CS-355 Database Systems Term: Fall 2023 HW 04 Solution:

Question 1: [5 points]

(a) In the context of transaction processing we have studied three different problems stated as follows. State an example of each from the University database in your textbook.

i. [1 point] the lost update problem,

ii. [1 point] the temporary read problem,

iii. [1 point] the incorrect summary problem.

(b) Some of the other types of concurrency problems are the:

i. [1 point] phantom read problem, and

ii. [1 point] non-repeatable read problem.

Explain briefly what is meant by these and construct an example of each from the University Database in your textbook.

Solution:

(a)

(i) The lost update problem happens when a transaction updates the value of a data-item while another transaction processes an earlier value of the same data-item. In this case, changes made by one of the transactions is lost, since both transactions may not write simultaneously.

In case of the university database: consider two transactions T1 and T2.

T1: reads and updates the total credits of a student by adding 3 credits he/she has taken.

T2: reads and updates the total credits of a student by adding 4 credits he/she has taken.

However, the sequence of operations is as follows:

Transaction T1	Transaction T2
read(tot_cred)	
	read(tot_cred)
credits = credits + 3	
	credits = credits + 4
write(tot_cred)	
	write(tot_cred)

For transactions T1 and T2, the update(s) made by T1 are lost.

(ii) Let's consider the same example of the two transactions above but in a slightly different order:

Transaction T1	Transaction T2
read(tot_cred)	
credits = credits + 3	
write(tot_cred)	
	read(tot_cred)
	credits = credits + 4
	write(tot_cred)
read(…)	

Transaction T1 has failed and must be rolled back. However, transaction T2 has made the changes to the database (i.e. the credits of the student), and now a compensating transaction must be undertaken in order to remove the effect of the temporary update by T2.

(iii) The Incorrect Summary Problem:

Suppose transaction T1 is updating the credits of each course by removing one credit from each Biology course, (i.e., courses with 'BIO' in their course code) and adding one credit to each Computer Science course(i.e., courses with 'CS' their course code), while transaction T2 is calculating the average of all credits of all courses.

Transaction T1	Transaction T2
read(biocourse1credits)	
biocourse1credits = biocourse1credits - 1	
write(biocourse1credits)	
…	sum = 0
	read(biocourse1credits)
	sum = sum + biocourse1credits
	read(cscourse1credits)
	sum = sum + cscourse1cre
read(cscourse1credits)	….
cscourse1credits += 1	
Write(cscourse1credits)	

At the end the total (sum) will be incorrect and therefore the average credits will also be incorrect.

(b)

(i) The Phantom Read Problem: The Phantom Read Problem occurs when a transaction executes the same query twice, but gets different results since another transaction may have added or removed rows. As an example, a transaction that relies on 'count' of all students will get different results if a student has been added or removed by another transaction in between the two queries.

(ii) The Unrepeatable Read Problem. In this case, a transaction T reads the same item twice and the item is changed by another transaction T′ between the two reads. An example can be reading the salary of an instructor twice, which has been changed (updated) by another transaction in between the two reads.

Question 2: [5 points]

Consider the following schedules: S1, S2, S3 and S4.

Note: rN(X) represents read(X) operation in Transaction N, while wN(X) represents write(X) operation in transaction N.

- S1: r1(X), r3(X), w1(X), r2(X), w3(X)
- S2: r1(X), r3(X), w3(X), w1(X), r2(X)
- S3: r3(X), r2(X), w3(X), r1(X), w1(X)
- S4: r3(X), r2(X), r1(X), w3(X), w1(X)

For clarity S1 is:

T1	T2	T3
read(X) [corresponds to r1(X)]		
		read(X) [corresponds to r3(X)]
write(X) [corresponds to w1(X)]		
	read(X) [corresponds to r2(X)]	
		write(X) [corresponds to w3(X)]

(a) Use swapping of operations to determine which of the above schedules is conflict serialisable?

(b) If a schedule is conflict serialisable, give its equivalent serial schedule. If a schedule is not conflict serialisable, provide a reason.

Solution:

(a) Only S3 is conflict serializable, since using swapping it can be written as:
S3: r2(X) [Transaction T2], r3(X), w3(X) [Transaction T3], r1(X), w1(X) [Transaction T1].

(b) For S3, the equivalent serial schedule is T2, T3, T1. For all others there are conflicts between read/write operations of various transactions that prevent serializability.

Question 3: [5 points]

Consider three transactions T1, T2 and T3 with each of their operations and the schedules S1 and S2 below.

(a) Draw the serialisability (precedence) graphs for S1 and S2.

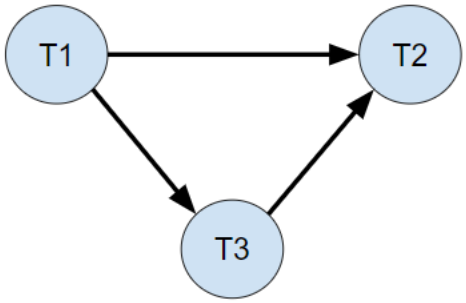
(b) State whether each schedule is serialisable or not.

(c) If a schedule is serialisable, write down the equivalent serial schedule.

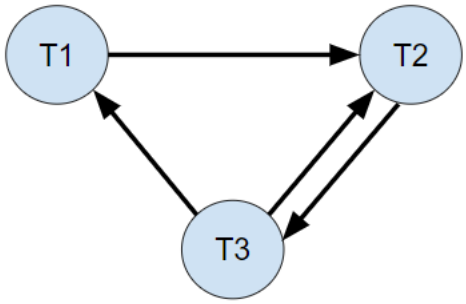
- S1: r1(X), r2(Z), r1(Z), r3(Y), w3(Y), w1(X), w3(X), r2(Y), w2(Z), w2(Y)
- S2: r1(X), r2(Z), r3(X), r1(Z), r2(Y), r3(Y), w1(X), w2(Z), w3(Y), w2(X).

Note: The conventions for the operation for each transaction are similar to the previous question.

(a) For S1:



For S2:



(b) S1 is serializable, S2 is not.

(c) The equivalent serial schedule for S1 is: T1, T3, T2.

Question 4: [5 points]

Write a Python function named `check_conflict_serializability` to determine whether a given schedule of transactions is conflict serialisable. The function should return a tuple indicating the conflict serialisability status and, if applicable, the order of transactions. You have already been a python file which includes skeleton code.

```
from graphlib import TopologicalSorter, CycleError

##########################################################################
### LIMITATIONS OF THE PROGRAM
### (a) This program uses python's graphlib for computing the topological order
sort
###     In cases where the topological sort exists, it outputs only one order
while several orders might be correct
### (b) Due to the way input is processed, it works only with 1 digit for the
transaction and one variable for the data item
###     This means it will work for upto 9 transactions T0-T9 and 26 data-items
A-Z
###     This however can be changed quite easily with some minor modifications to
the code
### (c) Any other limitations within graphlib apply here as well
##########################################################################
def process_schedule_list(schedule_list):
    precedence_graph = dict()

    for outer_index in range(len(schedule_list)):
        schedule_list_outer_item = schedule_list[outer_index]
        outer_operation = schedule_list_outer_item[0]
        outer_transaction_number = schedule_list_outer_item[1]
        outer_data_item = schedule_list_outer_item[3]

        for inner_index in range(outer_index + 1, len(schedule_list)):
            schedule_list_inner_item = schedule_list[inner_index]
            inner_operation = schedule_list_inner_item[0]
            inner_transaction_number = schedule_list_inner_item[1]
            inner_data_item = schedule_list_inner_item[3]
            ##print(schedule_list_outer_item + " " + schedule_list_inner_item)
            if ((outer_operation == "w" or inner_operation == "w") and
                (outer_transaction_number != inner_transaction_number) and
                (outer_data_item == inner_data_item)):
                ##print("Dependency between T" + outer_transaction_number + " and
T" + inner_transaction_number)
                source_vertex = "T" + outer_transaction_number
                dest_vertex = "T" + inner_transaction_number
                if dest_vertex in precedence_graph:
```

```
                    precedence_graph[dest_vertex].add(source_vertex)
                else:
                    precedence_graph[dest_vertex] = {source_vertex}

    ## print(precedence_graph)
    return topological_order_sort(precedence_graph)

def topological_order_sort(precedence_graph):
    ordered_vertices_list = []
    serializable = False
    try:
        ts = TopologicalSorter(precedence_graph)
        ordered_vertices_list.extend(list(ts.static_order()))
        print("Top Sort exists and order is " + str(ordered_vertices_list))
        serializable = True
    except CycleError:
        print("Topological Order Sort Does not exist")

    return (serializable, ordered_vertices_list)

def check_conflict_serializability(schedule):
    return process_schedule_list(schedule)
    """
    Check if a given schedule is conflict serializable.

    Args:
    - schedule (list): A list of transactions in the specified format.

    Returns:
    - tuple: A tuple (conflict_serializable, order_of_transactions),
             where conflict_serializable is a boolean indicating whether the
schedule is conflict serializable,
             and order_of_transactions is a list representing the order of
transactions if conflict_serializable is True.

    Example:
    >>> check_conflict_serializability(["r1(X)", "r2(X)", "w2(X)", "r3(X)",
"w1(Y)", "w3(X)"])
    (True, ['T1', 'T2', 'T3'])

    >>> check_conflict_serializability(["r1(X)", "w2(X)", "w1(X)"])
    (False, [])
    """
    # Your implementation for checking conflict serializability and determining
the order of transactions goes here
```

```
    pass

# Test Cases
# Note: We are only using test cases for the 'True' case where exactly one
ordering of vertices is possible
schedule_1 = ["r3(X)", "r2(X)", "w3(X)", "r1(X)", "w1(X)"] ## True, T2, T3, T1
schedule_2 = ["r1(X)", "w2(X)", "w1(X)"] ## False
schedule_3 = ["r1(X)", "r3(X)", "w1(X)", "r2(X)", "w3(X)"] ## False
schedule_4 = ["r1(X)", "r2(Z)", "r1(Z)", "r3(Y)", "w3(Y)", "w1(X)", "w3(X)",
"r2(Y)", "w2(Z)", "w2(Y)"] ## True, T1, T3, T2
schedule_5 = ["r1(X)", "r2(Z)", "r3(X)", "r1(Z)", "r2(Y)", "r3(Y)", "w1(X)",
"w2(Z)", "w3(Y)", "w2(X)"] ## False


# Process the schedules
print("For schedule 1" )
result_1 = check_conflict_serializability(schedule_1)
print("For schedule 2")
result_2 = check_conflict_serializability(schedule_2)
print("For schedule 3")
result_3 = check_conflict_serializability(schedule_3)
print("For schedule 4")
result_4 = check_conflict_serializability(schedule_4)
print("For schedule 5")
result_5 = check_conflict_serializability(schedule_5)


# Display the results
print(f"Conflict Serializable: {result_1[0]}\nOrder of Transactions: {result_1[1]
if result_1[0] else 'N/A'}")
print(f"\nConflict Serializable: {result_2[0]}\nOrder of Transactions:
{result_2[1] if result_2[0] else 'N/A'}")
print(f"\nConflict Serializable: {result_3[0]}\nOrder of Transactions:
{result_3[1] if result_3[0] else 'N/A'}")
print(f"\nConflict Serializable: {result_4[0]}\nOrder of Transactions:
{result_4[1] if result_4[0] else 'N/A'}")
print(f"\nConflict Serializable: {result_5[0]}\nOrder of Transactions:
{result_5[1] if result_5[0] else 'N/A'}")
```

Question 5: [5 points]

Consider the following three transactions:

T1	T2	T3
read(X)	Read(Y)	Read(X)
Read(Y)	Y=Y+5	X = X-5
Y=X+5	Write(Y)	Write(X)
Write(Y)	commit	Read(A)
Read(A)		A = A+5
A=A+5		Write(A)
Write(A)		commit
commit		

Consider the following concurrent schedule:

Line No	T1	T2	T3
	read(X)		
	Read(Y)		
	Y=X+5		
	Write(Y)		
			Read(X)
			X = X-5
			Write(X)
		Read(Y)	
	Read(A)		
	A=A+5		
	Write(A)		
			Read(A)
			A = A+5
	commit		
		Y=Y+5	
			Write(A)
			commit
		Write(Y)	
		commit	

(a) Is this schedule recoverable?

(b) Develop a cascadeless concurrent schedule for T1, T2, and T3.

Solution:

a) Yes, the given transaction schedule is recoverable

b) A possible cascadeless concurrent schedule for T1, T2, and T3 is given below:

T1	T2	T3
read(X)		
Read(Y)		
Y=X+5		
Write(Y)		
		Read(X)
		X = X-5
		Write(X)
Read(A)		
A=A+5		
Write(A)		
commit		
	Read(Y)	
		Read(A)
		A = A+5
	Y=Y+5	
		Write(A)
		commit
	Write(Y)	
	commit	