Exercise Sheet 3 (WS 2019) – Sample Solution

3.0 VU Datenmodellierung 2 / 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, typset 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
07.01.	13:00 pm	Evaluation and feedback is provided in TUWEL

Consultation Hours (optional)

In the week before the deadline there are consultation hours held by tutors. If you have problems understanding the topics of the exercise sheet or questions about the exercises, you are welcome to just drop in at these consultation hours. The tutors will gladly answer your questions and help you understand the subjects.

The goal of these consultation hours is to help you with **understanding the topics and specific tasks** of the exercise sheet. The tutors will not solve your exercises or check your answers before you hand them in.

Participation is completely voluntary — dates and locations of the consultation hours can be found in TUWEL.

Debriefing (optional)

A few days after you received your feedback and grading of this exercise sheet, there is the opportunity to go through the tasks in small groups (max. 25 persons). The (relative) small group size is intended to enable an active participation. Each of these groups will be held by an assistant. The specific agenda of these meetings will depend on the interests and question of the participants (i.e., you). The main objectives are answering your open questions and resolving your remaining issues regarding the exercises. Therefore, please have a look at your feedback and evaluation to identify such problems before you attend the class. When participating, dare to ask your questions – no question has a (negative) impact on your grade.

Participation is absolutely voluntary. Registration in TUWEL is required to keep the size of the groups small. Dates and locations can be found 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.

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:
⟨`LSN, TA, PageID, Redo, PrevLSN, UndoNextLSN`⟩,
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
⟨`LSN, TA, BOT, PrevLSN`⟩
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 21 finishes before the write in step 22 takes place).

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

Figure 1: Schedule for exercise 1.

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

$$A\colon 10 \qquad B\colon 57 \qquad C\colon 31 \qquad D\colon 43$$

(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 corresponding line number of the schedule (for all changes caused by the ROLLBACK, use line number 21).

Lösung:

	Wert
3	$a_2 = 10$
4	$B = 25$
6	$b_2 = 25$
7	$c_3 = 31$
8	$A = 5$
9	$B = 18$
10	$c_1 = 31$
11	$D = 35$
12	$C = 26$
13	$C = 46$
14	$d_1 = 35$
15	$d_3 = 35$
16	$a_2 2 = 5$
17	$b_3 = 18$
19	$D = 13$
20	$A = 5$
21	$D = 35$
21	$C = 51$
21	$B = 25$
22	$d_2 = 35$
23	$A = 30$

(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_3.

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_2, `BOT`, #0]
2	[#2, T_1, `BOT`, #0]
4	[#3, T_1, P_B, B-=32, B+=32, #2]
5	[#4, T_3, `BOT`, #0]
8	[#5, T_2, P_A, A-=5, A+=5, #1]
9	[#6, T_3, P_B, B-=7, B+=7, #4]
11	[#7, T_2, P_D, D-=8, D+=8, #5]
12	[#8, T_3, P_C, C-=5, C+=5, #6]
13	[#9, T_1, P_C, C+=20, C-=20, #3]
18	[#10, T_1, `COMMIT`, #9]
19	[#11, T_3, P_D, D-=22, D+=22, #8]
20	[#12, T_2, P_A, A+=0, A-=0, #7]
21	$\langle$#13, T_3, P_D, D+=22, #11, #8$\rangle$
21	$\langle$#14, T_3, P_C, C+=5, #13, #6$\rangle$
21	$\langle$#15, T_3, P_B, B+=7, #14, #4$\rangle$
21	$\langle$#16, T_3, `BOT`, #15$\rangle$
23	[#17, T_2, P_A, A+=25, A-=25, #12]
24	[#18, T_2, `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_B, and P_D is illustrated.

Log-archive

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

Pages in the background memory

P_A	LSN: #5
$A = 50$	

P_B	LSN: #13
$B = 70$	$C = 111$

P_D	LSN: #6
$D = 66$	

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

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

Lösung:

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

$\langle$#22, T_4, P_B, B-=183, #18, #14$\rangle$
$\langle$#23, T_3, P_C, C+=206, #16, #6$\rangle$
$\langle$#24, T_4, P_D, D-=5, #22, #7$\rangle$
$\langle$#25, T_4, `BOT`, #24$\rangle$
$\langle$#26, T_3, P_D, D-=111, #23, #3$\rangle$
$\langle$#27, T_2, P_A, A-=1, #15, #1$\rangle$
$\langle$#28, T_3, `BOT`, #26$\rangle$
$\langle$#29, T_2, `BOT`, #27$\rangle$

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

Lösung:

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

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:
$\mathcal{T} = \{T_1, T_2, T_3, T_4, T_5, T_6, T_7\}$
$\mathcal{H} = b_1 \to r_1(A) \to b_2 \to r_2(B) \to r_2(A) \to w_2(B) \to b_3 \to w_3(C) \to b_7 \to w_1(A) \to r_7(C)$
$\to b_6 \to b_4 \to r_6(A) \to w_4(A) \to r_7(A) \to b_5 \to w_6(A) \to r_3(A) \to r_7(B) \to r_6(B) \to r_7(C)$
$\to w_5(D) \to w_6(C) \to a_6 \to a_1 \to r_5(B) \to r_2(C) \to r_5(C) \to c_2 \to r_3(C) \to w_5(C) \to c_3$
$\to w_5(B) \to c_7 \to c_5 \to w_4(D) \to c_4$.

(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 $T_2 \to T_5$, $T_3 \to T_6$, and $T_4 \to T_7$ (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 $\mathcal{H}$:

- Recoverable
- Avoids cascading abort
- Strict

Justify your answer.

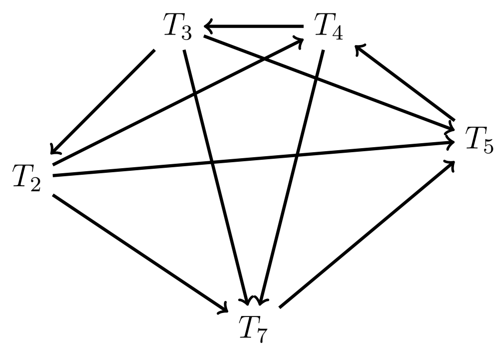
(f) Determine which of the properties from part (e) of this task (Recoverable, avoids cascading abort, and strict) are satisfied by the following schedule consisting of four transactions T_1, T_2, T_3 and T_4:

$b_1 \to b_2 \to b_3 \to b_4 \to r_3(A) \to w_2(A) \to w_4(C) \to r_1(B) \to r_2(A) \to w_4(C) \to r_2(A) \to a_4 \to w_3(C) \to w_2(B) \to c_2 \to r_1(A) \to w_3(B) \to r_3(A) \to w_1(C) \to c_3 \to r_1(C) \to r_1(B) \to c_1$

Again, justify/explain your answer.

Lösung:

(a) **Precedence graph:**



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

$T_2 \to T_4$

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

$T_2 \to T_5$

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

$T_2 \to T_7$

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

$T_3 \to T_2$

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

$T_3 \to T_5$

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

$T_3 \to T_7$

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

$T_4 \to T_3$

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

$T_4 \to T_7$

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

$T_5 \to T_4$

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

$T_7 \to T_5$

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

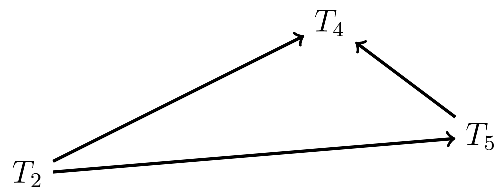
(c) **Conflict serializable:**

No, the schedule **is not** conflict serializable: The precedence graph contains several cycles, e.g.

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

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

Aborting only one transaction is not enough. More precisely at least two transactions must be canceled in order to obtain a conflict serializable schedule. There are several possible solutions, e.g. abort T_3 and T_7. The resulting precedence graph



(This precedence graph was not asked in the exercises!)

The schedule of the remaining transactions is conflict serializable. One possible equivalent serial execution is

$$T_2 \text{ before } T_5 \text{ before } T_4$$

(d) **Reading dependencies:**

- T_2 reads from T_3: $(w_3(C), r_2(C))$ ($w_6(C)$ has been reset)
- T_3 reads from T_6: $(w_6(A), r_3(A))$
- T_5 reads from T_2: $(w_2(B), r_5(B))$
- T_5 reads from T_3: $(w_3(C), r_5(C))$ ($w_6(C)$ has been reset)
- T_6 reads from T_2: $(w_2(B), r_6(B))$
- T_7 reads from T_2: $(w_2(B), r_7(B))$
- T_7 reads from T_3: $(w_3(C), r_7(C))$, $(w_3(C), r_7(C))$
- T_7 reads from T_4: $(w_4(A), r_7(A))$

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

- **Recoverable:**

  No, the schedule *is not* recoverable. A schedule is recoverable if no transaction T_i `COMMITs` before a transactions from which T_i has read.

  Except from T_6, which reads from T_2 and aborts (therefore, never `COMMITs`) and T_7 which `COMMITs` before T_4 and depends on the value read from T_4, all reading transactions commit before transactions from which they read.

- **Avoid cascading abort:**

  No. Clearly this results from the fact that the schedule is not recoverable. However, there is also a concrete counter example. 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 in many cases. One example can be found at the end of the first row: $r_7(c)$ reads a value of $w_3(C)$, although T_3 is still working at that time.

- **Strict Schedule:**

  No. Firstly, this is because the schedule does not avoid cascading aborts. Moreover, the counter example mentioned above $r_7(C)$ and $w_3(C)$ is also an example of a non strict schedule. A further example which additionally violates the condition for a strict schedule is the operation $w_5(C)$ at the end of the third row. This operation overwrites $w_3(C)$, but T_3 is not finished yet.

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

- **Recoverable:**

  Yes, the schedule is recoverable T_1 and T_3 read from T_2 and T_2 commits before T_1 and T_3. Additionally, T_1 reads from T_3, but commits after T_3.

- **Avoids cascading aborts:**

  Yes, the schedule avoids cascading aborts. Each reading access to a date written by another transaction occurs after the commit of the writing transaction.

- **Strict schedule:**

  No. The schedule is not strict, because T_1 overwrites the value C before T_3 (the transaction which wrote the value before T_1) commits. I.e. we found $w_3(C)$ before $w_1(C)$, without a c_3 before $w_1(C)$.

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

(a) Assume some DBMS receives the above sequence of lock requests and works through them in the given order. 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

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

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

(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.

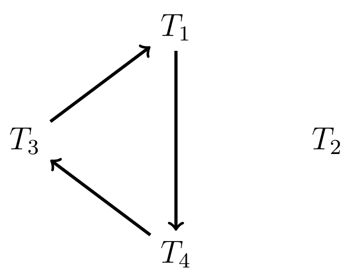
Lösung:

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

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

Lösung:

wait-for graph:



(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 revokes a transactions, 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_4 (i.e. all locks of T_4 are released immediately).

Lösung:
As there is currently a deadlock, transaction T_4 is aborted. This leads to the following release:

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

Due to this release T_1 can continue and we get:

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

Now T_1 can release its locks again.

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

Thus transaction T_3 can proceed

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

Now T_2 can release the locks

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

Which leads to the fact that T_3 gets the last lock and subsequently also all locks can be released.

$$\texttt{resume}(3) \rightarrow \texttt{grantX}_3(B) \rightarrow \texttt{rel}_3(C) \rightarrow \texttt{relS}_3(C) \rightarrow \texttt{rel}_3(B) \rightarrow \texttt{relX}_3(B)$$

(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).

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

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 operations (read, write, commit) before the next transaction is scheduled. I.e., in our case the real sequence of actions is $r_1(A) \rightarrow r_2(A) \rightarrow w_3(C) \rightarrow r_1(B) \rightarrow \ldots$. 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 transaction is revoked. 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 $\alpha_5, \beta_3, \alpha_6, \beta_4, \ldots$.

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

Consider the hierarchy within a database depicted in Figure 3.

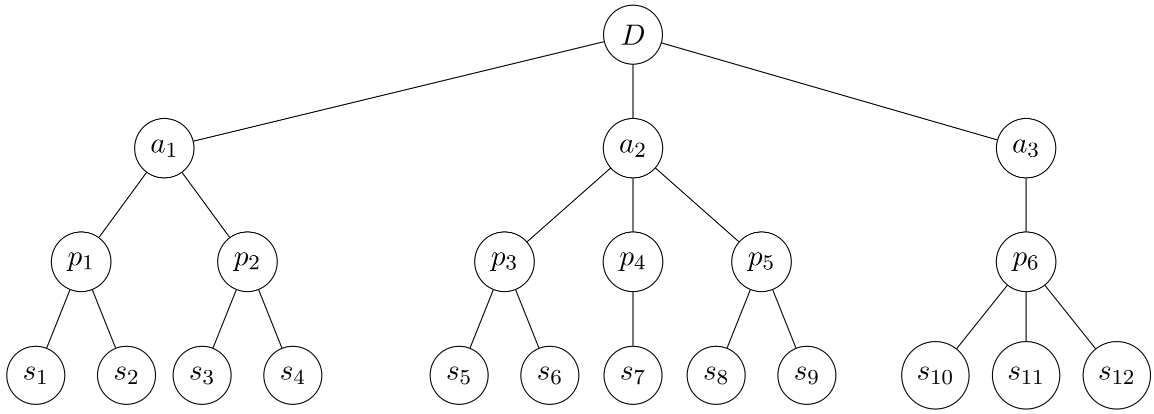


Figure 3: Hierarchy of database objects for Task 6

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) $\texttt{lockS}_1(p_2) \rightarrow \texttt{lockX}_2(s_9) \rightarrow \texttt{lockS}_3(p_6) \rightarrow \texttt{lockX}_1(p_3) \rightarrow \texttt{lockS}_3(s_4) \rightarrow \texttt{lockX}_2(s_3) \rightarrow \texttt{lockS}_4(a_1) \rightarrow \texttt{rel}_1(p_2)$

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

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 transaction revoked. This may happen because of another transaction releasing the lock (Lock requests and releases of locks of blocked transactions are omitted). Once the transaction is revoked, all omitted actions are made up for 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).

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.

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

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 `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.

(b) Is the created schedule recoverable?

(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 `block`$_i$ to the schedule.

(d) Is the created schedule 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 restartet (i.e. no suitable *res*? record exists), then this transaction is just not restartet.

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).

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

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:* Alice and Bob both insert a new task, for which the other person is charged, respectively. When doing so, they subtract from the actual cost of the task the amount of money they owe the other person. The company has strict conditions for inserting new tasks: tasks are not allowed to be inserted in parallel, but it must be possible to identify a clear order in which the tasks were added.

Schedule:

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

where `Tasks(a,b)` stands for the following string:

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

Lösung:
Isolation Level: `SERIALIZABLE`

(b) *Description:* Before customers can be charged for tasks, the corresponding task has to be approved by the responsible supervisor. The approvals are inserted in the database within the scope of transactions. These transactions should restrict the possibility of executing transactions in parallel as little as possible. In addition, it is no problem if finished parallel updates on the database become visible during the course of these transactions.

Schedule:

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

where `¬approved` and `¬paid` stand for the conditions:
`task_id NOT IN (SELECT task_id FROM approved)` resp.
`task_id NOT IN (SELECT task_id FROM paid)`

Lösung:
Isolation Level: `READ COMMITTED`

(c) *Description:* Accounting wants to issue invoices to customers with approved, but yet unpaid tasks. For this operation (i.e., billing), it is important to work with consistent data, i.e. the data must not change due to parallel writes while the invoices are created.

Schedule:

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

where `open` stands for the condition
`tasks NATURAL JOIN approved WHERE task_id NOT IN (SELCT task_id FROM paid)`

Lösung:
Isolation Level: `REPEATABLE READ`

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.