# 3. Übungsblatt (WS 2020) – 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 11.12.** | **12:00 Uhr**: | Upload der Abgabe über TUWEL |
| ab 12.01. | 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 $\Gamma$ wird aus der Datenbasis in eine lokale Variable $\gamma$ gelesen) und $w_i(\Gamma, \gamma)$ eine Schreiboperation (der Wert $\gamma$ wird in das Feld $\Gamma$ 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 21 erfolgt).

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

$$A\colon 22 \qquad B\colon 56 \qquad C\colon 0 \qquad D\colon -3$$

| | $T_1$ | $T_2$ | $T_3$ |
|---|---|---|---|
| 1 | `BOT` | | |
| 2 | $r_1(B, b_2)$ | | |
| 3 | | `BOT` | |
| 4 | $r_1(A, a_1)$ | | |
| 5 | | | `BOT` |
| 6 | | | $w_3(C, 0 + 13)$ |
| 7 | | $r_2(A, a_3)$ | |
| 8 | $w_1(B, a_1 + 14)$ | | |
| 9 | | $w_2(C, 0 - 11)$ | |
| 10 | | | $r_3(A, a_2)$ |
| 11 | $w_1(D, a_1 + b_2)$ | | |
| 12 | | $w_2(A, a_3 + 47)$ | |
| 13 | | | $w_3(A, a_2 - 24)$ |
| 14 | | | $r_3(D, d_1)$ |
| 15 | $r_1(B, b_7)$ | | |
| 16 | | $r_2(B, b_3)$ | |
| 17 | | | `COMMIT` |
| 18 | | $w_2(D, b_3 + a_3)$ | |
| 19 | $w_1(B, b_2 + b_7)$ | | |
| 20 | | `ROLLBACK` | |
| 21 | $r_1(D, d_3)$ | | |
| 22 | $w_1(B, d_3 + 17)$ | | |
| 23 | `COMMIT` | | |

Abbildung 1: Historie für Aufgabe 1.

(a) Geben Sie für jede Zeile der Historie, in welcher entweder der Wert eines Feldes, oder einer lokalen Variabel geändert wird, den Wert des entsprechenden Feldes/Variablen *nach* der Operation an. Geben Sie jeweils die dazugehörige Zeilennummer der Historie an (verwenden Sie für Änderungen auf Grund des `ROLLBACK`s die Zeilennummer 20).

**Lösung:**

| | Wert |
|---|---|
| 2 | $b_2 = 56$ |
| 4 | $a_1 = 22$ |
| 6 | $C = 13$ |
| 7 | $a_3 = 22$ |
| 8 | $B = 36$ |
| 9 | $C = -11$ |
| 10 | $a_2 = 22$ |
| 11 | $D = 78$ |
| 12 | $A = 69$ |
| 13 | $A = -2$ |
| 14 | $d_1 = 78$ |
| 15 | $b_7 = 36$ |
| 16 | $b_3 = 36$ |
| 18 | $D = 58$ |
| 19 | $B = 92$ |
| 20 | $D = 78$ |
| 20 | $A = -49$ |
| 20 | $C = 13$ |
| 21 | $d_3 = 78$ |
| 22 | $B = 95$ |

(b) Geben Sie eine Liste der entsprechenden Log-Einträge zu dieser Historie – in der Rei-

henfolge 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 $\Gamma$ auf der Seite $P_\Gamma$ 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:<br>`[LSN, TA, PageID, Redo, Undo, PrevLSN]` bzw.<br>$\langle$`LSN, TA, PageID, Redo, PrevLSN, UndoNextLSN`$\rangle$ |
|---|---|
| 1 | [#1, $T_1$, `BOT`, #0] |
| 3 | [#2, $T_2$, `BOT`, #0] |
| 5 | [#3, $T_3$, `BOT`, #0] |
| 6 | [#4, $T_3$, $P_C$, C+=13, C-=13, #3] |
| 8 | [#5, $T_1$, $P_B$, B-=20, B+=20, #1] |
| 9 | [#6, $T_2$, $P_C$, C-=24, C+=24, #2] |
| 11 | [#7, $T_1$, $P_D$, D+=81, D-=81, #5] |
| 12 | [#8, $T_2$, $P_A$, A+=47, A-=47, #6] |
| 13 | [#9, $T_3$, $P_A$, A-=71, A+=71, #4] |
| 17 | [#10, $T_3$, `COMMIT`, #9] |
| 18 | [#11, $T_2$, $P_D$, D-=20, D+=20, #8] |
| 19 | [#12, $T_1$, $P_B$, B+=56, B-=56, #7] |
| 20 | $\langle$#13, $T_2$, $P_D$, D+=20, #11, #8$\rangle$ |
| 20 | $\langle$#14, $T_2$, $P_A$, A-=47, #13, #6$\rangle$ |
| 20 | $\langle$#15, $T_2$, $P_C$, C+=24, #14, #2$\rangle$ |
| 20 | $\langle$#16, $T_2$, `BOT`, #15$\rangle$ |
| 22 | [#17, $T_1$, $P_B$, B+=3, B-=3, #12] |
| 23 | [#18, $T_1$, `COMMIT`, #17] |

**Aufgabe 2 (Recovery)** [1.5 Punkte]

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

**Log-Einträge**

[#1, $T_1$, `BOT`, #0]
[#2, $T_3$, `BOT`, #0]
[#3, $T_3$, $P_A$, A+=12, A-=12, #2]
[#4, $T_2$, `BOT`, #0]
[#5, $T_1$, $P_D$, D+=21, D-=21, #1]
[#6, $T_2$, $P_A$, A+=34, A-=34, #4]
[#7, $T_3$, $P_A$, B-=22, B+=22, #3]
⟨#8, $T_3$, $P_A$, B+=22, #7, #3⟩
[#9, $T_1$, $P_C$, C+=50, C-=50, #5]
[#10, $T_5$, `BOT`, #0]
[#11, $T_1$, $P_D$, D+=100, D-=100, #9]
⟨#12, $T_1$, $P_D$, D-=100, #11, #9⟩
[#13, $T_5$, $P_C$, C-=11, C+=11, #10]
⟨#14, $T_3$, $P_A$, A-=12, #8, #2⟩
⟨#15, $T_1$, $P_C$, C-=50, #12, #5⟩
[#16, $T_2$, $P_D$, D-=23, D+=23, #6]
⟨#17, $T_3$, `BOT`, #14⟩
[#18, $T_4$, `BOT`, #0]
[#19, $T_5$, $P_A$, A+=76, A-=76, #13]
[#20, $T_4$, $P_A$, B+=15, B-=15, #18]
[#21, $T_4$, `COMMIT`, #20]

**Seiten im Hintergrundspeicher**

| $P_A$ | LSN: #6 |
|---|---|
| $A = 55$ | $B = 7$ |

| $P_C$ | LSN: #13 |
|---|---|
| $C = 50$ | |

| $P_D$ | LSN: #11 |
|---|---|
| $D = 121$ | |

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:**

$A$: 119; $B$: 22; $C$: 0; $D$: -2

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

**Lösung:**

| Log: ⟨`LSN, TA, PageID, Redo, PrevLSN, UndoNextLSN`⟩ |
|---|
| ⟨#22, $T_5$, $P_A$, A-=76, #19, #13⟩ |
| ⟨#23, $T_2$, $P_D$, D+=23, #16, #6⟩ |
| ⟨#24, $T_5$, $P_C$, C+=11, #22, #10⟩ |
| ⟨#25, $T_5$, `BOT`, #24⟩ |
| ⟨#26, $T_2$, $P_A$, A-=34, #23, #4⟩ |
| ⟨#27, $T_1$, $P_D$, D-=21, #15, #1⟩ |
| ⟨#28, $T_2$, `BOT`, #26⟩ |
| ⟨#29, $T_1$, `BOT`, #27⟩ |

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

**Lösung:**

| $A$: 9; $B$: 22 $C$: 11 $D$: 0 |
|---|

# Aufgaben: Mehrbenutzersynchronisation

**Aufgabe 3 (Eigenschaften von Transaktionen)** [2.6 Punkte]

Gegeben ist die Menge $\mathcal{T}$ an Transaktionen sowie die dazugehörige Historie $\mathcal{H}$, welche durch ihre Folge von Elementaroperationen gegeben ist:
$\mathcal{T} = \{T_1, T_2, T_3, T_4\}$
$\mathcal{H} = b_1 \to r_1(A) \to b_3 \to r_1(B) \to b_2 \to w_1(B) \to r_2(C) \to b_4 \to w_2(C) \to r_4(C) \to w_3(D) \to r_3(C) \to r_2(B) \to w_1(D) \to a_3 \to r_1(D) \to c_1 \to w_2(A) \to c_2 \to r_4(A) \to w_4(A) \to c_4$.

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

(b) 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 $T_1 \to T_2$, $T_2 \to T_3$ und $T_2 \to T_4$ (sofern sie tatsächlich Teil des Graphen sind) *sämtliche* Konfliktoperationen an, welche verlangen dass diese Kanten Teil des Graph sind.

(c) Falls die Historie konfliktserialisierbar ist, geben Sie *eine* mögliche konfliktäquivalente serielle Reihenfolge an. Andernfalls geben Sie eine (möglichst kleine) Menge an Transaktionen an welche man abbrechen müsste damit die Historie konfliktserialisierbar wird. Geben Sie anschließend eine mögliche konfliktäquivalente serielle Reihenfolge an.

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

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

- Rücksetzbar
- Vermeidet kaskadierendes Rücksetzen
- Strikt

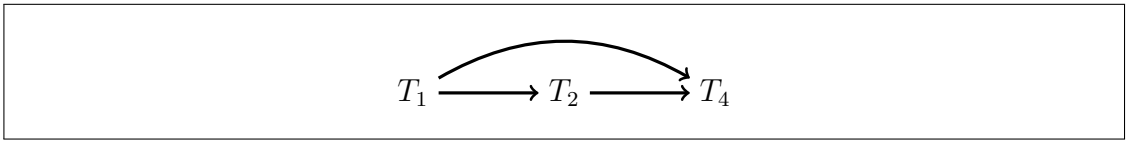
Begründen Sie jeweils Ihre Antwort.

(f) Bestimmen Sie, ob die folgende Historie konfliktserialisierbar ist und welche der drei Eigenschaften aus Aufgabe (e) (Rücksetzbar, vermeidet kaskadierendes Rücksetzen und strikt) die Historie erfüllt.

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

**Lösung:**

(a) **Serialisierbarkeitsgraph:**




(b) **"Begründung" für die Kanten:**

$T_1 \to T_2$

- $w_1(B) \to r_2(B)$
- $r_1(A) \to w_2(A)$

$T_2 \to T_4$

- $w_2(C) \to r_4(C)$
- $w_2(A) \to r_4(A)$
- $w_2(A) \to w_4(A)$

$T_1 \to T_4$

- $r_1(A) \to w_4(A)$

(c) **Konflikt-Serialisierbarkeit:**

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

Eine mögliche äquivalente serielle Ausführungsreihenfolge wäre

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

(d) **Lese-Abhängigkeiten:**

- $T_2$ liest von $T_1$: $(w_1(B), r_2(B))$
- $T_3$ liest von $T_2$: $(w_2(C), r_3(C))$
- $T_4$ liest von $T_2$: $(w_2(C), r_4(C))$ oder $(w_2(A), r_4(A))$

(e) **Klassifikation der Historie:**

- **Rücksetzbar:**

  **Ja**, die Historie *ist* rücksetzbar. Transaktion $T_2$ liest von $T_1$ und das `COMMIT` von $T_2$ kommt nach dem `COMMIT` von $T_1$. Ebenso liest $T_4$ von $T_2$ und das `COMMIT` von $T_4$ kommt nach dem `COMMIT` von $T_2$.

- **Vermeidet Kaskadierendes Rücksetzen:**

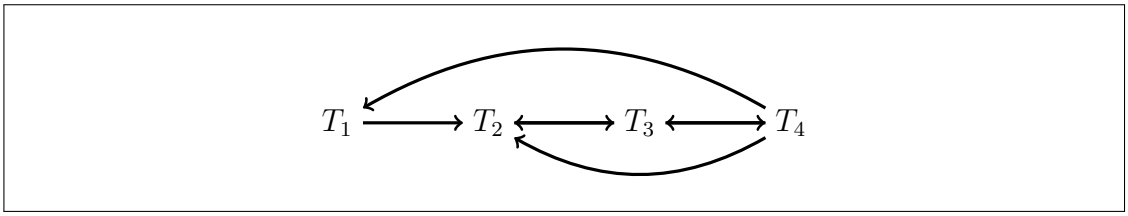
  **Nein**. Um kaskadierendes Rücksetzen zu vermeiden ist es notwendig dass bei jeder Leseoperation ein Wert gelesen wird, welcher von einer bereits erfolgreich abgeschlossenen (commited) Transaktion geschrieben wurde. Dies wird in der vorliegenden

Historie verletzt. Ein solches Beispiel ist: $r_2(B)$ 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 $r_2(B)$ und $w_1(B)$ auch ein Beispiel dafür, dass die Historie nicht strikt ist.

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




**(Dieser Serialisierbarkeitsgraph wird in der Angabe nicht gefordert!)**

Leseabhängigkeiten (nicht gefordert)

- $T_1$ liest von $T_4$: $(w_4(C), r_1(C))$
- $T_2$ liest von $T_3$: $(r_2(D), w_3(D))$
- $T_3$ liest von $T_4$: $(w_4(D), r_3(D))$

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

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

- **Vermeidet Kaskadierendes Rücksetzen:**

  **Nein**, die Historie vermeidet nicht kaskadierendes Rücksetzen. Beispielsweise liest $T_1$ von $T_4$, aber das `COMMIT` von $T_4$ ist erst nach dem Lesezugriff auf Datum $C$

- **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 $w_1(A) \rightarrow w_2(A)$ gilt, aber dazwischen kein `COMMIT` von $T_1$ 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:

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

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

| | |
|---|---|
| 10: | $\texttt{wait}(2)$ |
| 11: | $\texttt{lockX}_4(C)$ |
| 12: | $\texttt{grantX}_4(C)$ |
| 13: | $\texttt{lockS}_3(B)$ |
| 12: | $\texttt{wait}(3)$ |
| 14: | $\texttt{rel}_4(A)$ |
| 15: | $\texttt{relS}_4(A)$ |
| 16: | $\texttt{lockS}_1(D)$ |
| 17: | $\texttt{wait}(1)$ |

(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.

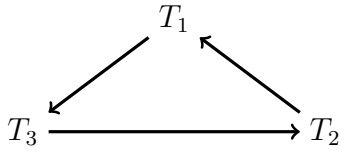
**Lösung:**

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

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

**Lösung:**

**Wartegraph:**




$T_4$

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

(e) Geben Sie eine mögliche Sequenz an Freigaben an, mit welcher sämtliche gehaltenen Sperren wieder freigegeben werden können. Sollte eine blockierte Transaktion durch die Freigabe einer Sperre weiterlaufen können, so müssen von dieser Transaktion zuerst alle ausstehenden Sperranfragen und Freigaben aus der Angabe abgearbeitet werden, bevor Sie zusätzliche Freigaben definieren können. Sollte derzeit ein Deadlock bestehen, so soll Transaktion $T_3$ abgebrochen werden (d.h. alle Sperren von $T_3$ werden sofort freigegeben).

**Lösung:**
Nachdem derzeit ein Deadlock besteht wird zuerst $T_3$ abgebrochen, was zu folgender Freigabe führt:

$$\texttt{rel}_3(D) \rightarrow \texttt{relX}_3(D)$$

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

$$\texttt{resume}(1) \rightarrow \texttt{grantS}_1(D)$$

Nun kann $T_1$ ihre Sperren wieder freigeben

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

Dadurch kann Transaktion $T_2$ weiterlaufen

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

Nun kann $T_4$ ihre Sperre aufgeben

$$\texttt{rel}_4(C) \rightarrow \texttt{relX}_4(C)$$

Was nun dazu führt dass $T_2$ die letzte Sperre bekommt,

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

und anschließend ebenfalls alle Sperren freigeben kann.

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

(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 $T_i$).

$$\begin{array}{lccccccccccc} T_1\colon & w_1(C) & \to & r_1(B) & \to & w_1(C) & \to & w_1(B) & \to & w_1(E) & \to & c_1 \\ T_2\colon & w_2(A) & \to & r_2(A) & \to & w_2(B) & \to & r_2(D) & \to & r_2(E) & \to & c_2 \\ T_3\colon & r_3(C) & \to & r_3(D) & \to & w_3(C) & \to & w_3(A) & \to & r_3(D) & \to & c_3 \end{array}$$

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 dabei an, dass Sperren so sparsam wie möglich beantragt werden. D.h.
  - Es werden nur Sperren beantragt die auch tatsächlich benötigt werden.
  - Sperren werden so kurz wie möglich gehalten, d.h. sie werden spät wie möglich beantragt, und so früh wie möglich wieder freigegeben.

- *Verzahnung der Transaktionen:* Nehmen Sie an, dass jede Transaktion jeweils eine Operation (lesen, schreiben, commit) abarbeitet, und anschließend die nächste Transaktion ausgeführt wird. D.h. im gegeben Fall wäre die Reihenfolge der Aktionen $w_1(C) \to w_2(A) \to r_3(C) \to r_1(B) \to \ldots$. 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 $\beta_3$ 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 $\alpha_5$ wieder aufgegeben, so ist die weitere Abfolge $\alpha_5, \beta_3, \alpha_6, \beta_4, \ldots$.

**Lösung:**

| # | $T_1$ | $T_2$ | $T_3$ |
|---|---|---|---|
| 1 | $\texttt{lockX}_1(C)$ | | |
| 2 | $w_1(C)$ | | |
| 3 | | $\texttt{lockX}_2(A)$ | |
| 4 | | $w_2(A)$ | |
| 5 | | | $\texttt{lockS}_3(C)$ |
| 6 | $\texttt{lockS}_1(B)$ | | |
| 7 | $r_1(B)$ | | |
| 8 | | $r_2(A)$ | |
| 9 | $w_1(C)$ | | |
| 10 | | $\texttt{lockX}_2(B)$ | |
| 11 | $\texttt{lockX}_1(B)$ | | |
| 12 | $w_1(B)$ | | |
| 13 | $\texttt{lockX}_1(E)$ | | |
| 14 | $w_1(E)$ | | |
| 15 | $\texttt{rel}_1(C)$ | | |
| 16 | $\texttt{rel}_1(B)$ | | |
| 17 | $\texttt{rel}_1(E)$ | | |
| 18 | | $w_2(B)$ | |
| 19 | | | $r_3(C)$ |
| 20 | $c_1$ | | |
| 21 | | $\texttt{lockS}_2(D)$ | |
| 22 | | $r_2(D)$ | |
| 23 | | | $\texttt{lockS}_3(D)$ |
| 24 | | | $r_3(D)$ |
| 25 | | $\texttt{lockS}_2(E)$ | |
| 26 | | $r_2(E)$ | |
| 27 | | $\texttt{rel}_2(A)$ | |
| 28 | | $\texttt{rel}_2(B)$ | |
| 29 | | $\texttt{rel}_2(D)$ | |
| 30 | | $\texttt{rel}_2(E)$ | |
| 31 | | | $\texttt{lockX}_3(C)$ |
| 32 | | | $w_3(C)$ |
| 33 | | $c_2$ | |
| 34 | | | $\texttt{lockX}_3(A)$ |
| 35 | | | $w_3(A)$ |
| 36 | | | $\texttt{rel}_3(A)$ |
| 37 | | | $\texttt{rel}_3(C)$ |
| 38 | | | $r_3(D)$ |
| 39 | | | $\texttt{rel}_3(D)$ |
| 40 | | | $c_3$ |

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

Betrachten Sie die Datenbasis-Hierarchie in Abbildung 3.

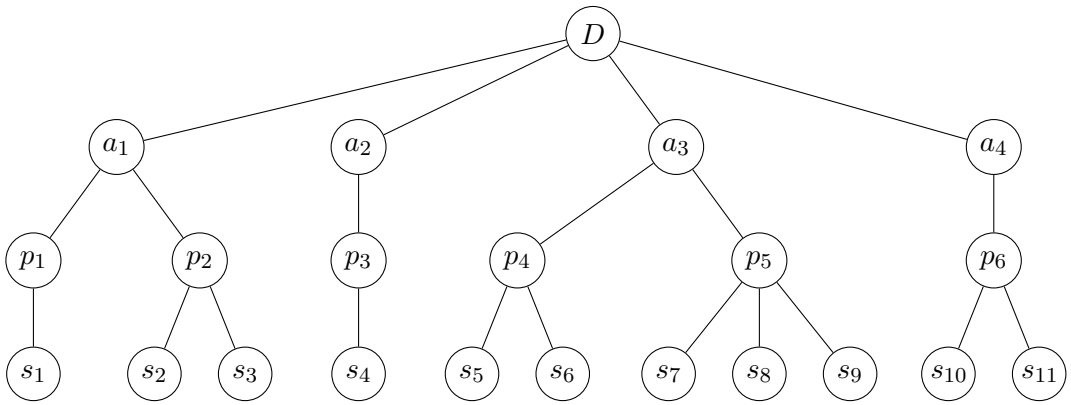



Abbildung 3: Datenbasis-Hierarchie zu Aufgabe 6

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) $\texttt{lockS}_2(p_2) \rightarrow \texttt{lockX}_1(s_8) \rightarrow \texttt{lockS}_1(p_3) \rightarrow \texttt{lockS}_4(s_{11}) \rightarrow \texttt{lockX}_3(s_{10}) \rightarrow \texttt{lockX}_4(s_4) \rightarrow \texttt{lockS}_2(s_6) \rightarrow \texttt{rel}_3(s_{10})$

(b) $\texttt{lockS}_4(p_1) \rightarrow \texttt{lockS}_3(p_5) \rightarrow \texttt{lockS}_1(s_4) \rightarrow \texttt{lockS}_2(p_6) \rightarrow \texttt{lockX}_4(s_5) \rightarrow \texttt{lockX}_3(p_2) \rightarrow \texttt{lockX}_2(p_1) \rightarrow \texttt{lockS}_1(s_1) \rightarrow \texttt{lockS}_4(s_7) \rightarrow \texttt{lockS}_1(p_4) \rightarrow \texttt{rel}_3(p_5) \rightarrow \texttt{lockX}_4(s_{10}) \rightarrow \texttt{lockS}_3(a_3) \rightarrow \texttt{rel}_4(s_5) \rightarrow \texttt{rel}_2(p_6) \rightarrow \texttt{rel}_4(p_1) \rightarrow \texttt{lockX}_1(s_6)$

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 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: | $\texttt{lockIS}_2(D)$ |
| 2: | $\texttt{lockIS}_2(a_1)$ |
| 3: | $\texttt{lockS}_2(p_2)$ |
| 4: | $\texttt{lockIX}_1(D)$ |
| 5: | $\texttt{lockIX}_1(a_3)$ |
| 6: | $\texttt{lockIX}_1(p_5)$ |
| 7: | $\texttt{lockX}_1(s_8)$ |
| 8: | $\texttt{lockIS}_1(a_2)$ |
| 9: | $\texttt{lockS}_1(p_3)$ |
| 10: | $\texttt{lockIS}_4(D)$ |
| 11: | $\texttt{lockIS}_4(a_4)$ |
| 12: | $\texttt{lockIS}_4(p_6)$ |
| 13: | $\texttt{lockS}_4(s_{11})$ |
| 14: | $\texttt{lockIX}_3(D)$ |
| 15: | $\texttt{lockIX}_3(a_4)$ |
| 16: | $\texttt{lockIX}_3(p_6)$ |
| 17: | $\texttt{lockX}_3(s_{10})$ |
| 18: | $\texttt{lockIX}_4(D)$ |
| 19: | $\texttt{lockIX}_4(a_2)$ |
| 20: | $\texttt{lockIX}_4(p_3)$ (1) |
| 21: | $\texttt{lockIS}_2(a_3)$ |
| 22: | $\texttt{lockIS}_2(p_4)$ |
| 23: | $\texttt{lockS}_2(s_6)$ |
| 24: | $\texttt{relX}_3(s_{10})$ |
| 25: | $\texttt{relIX}_3(p_6)$ |
| 26: | $\texttt{relIX}_3(a_4)$ |
| 27: | $\texttt{relIX}_3(D)$ |

**(b)**

| | |
|---|---|
| 1: | $\texttt{lockIS}_4(D)$ |
| 2: | $\texttt{lockIS}_4(a_1)$ |
| 3: | $\texttt{lockS}_4(p_1)$ |
| 4: | $\texttt{lockIS}_3(D)$ |
| 5: | $\texttt{lockIS}_3(a_3)$ |
| 6: | $\texttt{lockS}_3(p_5)$ |
| 7: | $\texttt{lockIS}_1(D)$ |
| 8: | $\texttt{lockIS}_1(a_2)$ |
| 9: | $\texttt{lockIS}_1(p_3)$ |
| 10: | $\texttt{lockS}_1(s_4)$ |
| 11: | $\texttt{lockIS}_2(D)$ |
| 12: | $\texttt{lockIS}_2(a_4)$ |
| 13: | $\texttt{lockS}_2(p_6)$ |
| 14: | $\texttt{lockIX}_4(D)$ |
| 15: | $\texttt{lockIX}_4(a_3)$ |
| 16: | $\texttt{lockIX}_4(p_4)$ |
| 17: | $\texttt{lockX}_4(s_5)$ |
| 18: | $\texttt{lockIX}_3(D)$ |
| 19: | $\texttt{lockIX}_3(a_1)$ |
| 20: | $\texttt{lockX}_3(p_2)$ |
| 21: | $\texttt{lockIX}_2(D)$ |
| 22: | $\texttt{lockIX}_2(a_1)$ |
| 23: | $\texttt{lockX}_2(p_1)$ (1) |
| 24: | $\texttt{lockIS}_1(a_1)$ |
| 25: | $\texttt{lockIS}_1(p_1)$ |
| 26: | $\texttt{lockS}_1(s_1)$ |
| 27: | $\texttt{lockIS}_4(p_5)$ |
| 28: | $\texttt{lockS}_4(s_7)$ |
| 29: | $\texttt{lockIS}_1(a_3)$ |
| 30: | $\texttt{lockS}_1(p_4)$ (2) |
| 31: | $\texttt{relS}_3(p_5)$ |
| 32: | $\texttt{relIS}_3(a_3)$ |
| 33: | $\texttt{lockIX}_4(a_4)$ |
| 34: | $\texttt{lockIX}_4(p_6)$ (3) |
| 35: | $\texttt{lockS}_3(a_3)$ (4) |

(a)

- Probleme bei Sperranforderungen:
  - (1): Die IX-Sperre kann wegen der Lesesperre von $T_1$ auf $p_3$ nicht gewährt werden.
- Nein, es kommt zu keinem Deadlock. Die Transaktion $T_4$ wartet auf eine Sperre die $T_1$ hält, die Transaktion $T_1$ kann aber weiterlaufen.
- Die folgenden Freigaben sind nötig damit $T_4$ weiterlaufen kann: $\texttt{relS}_1(p_3) \rightarrow \texttt{relIS}_1(a_2)$. Anschließend erhält $T_4$ die gewünschte Lesesperre, und jede der Transaktionen hat alle Einträge der Sequenz abgearbeitet.

(b)
- Probleme bei Sperranforderungen:
  - (1): Die Schreibsperre für $T_2$ auf $p_1$ kann wegen der Lesesperre von $T_4$ nicht gewährt werden.
  - (2): Die Lesesperre für $T_1$ auf $p_4$ kann wegen der IX-Sperre von $T_4$ nicht gewährt werden.
  - (3): Die IX-Sperre von $T_4$ auf $p_6$ kann wegen der Lesesperre von $T_2$ nicht gewährt werden.
  - (4): Die Lesesperre von $T_3$ auf $a_3$ kann wegen der IX-Sperre von $T_4$ nicht gewährt werden.
- Ja, es kommt zu einem Deadlock, da $T_4$ auf eine Sperre wartet, welche von $T_2$ gehalten wird, und $T_1$, $T_2$ und $T_3$ auf Sperren warten welche von $T_4$ gehalten werden. Da jedoch alle Transaktionen blockiert sind, wird keine ausgeführt werden, und somit kann keine der Sperren freigegeben werden.

**Aufgabe 7 (Zeitstempelbasiertes Sperrverfahren)** [3 Punkte]

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

$\texttt{BOT}_1 \rightarrow w_1(B) \rightarrow \texttt{BOT}_2 \rightarrow r_2(B) \rightarrow \texttt{BOT}_3 \rightarrow w_1(B) \rightarrow w_1(A) \rightarrow w_3(A) \rightarrow r_1(A) \rightarrow \texttt{BOT}_4 \rightarrow res? \rightarrow w_4(C) \rightarrow r_3(B) \rightarrow r_1(C) \rightarrow w_4(A) \rightarrow c_1 \rightarrow c_2 \rightarrow res? \rightarrow c_3 \rightarrow c_4$

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($C$) |
|---|---|---|---|---|---|---|---|

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:**

| # | $T$ | rTS($A$) | wTS($A$) | rTS($B$) | wTS($B$) | rTS($C$) | wTS($C$) |
|---|---|---|---|---|---|---|---|
| 1 | $\texttt{BOT}_1$ | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | $w_1(B)$ | 0 | 0 | 0 | 1 | 0 | 0 |
| 3 | $\texttt{BOT}_2$ | 0 | 0 | 0 | 1 | 0 | 0 |
| 4 | $r_2(B)$ | 0 | 0 | 3 | 1 | 0 | 0 |
| 5 | $\texttt{BOT}_3$ | 0 | 0 | 3 | 1 | 0 | 0 |
| 6 | $\texttt{reset}_1$ $[w_1(B)]$ | 0 | 0 | 3 | 1 | 0 | 0 |
| 7 | $w_3(A)$ | 0 | 5 | 3 | 1 | 0 | 0 |
| 8 | $\texttt{BOT}_4$ | 0 | 5 | 3 | 1 | 0 | 0 |
| 9 | $\texttt{BOT}_1$ | 0 | 5 | 3 | 1 | 0 | 0 |
| 10 | $w_1(B)$ | 0 | 5 | 3 | 9 | 0 | 0 |
| 11 | $w_1(B)$ | 0 | 5 | 3 | 9 | 0 | 0 |
| 12 | $w_1(A)$ | 0 | 9 | 3 | 9 | 0 | 0 |
| 13 | $r_1(A)$ | 9 | 9 | 3 | 9 | 0 | 0 |
| 14 | $w_4(C)$ | 9 | 9 | 3 | 9 | 0 | 8 |
| 15 | $\texttt{reset}_3$ $[r_3(B)]$ | 9 | 9 | 3 | 9 | 0 | 8 |
| 16 | $r_1(C)$ | 9 | 9 | 3 | 9 | 9 | 8 |
| 17 | $\texttt{reset}_4$ $[w_4(A)]$ | 9 | 9 | 3 | 9 | 9 | 8 |
| 18 | $c_1$ | 9 | 9 | 3 | 9 | 9 | 8 |
| 19 | $c_2$ | 9 | 9 | 3 | 9 | 9 | 8 |
| 20 | $\texttt{BOT}_3$ | 9 | 9 | 3 | 9 | 9 | 8 |
| 21 | $w_3(A)$ | 9 | 20 | 2 | 9 | 9 | 8 |
| 22 | $r_3(B)$ | 9 | 20 | 20 | 9 | 9 | 8 |
| 23 | $c_3$ | 9 | 20 | 20 | 9 | 9 | 8 |

(b) Ist die erzeugte Historie rücksetzbar?

**Lösung:**
**Nein.** Obwohl das verwendete Verfahren nicht garantiert, dass die erzeugten Historien rücksetzbar sind, könnte es ja in konkreten Fällen trotzdem sein, dass die resultierenden Historien rücksetzbar sind. Dies ist hier jedoch nicht der Fall: In Schritt 16 liest $T_1$ das Datum $C$, welches zuvor von $T_4$ geschrieben wurde. Rücksetzbarkeit verlangt nun, dass $T_1$ nicht vor $T_4$ committed. Genau das passiert aber in Schritt 18: $T_1$ macht ihr commit, obwohl $T_4$ zu diesem Zeitpunkt noch nicht committed wurde.

(c) Verwenden Sie nun die in der Vorlesung vorgestellte Variante der Zeitstempel-Synchronisationsmethode welche *strikte Historien* erzeugt. (Verwenden Sie die Methode mittels

*dirty bit.*) Im Falle eines Zurücksetzens sollen diesesmal – falls anwendbar – auch die Schreibzeitstempel zurückgesetzt werden. Lesezeitstempel bleiben wiederum unverändert.

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

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

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

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

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

**Lösung:**

| # | $T$ | rTS($A$) | wTS($A$) | d($A$) | rTS($B$) | wTS($B$) | d($B$) | rTS($C$) | wTS($C$) | d($C$) |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | $\texttt{BOT}_1$ | 0 | 0 | | 0 | 0 | | 0 | 0 | |
| 2 | $w_1(B)$ | 0 | 0 | | 0 | 1 | ✓(1) | 0 | 0 | |
| 3 | $\texttt{BOT}_2$ | 0 | 0 | | 0 | 1 | ✓(1) | 0 | 0 | |
| 4 | $\texttt{block}_2\ [r_2(B)]$ | 0 | 0 | | 0 | 1 | ✓(1) | 0 | 0 | |
| 5 | $\texttt{BOT}_3$ | 0 | 0 | | 0 | 1 | ✓(1) | 0 | 0 | |
| 6 | $w_1(B)$ | 0 | 0 | | 0 | 1 | ✓(1) | 0 | 0 | |
| 7 | $w_1(A)$ | 0 | 1 | ✓(1) | 0 | 1 | ✓(1) | 0 | 0 | |
| 8 | $\texttt{block}_3\ [w_3(A)]$ | 0 | 1 | ✓(1) | 0 | 1 | ✓(1) | 0 | 0 | |
| 9 | $r_1(A)$ | 1 | 1 | ✓(1) | 0 | 1 | ✓(1) | 0 | 0 | |
| 10 | $\texttt{BOT}_4$ | 1 | 1 | ✓(1) | 0 | 1 | ✓(1) | 0 | 0 | |
| 11 | $w_4(C)$ | 1 | 1 | ✓(1) | 0 | 1 | ✓(1) | 0 | 10 | ✓(4) |
| 12 | $\texttt{reset}_1\ [r_1(C)]$ | 1 | 0 | | 0 | 0 | | 0 | 10 | ✓(4) |
| 13 | $r_2(B)$ | 1 | 0 | | 3 | 0 | | 0 | 10 | ✓(4) |
| 14 | $w_3(A)$ | 1 | 5 | ✓(3) | 3 | 0 | | 0 | 10 | ✓(4) |
| 15 | $r_3(B)$ | 1 | 5 | ✓(3) | 5 | 0 | | 0 | 10 | ✓(4) |
| 16 | $\texttt{block}_4\ [w_4(A)]$ | 1 | 5 | ✓(3) | 5 | 0 | | 0 | 10 | ✓(4) |
| 17 | $c_2$ | 1 | 5 | ✓(3) | 5 | 0 | | 0 | 10 | ✓(4) |
| 18 | $\texttt{BOT}_1$ | 1 | 5 | ✓(3) | 5 | 0 | | 0 | 10 | ✓(4) |
| 19 | $w_1(B)$ | 1 | 5 | ✓(3) | 5 | 18 | ✓(1) | 0 | 10 | ✓(4) |
| 20 | $w_1(B)$ | 1 | 5 | ✓(3) | 5 | 18 | ✓(1) | 0 | 10 | ✓(4) |
| 21 | $\texttt{block}_1\ [w_1(A)]$ | 1 | 5 | ✓(3) | 5 | 18 | ✓(1) | 0 | 10 | ✓(4) |
| 22 | $c_3$ | 1 | 5 | | 5 | 18 | ✓(1) | 0 | 10 | ✓(4) |
| 23 | $w_4(A)$ | 1 | 10 | ✓(4) | 5 | 18 | ✓(1) | 0 | 10 | ✓(4) |
| 24 | $c_4$ | 1 | 10 | | 5 | 18 | ✓(1) | 0 | 10 | |
| 25 | $w_1(A)$ | 1 | 18 | ✓(1) | 5 | 18 | ✓(1) | 0 | 10 | |
| 26 | $r_1(A)$ | 18 | 18 | ✓(1) | 5 | 18 | ✓(1) | 0 | 10 | |
| 27 | $r_1(C)$ | 18 | 18 | ✓(1) | 5 | 18 | ✓(1) | 18 | 10 | |
| 28 | $c_1$ | 18 | 18 | | 5 | 18 | | 18 | 10 | |

(d) Ist die erzeugte Historie rücksetzbar?

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

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

- Nehmen Sie an, dass die Anfangswerte für die Werte `readTS` und `writeTS` aller drei Felder $A$, $B$ und $C$ jeweils 0 sind.
- Als Zeitstempel für die Transaktionen nehmen Sie bitte die # ihres des jeweiligen `BOT`-Eintrags.
- Wenn Sie am Ende der Historie angekommen sind, und nicht alle rückgestzten Transaktionen neu gestartet wurden (kein passender *res*? Eintrag), dann wird diese Transaktion einfach nicht erneut durchgeführt.

**Aufgabe 8 (Transaktionen in SQL)** [0.5 Punkte]

Betrachten Sie das folgende Relationenschema eines Unternehmens, in welchem durchgeführte Arbeiten und deren Verrechnung gespeichert werden:
(Primärschlüssel sind unterstrichen, Fremdschlüssel sind kursiv geschrieben).

```
Deliveries    ( package_id, sender, address, weight )
distributed   ( package_id: Deliveries.package_id )
customs_fees  ( package_id: Deliveries.package_id, 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 ist verlangt, dass eine Lieferung erst in Verteilung gehen darf, nachdem die erwartbaren Zölle für diese Lieferung abgeschätzt wurden. Diese Vorabschätzung erfolgt durch eine_n Vorarbeiter_in und wird durch eigene Transaktionen in die Datenbank eingetragen. Es ist erlaubt, dass andere Transaktionen parallel auf die Datenbank zugreifen können, solange nur abgeschlossene Änderungen der Transaktionen der Vorarbeiter_in nach außen sichtbar sind.

*Historie:*

| **Supervisor** | **Parallele Anfragen** |
|---|---|
| `BEGIN;`<br>`SET TRANSACTION ISOLATION LEVEL ____` | |
| `SELECT * FROM Deliveries WHERE ¬fee_determined` | |
| | `INSERT INTO Deliveries VALUES (12, ...);` |
| | `INSERT INTO Deliveries VALUES (13, ...);` |
| `SELECT * FROM Deliveries NATURAL JOIN distributed` | |
| `SELECT * FROM Deliveries NATURAL JOIN distributed`<br>`WHERE ¬distributed` | |
| `INSERT INTO customs_fees VALUES (5, ...);` | |
| `SELECT * FROM Deliveries WHERE ¬distributed` | |
| | `INSERT INTO Deliveries VALUES (10, ...);` |
| `INSERT INTO customs_fees VALUES (12, ...);` | `INSERT INTO Deliveries VALUES (11, ...);` |
| `SELECT * FROM Deliveries WHERE ¬distributed` | |
| `COMMIT;` | |

wobei `¬fee_determined` und `¬distributed` für die Bedingungen
`package_id NOT IN (SELECT package_id FROM customs_fees)` bzw.
`package_id NOT IN (SELECT package_id FROM distributed)` stehen.

**Lösung:**
Isolation Level: `READ COMMITTED`

Die Transaktion kann wie gewünscht durchgeführt werden.

(b) *Beschreibung:*

Ein Manager möchte eine Übersicht über alle Lieferungen für die bereits die Zölle abgeschätzt wurden, die jedoch noch nicht in Verteilung sind. Es ist entscheident, dass während diese Übersicht erstellt wird eine konsistente Datenbasis besteht, welche nicht durch parallele Schreibzugriffe verändert werden darf.

*Historie:*

| **Supervisor** | **Parallele Anfragen** |
|---|---|
| `BEGIN;`<br>`SET TRANSACTION ISOLATION LEVEL ____` | |
| `SELECT * FROM undistributed` | |
| | `INSERT INTO distributed VALUES ( ...);` |
| `SELECT sum(weight) FROM undistributed` | |
| | `INSERT INTO Deliveries VALUES ( ...);` |
| | `INSERT INTO Deliveries VALUES ( ...);` |
| `SELECT count(package_id) FROM undistributed` | |
| | `INSERT INTO customs_fees VALUES ( ...);` |
| `COMMIT;` | |

wobei `undistributed` für den Ausdruck
`Deliveries NATURAL JOIN customs_fee WHERE package_id NOT IN (SELECT package_id FROM distributed)` steht.

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

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

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

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

(c) *Beschreibung:*

Theresa und Boris haben sich Zugang zur Datenbank des Transportunternehms verschafft, und wollen alle Pakete die vom jeweils anderen verschickt worden sind abfangen, und an ihre eigene Adresse schicken (natürlich nur jene Lieferungen, die nicht bereits in Verteilung sind). Das System des Transportunternehmens verlangt allerdings (aus irgendeinem gutem, aber uns unbekannten Grund), dass neue Lieferungen nicht parallel angelegt werden dürfen, sondern immer eine klare Reihenfolge der Aktionen nachvollziehbar sein muss.

*Historie:*

| **Boris** | **Theresa** |
|---|---|
| | `BEGIN;`<br>`SET TRANSACTION ISOLATION LEVEL ____` |
| `BEGIN;`<br>`SET TRANSACTION ISOLATION LEVEL ____` | |
| `SELECT sum(weight)`<br>`FROM Deliveries('Theresa');` | |
| `INSERT INTO Deliveries VALUES`<br>`(8, 'Boris', 'Downing Street 10', 211.10);` | |
| | `SELECT sum(weight)`<br>`FROM Deliveries('Boris');` |
| | `INSERT INTO Deliveries VALUES`<br>`(9, 'Theresa', 'Posh Street 20', 99.99);` |
| `COMMIT;` | |
| | `COMMIT;` |

wobei `Deliveries(a)` für folgende Zeichenkette steht:

```
Deliveries WHERE sender=a AND package_id NOT IN (SELECT package_id FROM distributed)
```

**Lösung:**
Isolation Level: `SERIALIZABLE`

(Datenbank: PostgreSQL 11.5) Bei diesem Isolation Level kommt es zu einem Fehler wenn die Transaktion in Szenario3-b versucht zu committen.

Das Problem ist, dass die beiden Transaktionen nicht serialisierbar sind. Der Grund dafür ist, dass beide Transaktionen einen neuen Eintrag in die Tabelle `Deliveries` einfügen, der Wert in diesem neuen Eintrag jedoch vom Inhalt der Tabelle `Deliveries` abhängt. Das heißt, sowohl Boris als auch Thersa 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 Boris den Wert welchen Thersa einfügt, 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 eingfügten Wert berücksichtigen, wird abgebrochen.

Die Fehlermeldung lautet:

```
psql:Szenario3-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.
```

*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 `ıSzenario1-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.