

DATABASE SYSTEMS

CS - 355/CE - 373

Instructor: Maria N. Samad

October 23rd, 2024

REVIEW – TRANSACTIONS

- A transaction (with its steps) must appear to the user as a single, indivisible unit. A
 transaction executes in its entirety or not at all.
- Several problems can occur due to concurrent execution, leading to the concepts of transaction processing.
- A transaction must have ACID properties: Atomicity, Consistency, Isolation, Durability.
- A transaction must be in one of the following states: Active, Partially Committed, Failed, Aborted or Committed.

TRANSACTION SCHEDULES

- When transactions are executing concurrently in an interleaved fashion, then the
 order of execution of operations from all the various transactions is known as a
 schedule (or history).
- More formally, a **schedule** (or **history**) S of n transactions $T_1, T_2, ..., T_n$ is an ordering of the operations of the transactions.
- Operations from different transactions can be interleaved in the schedule S.
- However, for each transaction T_i that participates in the schedule S, the operations of T_i in S must appear in the same order in which they occur in T_i .

SERIAL SCHEDULE

A serial schedule
 consists of a sequence
 of instructions from
 various transactions,
 where the instructions
 belonging to one single
 transaction appear
 together in that
 schedule.

T_1	T_2	T_1	T_2
read(A) A := A - 50 write(A) read(B) B := B + 50 write(B) commit	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B) $B := B + temp$ write(B) commit	read(A) A := A - 50 write(A) read(B) B := B + 50 write(B) commit	read(A) temp := A * 0.1 A := A - temp write(A) read(B) B := B + temp write(B) commit

CONCURRENT SCHEDULE – CONSISTENT STATE

T_1	T_2
read(A) $A := A - 50$ $write(A)$	
	read(A)
	temp := A * 0.1
	A := A - temp write(A)
read(B)	Wille(A)
B := B + 50	
write(B)	
commit	
	read(B)
	B := B + temp
	write(B)
	commit

Figure 17.4 Schedule 3—a concurrent schedule equivalent to schedule 1.

CONCURRENT SCHEDULE – INCONSISTENT STATE

T_1	T_2
read(A)	
A := A - 50	
	read(A)
	temp := A * 0.1
	A := A - temp
	write(A)
	read(B)
write(A)	
read(B)	
B := B + 50	
write(B)	
commit	D D
	B := B + temp
	write(B)
	commit

Figure 17.5 Schedule 4—a concurrent schedule resulting in an inconsistent state.

SERIALIZABLE SCHEDULE

- A serial schedule (as shown in one of the previous slides) is always consistent, i.e. it maintains the consistent state of the database.
- However, the same cannot be guaranteed for a concurrent schedule.
- If a concurrent schedule can be shown to have the same effect as a serial schedule, (in other words it is shown to be equivalent to a serial schedule), then it can ensure the consistency of the database.
- Such a schedule is called a serializable schedule.

T_1	T_2
read(A)	
A := A - 50	
write(A)	
	read(A)
	temp := A * 0.1
	A := A - temp
	write(A)
read(B)	
B := B + 50	
write(B)	
commit	
	read(B)
	B := B + temp
	write(B)
	commit

T_1	T_2
read(A) $A := A - 50$ write(A) read(B) $B := B + 50$ write(B) commit	read(A) temp := A * 0.1 A := A - temp write(A) read(B) B := B + temp write(B) commit

Figure 17.4 Schedule 3—a concurrent schedule equivalent to schedule 1.

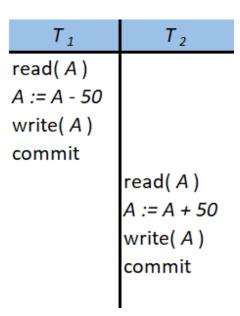
Figure 17.2 Schedule 1—a serial schedule in which T_1 is followed by T_2 .

SERIALIZABLE SCHEDULE

- In the previous slide, it may seem we have simply combined the operations of each transactions and executed them one after the other. However, that may not be possible each time
- This happened because there was no dependency of data from one transaction to another.
- When there are such dependencies, we can't simply merge the operations and execute them in serial manner, as this may lead to inconsistent and/or incorrect output

T_1	T 2
read(A)	
A := A - 50	
	read(<i>A</i>)
	A := A + 50
write(A)	
commit	
	write(A)
	commit





SERIALIZABLE SCHEDULE

- So how do we know if a schedule is serializable or not?
- We achieve that by resolving conflicts between operations. If there are no conflicts, we can swap the operations with each other to get a resultant serial schedule
- We consider only read(X) and write(X) operations. The underlying assumption is that these two operations are the most significant operations from a scheduling perspective.
- All the other operations that happen between these significant operations will automatically be handled along with the reads and writes

- Lets consider a schedule S in which there are two consecutive instructions I and J of transactions T_i and T_j respectively.
- If I and J refer to different data items, then we can swap I and J without affecting the results of instructions in the schedule.
- However, if *I* and *J* refer to the same data item *Q*, then the order of the two steps may matter.

T1	T2
read(X)	
	read(Y)

Initial (Above), Swapped (Below)

T1	T2
	read(Y)
read(X)	

• Since we are dealing with only *read* and *write* instructions, there are four cases that we need to consider:

• [CASE 1] I = read(Q), J = read(Q):

• The order of I and J does not matter, since the same value of Q is read by T_i and T_i , regardless of the order.

T1	T2
read(Q)	
	read(Q)

Initial (Left), Swapped (Right) Equivalent

T1	T2
	read(Q)
read(Q)	

• [CASE 2] I = read(Q), J = write(Q):

- If I comes before J, then T_i does not read the value of Q that is written by T_j in instruction J.
- If J comes before I, then T_i reads the value of Q that is written by T_j . Thus, the order of I and J matters.

T1	T2
read(Q)	
	write(Q)

Initial (Left), Swapped (Right) Not Equivalent

T1	Т2			
	write(Q)			
read(Q)				

- [CASE 3] *I* = write(*Q*), *J* = read(*Q*):
 - The order of I and J matters for reasons similar to those of the previous case.

T1	Т2
write(Q)	
	read(Q)

Initial (Left), Swapped (Right) Not Equivalent

T1	Т2				
	read(Q)				
write(Q)					

• [CASE 4] *I* = write(*Q*), *J* = write(*Q*):

- Since both instructions are write operations, the order of these instructions does not affect either T_i or T_i .
- However, the value obtained by the next read(Q) instruction of S is affected, since the result of only the latter of the two write instructions is preserved in the database.
- If there is no other write(Q) instruction after I and J in S, then the order of I and J
 directly affects the final value of Q in the database state that results from
 schedule S.

T1	T2
write(Q)	
	write(Q)

Initial (Left),
Swapped (Right)
Equivalent
ONLY when there is no read(Q) in between

T1	T2			
	write(Q)			
write(Q)				

SERIALIZABLE SCHEDULES – CONFLICTS

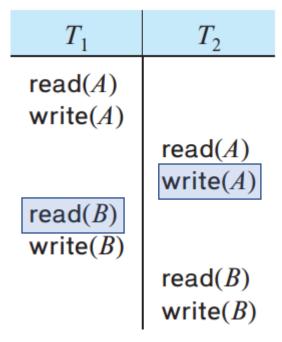
- Thus, only in the case where both I and J are read instructions does the relative order of their execution not matter.
- We say that I and J conflict if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

In:	Initial Schedule		Swapped Schedule		Conflict?		
	T1	T2			T1	T2	
	read(Q)					read(Q)	No
		read(Q)			read(Q)		
	T1	T2			T1	T2	
	read(Q)					write(Q)	Yes
		write(Q)			read(Q)		
	T1	T2			T1	T2	
	write(Q)					read(Q)	Yes
		read(Q)			write(Q)		
	T1	Т2			T1	T2	
	write(Q)					write(Q)	Yes
		write(Q)			write(Q)		

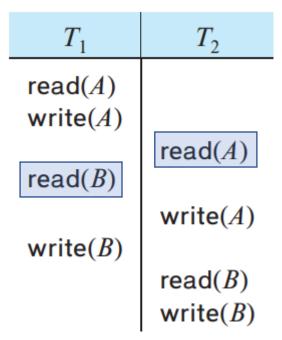
SERIALIZABLE SCHEDULES – SWAPPING

- Let I and J be consecutive instructions of a schedule S.
- If I and J are instructions of different transactions and I and J do not conflict, then we can swap the order of I and J to produce a new schedule S'.
- S is equivalent to S', since all instructions appear in the same order in both schedules except for I and J, whose order does not matter.

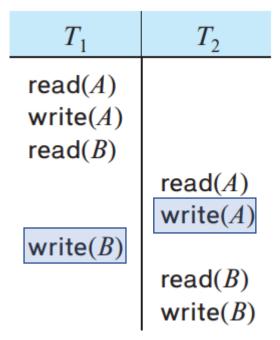
- Consider the following schedule \rightarrow
- We can swap the instructions
 read(B) and write(A) as they do not
 conflict.



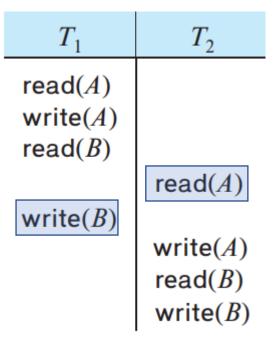
- The resulting schedule is shown \rightarrow
- We can again swap the instructions read(B) and read(A) as they do not conflict.



- The resulting schedule is shown \rightarrow
- We can now swap the instructions write(B) and write(A) as they do not conflict.



- The resulting schedule is shown →
- Finally, we can swap the instructions write(B) and read(A) as they do not conflict.



- The resulting schedule is shown \rightarrow
- Note that this schedule is a serial schedule.
- Since the original schedule has been shown to be equivalent to a serial schedule, we conclude that the original schedule maintains the consistency of the database.

T_1	T_2
read(A) write(A) read(B) write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

• If a schedule S can be transformed into a schedule S' by a series of swaps of nonconflicting instructions, we say that S and S' are **conflict equivalent**.

T_1	T_2	E Q	T_1	T_2
read(A)		U	read(A)	
write(A)		V	write(A)	
	read(A)	A	read(B)	
	write(A)	L -	write(B)	
read(B)		E N		read(A)
write(B)		T		write(A)
	read(B)			read(B)
	write(B)	T 0		write(B)

- Example: Given three schedules as follows:
 - S1: r1(X), w1(X), r2(X), w2(X), r1(Y), w1(Y), r2(Y), w2(Y)
 - S2: r1(X), w1(X), r1(Y), r2(X), w2(X), w1(Y), r2(Y), w2(Y)
- Check if the two schedules are conflict equivalent to each other by using swapping techniques
- Solution: On board

- Example: Given two schedules as follows:
 - S1: r1(X), r2(Y), w2(Y), w1(X), w3(X)
 - S2: r1(X), w1(X), r2(Y), w3(X), w2(Y)
- Check if the schedules are conflict equivalent to each other by using swapping techniques
- Solution: On board

- Example: Given two schedules as follows:
 - S1: r1(X), r2(X), w1(X), w2(X), w3(X)
 - S2: r1(X), w1(X), r2(X), w2(X), w3(X)
- Check if the schedules are conflict equivalent to each other by using swapping techniques
- Solution: On board

Class Activity

- Class Activity Solution
 - Conflict Equivalent Schedules Solution