	11	3
14:	$w_1(D)$	0	11	0	0	3	5	11	11
15:	$w_4(B)$	0	11	0	10	3	5	11	11
16:	$r_4(C)$	0	11	0	10	10	5	11	11
17:	$w_1(B)$	0	11	0	11	10	5	11	11
18:	c_4	0	11	0	11	10	5	11	11
19:	c_1	0	11	0	11	10	5	11	11
20:	c_3	0	11	0	11	10	5	11	11
21:	$\texttt{reset}_2\ [r_2(D)]$	0	11	0	11	10	5	11	11
22:	$\texttt{BOT}_2$	0	11	0	11	10	5	11	11
23:	$r_2(C)$	0	11	0	11	22	5	11	11
24:	$w_2(D)$	0	11	0	11	22	5	11	22
25:	$r_2(D)$	0	11	0	11	22	5	22	22
26:	c_2	0	11	0	11	22	5	22	22

(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_4 das Datum C, welches zuvor von T_3 geschrieben wurde. Rücksetzbarkeit verlangt nun, dass T_4 nicht vor T_3 committed. Genau das passiert aber in Schritt 19: T_4 macht ihr commit, obwohl T_3 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 diesesmal – falls anwendbar – auch die Schreibzeitstempel zurückgesetzt werden. Lesezeitstempel bleiben wiederum unverändert.

Um in der Tat alle Deadlocks auszuschlißen 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, C und D 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)	rTS(D)	wTS(D)	d(D)

- Wenn eine Transaktion T_i blockiert, ergänzen Sie die Historie mit einem Eintrag $\texttt{block}_i$.

Lösung:

#	Aktion	rTS(A)	wTS(A)	d(A)	rTS(B)	wTS(B)	d(B)	rTS(C)	wTS(C)	d(C)	rTS(D)	wTS(D)	d(D)
1:	$\texttt{BOT}_1$	0	0		0	0		0	0		0	0	
2:	$w_1(A)$	0	1	✓1	0	0		0	0		0	0	
3:	$\texttt{BOT}_2$	0	1	✓1	0	0		0	0		0	0	
4:	$r_2(C)$	0	1	✓1	0	0		3	0		0	0	
5:	$\texttt{BOT}_3$	0	1	✓1	0	0		3	0		0	0	
6:	$w_3(C)$	0	1	✓1	0	0		3	5	✓3	0	0	
7:	$r_1(D)$	0	1	✓1	0	0		3	5	✓3	1	0	
8:	$w_2(D)$	0	1	✓1	0	0		3	5	✓3	1	3	✓2
9:	$\texttt{reset}_1\ [w_1(D)]$	0	0		0	0		3	5	✓3	1	3	✓2
10:	$\texttt{BOT}_4$	0	0		0	0		3	5	✓3	1	3	✓2
11:	$\texttt{BOT}_1$	0	0		0	0		3	5	✓3	1	3	✓2
12:	$w_1(A)$	0	11	✓1	0	0		3	5	✓3	1	3	✓2
13:	$\texttt{block}_1\ [r_1(D)]$	0	11	✓1	0	0		3	5	✓3	1	3	✓2
14:	$w_4(B)$	0	11	✓1	0	10	✓4	3	5	✓3	1	3	✓2
15:	$\texttt{block}_4\ [r_4(C)]$	0	11	✓1	5	10	✓4	3	5	✓3	1	3	✓2
16:	c_3	0	11	✓1	5	10	✓4	3	5		1	3	✓2
17:	$\texttt{cont.}_4\ [r_4(C)]$	0	11	✓1	5	10	✓4	10	5		1	3	✓2
18:	c_4	0	11	✓1	5	10		10	5		1	3	✓2
19:	$r_2(D)$	0	11	✓1	5	10		10	5		3	3	✓2
20:	c_2	0	11	✓1	5	10		10	5		3	3	
21:	$\texttt{cont.}_1\ [r_1(D)]$	0	11	✓1	5	10		10	5		11	3	
22:	$w_1(D)]$	0	11	✓1	5	10		10	5		11	11	
23:	$w_1(B)$	0	11	✓1	5	11	✓1	10	5		11	11	
24:	c_1	0	11		5	11		10	5		11	11	

(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, C und D 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 einer Bank, in welchem durchgeführte Transaktion und deren Verwaltung gehandhabt wird werden:
(Primärschlüssel sind unterstrichen, Fremdschlüssel sind kursiv geschrieben).

```
account      ( no, balance, max_overdraft, fee            )
transaction  ( sender: account.no, recipient: account.no  )
deposit      ( id, to: account.no, amount                  )
withdrawal   ( date, from: account.no, amount              )
```

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 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:*

Es gilt die jährlichen Gebühren abzurechnen. Dies wird von einer Betreuerin durchgeführt. Während diese an den Konten arbeitet, sollen parallele Anfragen möglich sein, solange dies zu keinen Anomalien, wie etwa falschen Kontoständen führt.

Historie:

Betreuerin

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL ?

SELECT balance, fee INTO b2 FROM account WHERE no = 2;
SELECT balance, fee INTO b4 FROM account WHERE no = 4;









UPDATE account SET balance = b2.balance - b2.fee
FROM b2 WHERE no = 2;
UPDATE account SET balance = b4.balance - b4.fee
FROM b4 WHERE no = 4;

INSERT INTO withdrawal SELECT CURRENT_DATE,2,fee FROM b2;
INSERT INTO withdrawal SELECT CURRENT_DATE,4,fee FROM b4;
COMMIT;
```

Parallele Anfragen

```
--




INSERT INTO deposit VALUES (3,2,100);
UPDATE account
SET balance = balance+100 WHERE no = 2;

INSERT INTO deposit VALUES (4,4,400);
UPDATE account
SET balance = balance+400 WHERE no = 4;









--
```

Lösung:
Isolation Level: `REPEATABLE READ`

Es kommt zu Problemen, da Postgres erkennt, dass zwischenzeitlich die Tabelle verändert wurde. Es kommt zu einem Abbruch, und die Betreuerin Transaktion muss später wiederholt werden.

(b) *Beschreibung:*

Ein Mitarbeiter der Bank fragt manuell nach noch nicht bewilligten Transaktionen und führt diese durch, mit Abgleich des Kontostands. Es ist erlaubt, dass abgeschlossene parallele Anfragen währenddessen neue Transaktionen in die Datenbasis einfügen.

Historie:

Mitarbeiter

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL ?;

-- Note that 'x += y' stands for 'x = x + y'
UPDATE account SET balance += 200 where no = 2 ;
UPDATE account SET balance -= 200 where no = 5 ;
INSERT INTO approved VALUES  (2);




UPDATE account SET balance += 200 where no = 4 ;
UPDATE account SET balance -= 200 where no = 3;
INSERT INTO approved VALUES (6);



COMMIT;
```

Parallele Anfragen

```
--




INSERT INTO transaction VALUES
       (6, 4,3,200);




INSERT INTO transaction VALUES
       (7, 2,6,800);
--
```

Lösung:

Isolation Level: `READ COMMITTED`

Hinweis: Dies entspricht in PostgreSQL auch dem Default, und daher wäre es auch erlaubt, wenn man den SET TRANSACTION Befehl einfach weglässt.

Die Transaktionen sollten auf jeden Fall wie gewünscht durchgeführt werden können. Der Isolation Level `READ UNCOMMITTED` wäre nicht erlaubt, da nach der Angabe nur *abgeschlossene* Änderungen sichtbar sein sollen.

(c) *Beschreibung:*

Die Buchhaltung muss regelmäßige Analysen über den aktuellen Datenbestand durchführen. Da die Bank währenddessen weiterarbeitet, sind parallele Anfragen grundsätzlich erlaubt. Allerdings *müssen* die Anfragen der Buchhaltung über einen einheitlichen, konsistenten Zustand der Datenbank ablaufen.

Historie:

Buchhaltung

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL ?


SELECT id FROM transaction EXCEPT
(SELECT * FROM approved);


SELECT AVG(amount) FROM deposit;
SELECT AVG(amount) FROM withdrawal;




SELECT SUM(balance) FROM account
WHERE no NOT IN (SELECT sender FROM transaction);
COMMIT;
```

Parallele Anfragen

```
--

INSERT INTO transaction VALUES
       (8,4,1,220);


INSERT INTO deposit (6,8, 1220 );
INSERT INTO withdrawal (CURRENT_DATE,2,2);



INSERT INTO transaction VALUES
       (9,4,2,120);


--
```

Lösung:
Isolation Level: `SERIALIZABLE`

Unter PostgresSQL ist bereits der Isolation Level `REPEATABLE READ` genug um sicherzustellen, dass die parallelen Anfragen keinen Effekt auf die Werte haben, die die Buchhaltung in ihren Anfrage zurückbekommt. Streng nach dem SQL Standard ist allerdings erst `SERIALIZABLE` genug um das Phantomproblem zu vermeiden, um welches ja in diesem Beispiel geht.

Einen korrekten Ablauf können Sie kontrollieren in dem Sie Szenario3-a einmal alleine ausführen, und die Werte dann vergleichen mit dem parallelen Ablauf von Szenario3-a und Szenario3-b. Die Angabe verlangt, dass für Szenario3-a die gleichen Werte zurückbekommen, also die parallelen Anfragen nicht sichtbar sind für die Buchhaltung.

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.