Exercise Sheet 3 (WS 2020) – Sample Solution

6.0 VU Datenbanksysteme

About the exercises

General information

In this part of the exercises you apply the theoretical knowledge about transaction management, recovery, and multi-user synchronization that you have gained in the lecture.

We recommend you to solve the problems **on your own** (it is a good preparation for the exam – and also for your possible future job – to carry out the tasks autonomously). Please note that if we detect duplicate submissions or any plagiarism, both the "original" and the "copy" will be awarded 0 points.

Your submission must consist of a single, typeset PDF document (max. 5MB). **We do not accept PDF files with handwritten solutions.**

In total there are 8 tasks and at most 15 points that can be achieved on the entire sheet.

Deadlines

until 11.12.	**12:00 pm**:	Upload your solutions to TUWEL
on 12.01.	13:00 pm:	Evaluation and feedback is provided in TUWEL

Further questions – TUWEL forum

If you have any further questions concerning the contents or organization, do not hesitate to ask them on TUWEL forum. **Under no circumstances should you post (partial) solutions on the forum!**

Changes due to COVID-19

Due to the ongoing situation with COVID-19 we will not offer in-person office hours for the exercise sheets. If you have technical issues, trouble understanding the tasks on this sheet, or other questions please use the TUWEL forum.

We also recommend that you get involved in the forum and actively discuss with your colleagues on the forum. From experience we believe that this helps all parties in the discussion greatly to improve their understanding of the material.

Tasks: Recovery

This section is about the use of log records to ensure atomicity and durability of transactions. For log records the format presented in the lecture is used, which is summarized here:
Log records for actions performed by a transaction have the following format

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

where `LSN` indicates the LogSequenceNumber of the record, `TA` identifies the transaction performing the action and `PageID` specifies the page that was changed. `Redo` and `Undo` contain the redo/undo information and `PrevLSN` the LSN of the previous log record of the same transaction.

Redo- and *Undo*-information are recorded by describing the necessary changes using addition and subtraction (we will only deal with numerical values):

An example for such a log record is

$[\#i, T_j, P_X, X{+}{=}d_1, X{-}{=}d_2, \#k]$,

which indicates that according to the log record with the number i, the transaction T_j wrote a field X on page P_X. In order to redo this action, X needs to be increased by d_1, whereas to *undo* this step, X must be decreased by d_2. Finally, the previous log record of this transaction has number k.

Log records recording the begin of a transaction (BOT) and commit of transaction only contain the `LSN`, the `TA`, the type of operation (BOT or commit), and the `PrevLSN`. The corresponding log records thus have the following format:

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

Compensation Log Records (CLRs) are formatted as follows:

$\langle$`LSN, TA, PageID, Redo, PrevLSN, UndoNextLSN`$\rangle$,

with `UndoNextLSN` being the LSN of the next log record for the same transactions that needs to be undone. Like for the "standard" log records, also for BOT-CLRs the shortened format

$\langle$`LSN, TA, BOT, PrevLSN`$\rangle$

may be used.

Task 1 (Logging) [1 point]

Consider the schedule in Figure 1, which shows the three transactions T_1, T_2 and T_3. The capital letters A, B, C and D represent fields in the database and a_i, b_i, c_i and d_i are local variables of the different transactions. Moreover, $r_i(\Gamma, \gamma)$ denotes a read (the value of the field Γ is read from the database and stored in the local variable γ) and $w_i(\Gamma, \gamma)$ indicates a write (the value represented by γ is written to the field Γ in the database). `COMMIT` marks the successful termination of a transaction and `ROLLBACK` the rollback of a transaction. Assume that the rollback of a transaction takes places before the next action according to the schedule is performed, i.e. the rollback starts at the point marked with `ROLLBACK`, and completes successfully before the next action takes place. (e.g. the `ROLLBACK` in step 20 finishes before the read in step 21 takes place).

Finally, assume that at the start (line 1) relevant part of the database consists of the following values:

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

(a) For each line of the schedule where either a field in the database or a local variable is changed, provide the value of this field/variable *after* the operation. Provide the corre-

	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`		

Figure 1: Schedule for exercise 1.

sponding line number of the schedule (for all changes caused by the `ROLLBACK`, use line number 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) List the log records created by this schedule in the order of their creation. Use the format described at the beginning of this section. Recall that the redo- and undo information are supposed to be recorded using addition and subtraction. Assume that each field Γ is located on the page P_Γ. To increase readability, please use the style $\#i$ for the `LSN` and `PrevLSN`. If, for some record, no previous record exists, please use 0 as the previous LSN. Your list should also include the log records for the rollback of T_2.

Note: Format the log records in a readable way, e.g. in a real list (with one entry per row) or a table (again, one record per row). Do *not* write the log records as continuous text. If the answer is formatted in a way that is not readable, we may give 0 points for this exercise. (In case you use the LaTeX template provided, you can find a suggestion for a somewhat readable format there.)

Lösung:

	Log: `[LSN, TA, PageID, Redo, Undo, PrevLSN]` bzw. $\langle$`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]

Task 2 (Recovery) [1.5 points]

Suppose that after a crash of some database system you find the situation shown in Figure 2. The left side of the figure shows the content of the recovered logfile. On the right side, the content of the pages P_A, P_C, and P_D is illustrated.

Log-archive

[#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]

Pages in the background memory

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

P_C	LSN: #13
$C = 50$	

P_D	LSN: #11
$D = 121$	

Figure 2: Specification for Task 2: The content of the log archive (left) and the pages of the database (right) after some crash.

Use this information to carry out a recovery of the database.

(a) State the values of A, B, C, and D after the *redo*-step.

Lösung:

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

(b) State the Compensation Log Records (CLRs) created during the recovery.

Lösung:

Log:

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

(c) State the values of A, B, C, and D once the recovery is completed.

Lösung:

$$\boxed{A\colon\ 9;\quad B\colon\ 22\quad C\colon\ 11\quad D\colon\ 0}$$

Tasks: Concurrency Control

Task 3 (Properties of transactions) [2.6 points]

Consider the set $\mathcal{T}$ of transactions and the corresponding schedule $\mathcal{H}$, which is given by a sequence of basic operations:
$\mathcal{T} = \{T_1, T_2, T_3, T_4\}$
$\mathcal{H} = b_1 \to r_1(A) \to b_3 \to r_1(B) \to b_2 \to w_1(B) \to r_2(C) \to b_4 \to w_2(C) \to r_4(C) \to w_3(D) \to r_3(C) \to r_2(B) \to w_1(D) \to a_3 \to r_1(D) \to c_1 \to w_2(A) \to c_2 \to r_4(A) \to w_4(A) \to c_4$.

(a) Create the precedence graph (serializability graph) SG($\mathcal{H}$).

(b) For each edge in the precedence graph, provide at least one pair $p_i \to p_j$ of operations that justify the existence of this edge.

For the edges $T_1 \to T_2$, $T_2 \to T_3$, and $T_2 \to T_4$ (if they are part of the graph) list *all* pairs of operations that justify the existence of these edges.

(c) If the schedule is conflict serializable, state *one* possible conflict equivalent serial order of the transactions. If the schedule is not conflict serializable, state a minimal set of transactions that must be removed from the schedule to get a conflict serializable schedule. For the remaining schedule, state such a conflict equivalent serial schedule.

(d) List all pairs of transactions (T_i, T_j) in the schedule $\mathcal{H}$ such that T_j reads from T_i. For each such pair (T_i, T_j) state at least one pair $(w_i(X), r_j(X))$ of operations that make T_j reading from T_i.

(e) Determine which of the following properties hold for the schedule $\mathcal{H}$:

- Resettable
- Avoids cascading abort
- Strict

Justify your answer.

(f) For the following schedule, determine whether the schedule is conflict serializable, and determine which of the properties from part (e) of this task (Resettable, avoids cascading abort, and strict) are satisfied by the following schedule consisting of four transactions T_1, T_2, T_3 and T_4:

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

Again, justify/explain your answer.

Lösung:

(a) **Precedence graph:**

$$\begin{tikzcd} T_1 \arrow[r] \arrow[rr, bend left] & T_2 \arrow[r] & T_4 \end{tikzcd}$$

(b) **"Reasons" for the edges:**

$T_1 \to T_2$

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

$T_2 \to T_4$

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

$T_1 \to T_4$

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

(c) **Conflict serializable:**

Yes, the schedule **is** conflict serializable: The precedence graph does not contain cycles. One possible equivalent serial execution is

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

(d) **Reading dependencies:**

- T_2 reads from T_1: $(w_1(B), r_2(B))$
- T_3 reads from T_2: $(w_2(C), r_3(C))$
- T_4 reads from T_2: $(w_2(C), r_4(C))$ or $(w_2(A), r_4(A))$

(e) **Classification of the schedule:**

- **Resettable:**

  Yes, the schedule *is* resettable. A schedule is resettable if no transaction T_i `COMMITs` before a transactions from which T_i has read.

  Transaction T_2 reads fro T_1 and the `COMMIT` of T_2 comes after the `COMMIT` of T_1. Moreover, T_4 reads from T_2 and the `COMMIT` of T_4 comes afer the `COMMIT` of T_2.

- **Avoid cascading abort:**

  No. To avoid cascading aborts it is necessary that each operation reads a value written from a successfully committed transaction. The present schedule violates this condition. One example is $r_2(B)$ reads the value of $w_1(B)$, but T_1 is active at this point.

- **Strict Schedule:**

  No. Firstly, this is because the schedule does not avoid cascading aborts. Moreover, the counter example mentioned above $r_2(B)$ and $w_1(B)$ is also an example of a non strict schedule.

(f) **Properties of the second schedule:**

$$\begin{tikzcd}
T_1 \arrow[r] & T_2 \arrow[r, leftrightarrow] & T_3 \arrow[r, leftrightarrow] & T_4 \arrow[lll, bend right] \arrow[ll, bend left]
\end{tikzcd}$$

(Precedence graph (serializability graph) is not required.)

Reading dependencies: (not required by the task)

- 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: $(w_4(D), r_3(D))$

- The schedule is not conflict serializable. There is a cycle in the serializability graph.
- **Resettable:**

  Yes, the schedule is resettable. The reading dependencies are: T_1 reads from T_4, T_2 reads from T_3 and T_3 reads from T_4. The `COMMIT` sequence is: c_4 then c_1 then c_3 then c_2. This satisfies the condition of being resettable.
- **Avoids cascading aborts:**

  No, the schedule does not avoid cascading aborts. For instance, T_1 reads from T_4, but the `COMMIT` of T_4 occurs only after the reading directive on field C.
- **Strict schedule:**

  No. The schedule is not strict. This follows on the one hand that the schedule does not avoid cascading aborts. On the other hand, a concrete instance violating the property is that we have $w_1(A) \rightarrow w_2(A)$, yet, there is not `COMMIT` of T_1 in between.

Task 4 (Locks and Deadlocks) [2.5 points]

Consider the sequence of lock requests shown below, where "$\texttt{lockS}_i(O)$" (resp. "$\texttt{lockX}_i(O)$") indicates a transaction T_i requesting a shared (resp. an exclusive) lock on an object O, and "$\texttt{rel}_i(O)$" describes a transaction T_i releasing all locks on O:
$\texttt{lockX}_3(D) \rightarrow \texttt{lockS}_1(A) \rightarrow \texttt{lockX}_2(B) \rightarrow \texttt{lockS}_4(A) \rightarrow \texttt{lockX}_2(A) \rightarrow \texttt{lockX}_4(C) \rightarrow \texttt{lockS}_3(B) \rightarrow \texttt{rel}_4(A) \rightarrow \texttt{lockS}_1(D) \rightarrow \texttt{lockX}_2(C)$.

(a) Assume some DBMS receives the above sequence of lock requests and works through them in the given order for the transactions T_1, T_2, T_3 and T_4. Whenever some request cannot be granted to a transaction, this transaction is blocked (i.e. its subsequent lock requests are postponed until the transaction is revoked).

State the order in which the DBMS works through the lock requests, and state immediately after each request whether the lock is granted or whether the transaction has to wait for it. Use $\texttt{grantS}_i(O)$ and $\texttt{grantX}_i(O)$ to denote that a shared- resp. exclusive lock on an object O is granted, use $\texttt{wait}(i)$ to indicate that a transaction was blocked to wait for a lock, use $\texttt{relS}_i(O)$ and $\texttt{relX}_i(O)$ to show that a shared- resp. exclusive lock on object O was released (as the result of some $\texttt{rel}_i(O)$), and use $\texttt{resume}(i)$ to indicate that some transaction was revoked because the requested lock is now available and is granted.

To determine the correct order, assume that once a blocked transaction is revoked, it catches up all "skipped" operations, i.e. all of its omitted operations are performed, until either the transaction blocks again, or there are no more omitted operations for this transactions. Only if one of these two conditions is reached, proceed with working through the actions on the original release.

Example: Consider the sequence

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

of lock requests of two transactions T_1, T_2.

We obtain the following list:

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:	`lockX`$_3(D)$
2:	`grantX`$_3(D)$
3:	`lockS`$_1(A)$
4:	`grantS`$_1(A)$
5:	`lockX`$_2(B)$
6:	`grantX`$_2(B)$
7:	`lockS`$_4(A)$
8:	`grantS`$_4(A)$
9:	`lockX`$_2(A)$
10:	`wait`(2)
11:	`lockX`$_4(C)$
12:	`grantX`$_4(C)$
13:	`lockS`$_3(B)$
12:	`wait`(3)
14:	`rel`$_4(A)$
15:	`relS`$_4(A)$
16:	`lockS`$_1(D)$
17:	`wait`(1)

(b) Sketch the current situation of held locks resp. blocked transactions. Therefor provide a table as shown below. Into each field of the table, insert an X (a S) if the corresponding transaction has an exclusive (resp. a shared) lock on this object. For each blocked transaction fill in a WS (*wait shared*) or WX (*wait exclusive*) into the corresponding field for the lock request that blocked the transaction.

Lösung:

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

(c) Show the wait ("wait-for") graph for the current situation.

Lösung:

wait-for graph:

$$\begin{tikzcd}
& T_1 \arrow[dl] & \\
T_3 \arrow[rr] & & T_2 \arrow[ul] \\
& T_4 &
\end{tikzcd}$$

(d) State whether there currently exists a deadlock. **Solution: Yes**

(e) State one possible sequence of releases that releases all locks currently held. If releasing such a lock makes it possible to continue a transaction, then all lock- and release requests by this transactions have to be handled before additional releases may be defined. If there currently exists a deadlock, abort transaction T_3 (i.e. all locks of T_3 are released immediately).

Lösung:
As there is currently a deadlock, transaction T_3 is aborted. This leads to the following release:

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

Due to this release T_1 can continue and we get:

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

Now T_1 can release its locks again.

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

Thus transaction T_2 can proceed

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

Now T_4 can release the locks

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

Which leads to the fact that T_2 gets the last lock

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

and subsequently can release all remaining locks.

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

(f) Consider the given sequence of lock requests and releases. Does it violate two-phase locking (2PL)? What about the sequence you created in task (e)?

Lösung:
No. In both cases no transaction requests a lock after releasing the first lock. This meets the requirements of 2PL.

Task 5 (Two-Phase Locking) [1.5 points]

Consider the following three transactions T_1, T_2, and T_3, and the respective sequence of operations ($r_i(O)$ and $w_i(O)$ denote a read- resp. write operation by T_i on O, and c_i marks the commit of T_i).

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

Assume that *("normal") 2 Phase Locking* is applied to these transactions. State the resulting schedule (consisting of the lock requests, the read- and write operations, the releases of the locks, and the commits).

Make the following assumptions:

- *Notation:* Please use $\texttt{lockS}_i(O)$ and $\texttt{lockX}_i(O)$ to denote a request for a shared- resp. exclusive lock on object O by transaction T_i. Please also use $\texttt{rel}_i(O)$ to indicate the release of all locks by T_i on O. (*Hint:* You don't need to state explicitly whether a lock is granted or not. This information is given implicitly by the transaction performing a corresponding read resp. write afterwards – or not.)

- *Lock requests and releasing locks:* For each operation (read, write, commit), state the required lock requests (unless the transaction already holds the required locks). In doing so, assume that locks are requested as economical as possible, which means:
  - A lock is only requested if it is really needed.
  - A lock is held as short as possible, i.e. locks are requested at the latest possible time and released at the earliest possible point in time.

- *Scheduling of the transactions:* Assume that each transaction performs one operation (read, write, commit) before the next transaction is scheduled. I.e., in our case the real sequence of actions is $w_1(C) \rightarrow w_2(A) \rightarrow r_3(C) \rightarrow r_1(B) \rightarrow \ldots$. Lock requests and releasing locks do not count as operations, i.e. before and after each operation (read, write, commit), the transaction is allowed to request or release an arbitrary number of locks, before it is the next transactions turn.

  Deviations from this sequence are only allowed if one transaction is blocked. In such a case, the transaction is omitted every time it is blocked. This is continued until the block on the transaction is lifted. Once this happens, the transaction again takes her turns in the schedule as before – again only performing one operation (read, write, commit) per turn.

  The following example demonstrates the control flow for two transactions: Assume T_1 consists of actions $\alpha_1, \alpha_2, \ldots$, and T_2 of $\beta_1, \beta_2, \ldots$. The normal sequence is $\alpha_1, \beta_1, \alpha_2, \beta_2, \ldots$. Assuming T_2 requires a lock for β_3 that is held by T_1, i.e. T_2 blocks, then the sequence would continue $\alpha_3, \alpha_4, \alpha_5, \ldots$. If the lock is released after α_5, then the sequence continues with $\alpha_5, \beta_3, \alpha_6, \beta_4, \ldots$.

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

Task 6 (Multi-Granularity Locking) [2.4 points]

Consider the hierarchy within a database depicted in Figure 3.

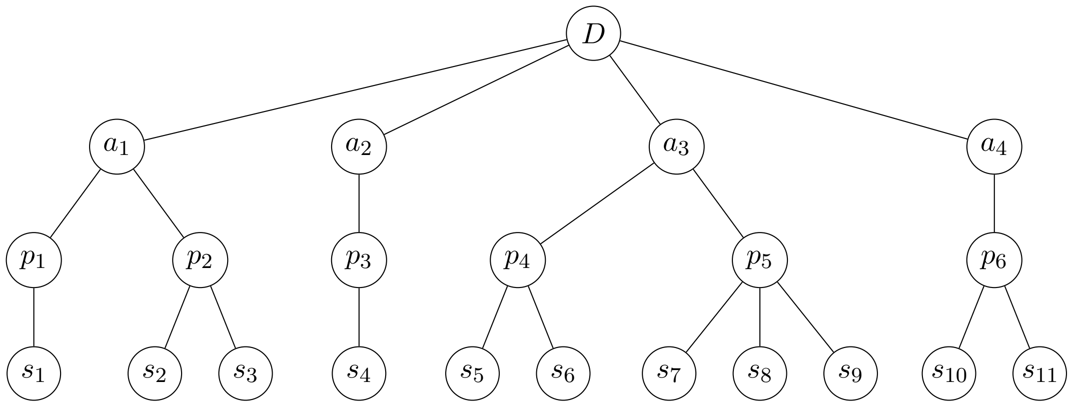

Figure 3: Hierarchy of database objects for Task 6

Consider the following sequences of lock requests and releases of locks of four transactions T_1, T_2, T_3, and T_4 on the resources described in Figure 3.

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

Within these sequences, $\texttt{lockS}_i(O)$ denotes transaction T_i requesting a shared lock on object O, $\texttt{lockX}_i(O)$ denotes transaction T_i requesting an exclusive lock on O, and $\texttt{rel}_i(O)$ denotes transaction T_i releasing all locks held on O.

Answer the following questions for both sequences:

- Show how one has to proceed to process the lock requests and releases of locks in accordance to the Multi Granularity Locking (MGL) protocol: State the sequence of required lock requests, and in the case of a release state which further locks can be released as a result. *Hint:* Make sure to keep the required order in both cases, when requesting and releasing a lock.

  Please use the following notation: $\texttt{lockIS}_i(O)$, $\texttt{lockIX}_i(O)$, $\texttt{lockS}_i(O)$, and $\texttt{lockX}_i(O)$ for transaction T_i requesting a IS-, IX-, S- resp. X-lock on object O; $\texttt{relIS}_i(O)$, $\texttt{relIX}_i(O)$, $\texttt{relS}_i(O)$, and $\texttt{relX}_i(O)$ for transaction T_i releasing a IS-, IS-, S- resp. X-lock on object O. Make also sure that a transaction only requires locks it does not already hold.

- Mark lock requests that cannot be granted. Assume that in such a case the corresponding transaction is blocked, i.e. no further actions of the transaction are performed until the lock is granted and the block on the transaction is lifted. This may happen because of another transaction releasing the lock (Lock requests and releases of locks of blocked transactions are omitted). Once the block on the transaction is lifted, all heretho omitted actions are executed before the processing of the sequence is continued at the position where the lock was released.

- For each declined lock, state why this lock is not granted.
- Does a deadlock occur as a result of this sequence? If this is the case, explain why.
- If no deadlock occurs, but there is at least one transaction blocked at the end of the sequence: Provide a minimal number of locks that must be released to continue these transactions. By doing that, note that transactions can only release shared and exclusive locks explicitly; but provide also the IX- and IS-locks which are released implicitly by them. (Don't forget to maintain the right order.)

  Once the blocked transaction is revoked, continue the execution of this transaction. If this makes the transaction block again, state again a minimal number of locks that need to be released in order to continue with processing the transaction.

Hint:

- If the situation occurs that a transaction acquires a lock it already holds on one or more children of this node, you may assume that these locks are automatically removed from the child nodes (you need not state the release of these locks explicitly).

Lösung:

(a)

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

(b)

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

(a)
- Problems with lock requests
  - (1): The IX-lock cannot be granted due to the shared lock of T_1 on p_3.
- No, there is no deadlock. The transaction T_4 waits on a lock by T_1, however T_1 itself can continue.
- The following locks need to be released so that T_4 can resume: relS$_1(p_3) \rightarrow$ relIS$_1(a_2)$. Subsequently T_4 will be granted its shared lock request, and thus all transactions can execute their respective operations in the sequence.

(b)
- Problems with lock requests

  - (1): The shared lock request of T_2 on p_1 cannot be granted due to the existing shared lock of T_4.
  - (2): The shared lock request by T_1 on p_4 cannot be granted due to the existing IX-lock of T_4.
  - (3): The IX lock request of T_4 on p_6 cannot be granted due to an existing shared lock of T_2.
  - (4): The shared lock request of T_3 on a_3 cannot be granted due to the existing IX-lock of T_4.
- Yes, there is a deadlock, since T_4 is waiting on a lock held by T_2, and T_1, T_2 and T_3 are each waiting on locks held by T_4. So since all transaction are blocked, none will be executed and no locks will be released.

Task 7 (Timestamp based Locking) [3 points]

Consider the following sequence of operations of four transactions T_1, T_2, T_3, and T_4 which access three objects A, B, and C.

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

In this sequence, $\texttt{BOT}_i$ denotes the start of a transaction i, $r_i(X)$ denotes a read (transaction T_i reads object X), $w_i(X)$ denotes a write (transaction T_i writes object X), and c_i denotes the successful termination (commit) of transaction i. Entries *res*? indicate that at these points in time, one transaction shall be restarted in case some transaction has been reset earlier but was not yet restarted. In case there is more than one such transaction, restart the transaction that was reset first. After the restart, execute all operations of this transaction prior to the current position (i.e. the position of the current *res*?) in the sequence.

(a) Use the concurrency control protocol based on timestamps discussed in the lecture to create a valid (according to this protocol) schedule. Use the version of the protocol that not necessarily creates recoverable schedules. I.e. writes are performed immediately and the access to such fields is controlled only by the read- and write timestamps (i.e. transaction are either reset or get access, but are not blocked).

In case some transaction is reset, you need not care about whether other transactions are affected by this rollback. Only the current transaction is reset, and no cascading rollback is performed (even if possible).

In case of such a rollback, both the read- and write TS of all fields remain unchanged.

Please state the resulting schedule as a table with the following columns:

#	action	rTS(A)	wTS(A)	rTS(B)	wTS(B)	rTS(C)	wTS(C)

The column `#` should contain an increasing number. For the column `action`, please use the format for `BOT` actions, `COMMIT` actions, as well as reads and writes described above and also used in the description. If a transaction is reset, please use $\texttt{reset}_i$ for the corresponding record. In the remaining columns, please state the values of the read- and write TS after the execution of the corresponding action.

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) Is the created schedule recoverable?

Lösung:
No. Even though the used procedure does not guarantee that the resulting histories are recoverable, it is still possible that in concrete examples the result is actually recoverable. In the example above, that is however not the case: In step 16 T_1 reads the data C,which was written to by T_4. Recoverability would now require that T_1 does not commit before T_4, however we see that exactly that happens in step 18, when T_1 commits, despite T_4 still running at that point.

(c) Next, use the variant of the protocol that guarantees *strict schedules.* (Apply the variant using a *dirty bit.*) In case a transaction is reset – if applicable – the write timestamps shall be reset as well. Read timestamps remain unchanged.

Like for exercise a), state the resulting schedule in a table. In addition to the information already provided in exercise a), also provide these information:

- state for each field A, B, and C whether the dirty bit is set or not. I.e., use a table with the following columns:

#	action	wTS(A)	d(A)	rTS(B)	wTS(B)	d(B)	rTS(C)	wTS(C)	d(C)

- If a transaction T_i is blocked, add $\texttt{block}_i$ to the schedule.

Lösung:

#	T	$\texttt{rTS}(A)$	$\texttt{wTS}(A)$	$\texttt{d}(A)$	$\texttt{rTS}(B)$	$\texttt{wTS}(B)$	$\texttt{d}(B)$	$\texttt{rTS}(C)$	$\texttt{wTS}(C)$	$\texttt{d}(C)$
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) Is the created schedule recoverable?

Lösung:
Yes. The used synchronisation method guarantees that the resulting histories are strict. The answer follows immediately, as every strict history is by definition also recoverable.

To solve these exercises, please consider the following assumptions and conventions:

- Assume that the initial values of `readTS` and `writeTS` of each, A, B, and C is 0.
- As timestamps for the transactions, use the # of the corresponding `BOT` record.

- When you reach the end of the schedule and there are still transactions that have been reset but not yet restarted (i.e. no suitable *res*? record exists), then this transaction is just not restarted.

Task 8 (Transactions in SQL) [0.5 points]

Consider the following relational schema of a company, that contains records of performed tasks and charging information on these tasks:
(Primary keys are underlined, foreign keys are italic).

```
Deliveries    ( package_id, sender, address, weight )
distributed   ( package_id: Deliveries.package_id )
customs_fees  ( package_id: Deliveries.package_id, amount )
```

Identify for every scenario described below the *lowest* isolation level that offers the required degree of isolation, respectively. Describe also, whether transactions can run with the given concurrency in their respective isolation level as desired, or whether problems are occurring. Due to lack of space, below we provide only sketches of the queries/transactions. You can find the complete queries in the available SQL files.

(a) *Description:*

It is asked, that each delivery may only be distributed after an estimation of its customs fees has been entered into the database. This estimation is done by a supervisor and is entered into the database through its own transaction. It is permitted that other transactions access the database concurrently, as long as changes by the supervisor are only visible after their transaction commits.

Schedule:

Supervisor	**Parallel Queries**
`BEGIN;` `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 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;`	

where `¬fee_determined` and `¬distributed` stand for
`package_id NOT IN (SELECT package_id FROM customs_fees)` resp.
`package_id NOT IN (SELECT package_id FROM distributed)`.

Lösung:
Isolation Level: `READ COMMITTED`

The transaction can be executed as desired.

(b) *Description:* A manager wants an overview over all deliveries for which customs fees have been estimated, but which are not yet in distribution. It is of key importance that there is a consistent state of the database while this overview is being created, a state which may not be manipulated by parallel write accesses.

Schedule:

Supervisor	**Parallel Queries**
`BEGIN;` `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;`	

where `undistributed` stands for the expression
`Deliveries NATURAL JOIN customs_fee WHERE package_id NOT IN (SELECT package_id FROM distributed)`.

Lösung:
The correct isolation level may vary among the “usual” DMBS. Generally, it is asked that an isolation level should be chosen where phantom reads are excluded.

According to the SQL standard, this is only the case in the isolation level `SERIALIZABLE`.

As was mentioned in the lectures, however, it is possible in some DBMS, such as PostgreSQL, that already the isolation level `REPEATABLE READ` suffices to make phantom reads disappear.

In any case, with such an isolation level, the transaction should be able to be executed as was desired.

(c) *Description:*

Theresa and Boris have somehow gained access to the database of the logistics company. Each of the two wants to find all deliveries send by the other, and reroute them to their own address (naturally only the deliveries with are not already in distribution). The system used by the logistic company requires (for some inexplicable reason), that new deliveries are only entered sequentially, never in parallel.

Schedule:

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

where `Deliveries(a)` stands for the string

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

Lösung:
Isolation Level: `SERIALIZABLE`

(used database: PostgreSQL 11.5)

with this isolation level, there is still an error when the transaction in Szenario3-b tries to commit.

The issue is that both transactions, together, are not serializable. The reason is that each transaction would ultimately perform a different action, depending on which was executed first. This interdependency is recognized by the DMBS, and the transaction which commits last, and thus would need to access information written by the other transaction, is aborted.

The error message is:

```
psql:Szenario3-b-muster.sql:23:  ERROR: could not serialize access due to read/write
dependencies among transactions
DETAIL: Reason code:  Cancelled on identification as a pivot, during commit
attempt.
```

Please note: You can test these scenarios with the SQL files on a DBMS (the files are written for Postgres, but they can be tested with minor changes on another DBMS). With Postgres, please proceed as follows:

1. Open a database console by the command `psql`.
2. In another terminal, open a second database console by `psql`.
3. In one terminal, load the scenario by `\i Szenario1-a.sql` (where “1” should be replaced by the desired number)
4. In the other terminal, load the parallel scenario by `ıSzenario1-b.sql` (where “1” should be replaced by the desired number)

5. The execution of the SQL commands should be interrupted in both terminals with the message `Press Enter to continue (i)`
6. Press [Enter] in that console where `i` has the smaller value.

If the result at a chosen isolation level differs from your expectations, feel free to document this. In this case please provide the DBMS where you executed the transactions.

You can either use a database on your machine, or you can use our server `bordo`. You can connect to the server via ssh and then use `psql` to get access to a PostgreSQL database. You can find additional information about how you can connect to the server and log in into the database in TUWEL. Your login information are provided in TUWEL.