DBMS: CSI 221

Transactions

Mohammad Imam Hossain
Lecturer, dept. of CSE, UIU

Transactions

- A collection of several operations on the database appears to be a single unit from the point of view of the database user.

  For example, a transfer of funds from a checking account to a savings account is single operation from the customer's standpoint; within the database system, however it consists of several operations.

- Collections of operations that form a single logical unit of work are called transactions.

  A database system
  -- must ensure proper execution of transactions despite failures either the entire transaction executes, or none of it does.
  -- must manage concurrent execution of transactions that avoids inconsistency.

Properties of Transactions(ACID)

- Atomicity: Either all operations of the transaction are reflected properly in the database, or none are.
- Consistency: Execution of a transaction in isolation preserves the consistency(both data integrity consistency constraints and application-dependent consistency constraints) of the database.
- Isolation: Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions T_i and T_j, it appears to T_i that either T_j finished execution before T_i started or T_j started execution after T_i finished. Thus, each transaction is unaware of other transactions executing concurrently in the system
- Durability: After a transaction completes successfully the changes it has made to the database persist, even of there are system failures.

A Simple Transaction Model

- As SQL is a powerful and complex language, we begin with a simple database language that focuses on when the data are moved from disk to main memory and from main memory to disk.
- The data items in our simplified model contain a single data value and each data item is identified by a name (ex. A, B, X etc.)
- Transactions access data using two operations:

  read(X) : transfers the data item X from the database to a variable, also called X, in a buffer in main memory belonging to the transaction that executed the read operation.

  write(X) : transfers the value in the variable X in the main-memory buffer of the transaction that executed the write to the data item X in the database.

Transaction Example

Let, T_i be a transaction that transfers $50 from account A to account B. This transaction can be defined as:

1. read(A);
2. A := A − 50;
3. write(A);
4. read(B);
5. B := B + 50;
6. write(B);

Transaction Example (ACID)

Let, T_i be a transaction that transfers $50 from account A to account B. This transaction can be defined as:

1. read(A);
2. A := A – 50;
3. write(A);
4. read(B);
5. B := B + 50;
6. write(B);

A = Atomicity

- If the transaction fails after step 3 and before step 6, money will be lost leading to an inconsistent database state. Failure could be due to software or hardware.
- The system should ensure that updates of a partially executed transaction are not reflected in the database.
- The basic idea behind ensuring atomicity is this: The database system keeps track of the old values of any data on which a transaction performs a write. This information is written to a file called the *log*.

Transaction Example (ACID)

Let, T_i be a transaction that transfers $50 from account A to account B. This transaction can be defined as:

1. read(A);
2. A := A − 50;
3. write(A);
4. read(B);
5. B := B + 50;
6. write(B);

D = Durability

- Once the user has been notified that the transaction has completed the updates to the database by the transaction must persist even if there are software or hardware failures.
- We can guarantee durability by ensuring any of the following:

  i) The updates carried out by the transaction have been written to disk before the transaction completes.

  ii) Information about the updates carried out by the transaction and written to disk is sufficient to enable that database to reconstruct the updates when the database system is restarted after the failure.

Transaction Example (ACID)

Let, T_i be a transaction that transfers $50 from account A to account B. This transaction can be defined as:

1. read(A);
2. A := A − 50;
3. write(A);
4. read(B);
5. B := B + 50;
6. write(B);

C = Consistency

- In general consistency requirements are 2 kinds:
  i) Explicit integrity constraints such as primary keys and foreign keys
  ii) Implicit integrity constrains such as the sum of A and B is unchanged by the execution of the transaction T_i
- A transaction when starting to execute must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent.

Transaction Example (ACID)

I = Isolation

- if between steps 3 and 6 (of the fund transfer transaction) , another transaction **T2** is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

T1	T2
1. **read**(A)	
2. $A := A - 50$	
3. **write**(A)	
	read(A), read(B), print(A+B)
4. **read**(B)	
5. $B := B + 50$	
6. **write**(B	

- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other.

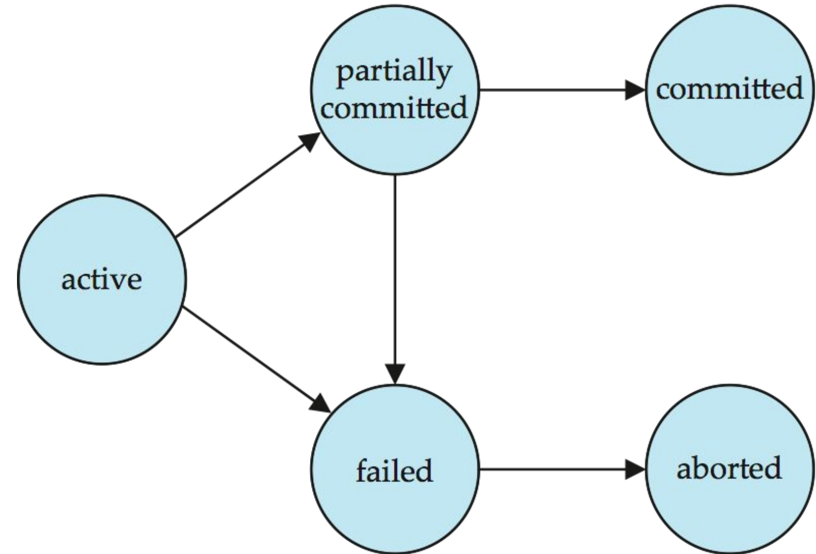
Transaction States

Active:

The initial state; the transaction stays in this state while it is executing.

Partially Committed:

After the final statement has been executed. At this point the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion.

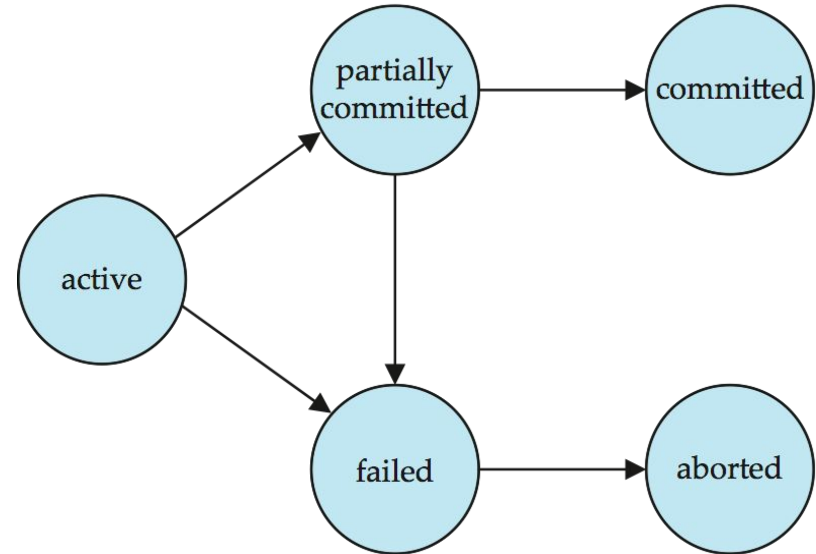


Transaction States

Failed:
After the discovery that normal execution can no longer proceed with its normal execution.

Aborted:
After the transaction has been rolled back, then it enters the aborted state.

Two options after it has been aborted:
- restart the transaction for hardware or software error that was not created through the internal logic of the transaction.
- Kill the transaction for logical error that can only be corrected by rewriting the application program.



Concurrent Execution

Transaction processing systems usually allow multiple transactions to run concurrently.

Advantages are:

- **Improved throughput and resource utilization**: The parallelism of the CPU and I/O system can therefore be exploited to run multiple transactions in parallel. While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read/write on behalf of 3rd transactions.

- **Reduced average response time**: short transactions need not wait behind long ones. If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them.

Schedules

A sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed

- A schedule for a set of transactions must *consist of all instructions* of those transactions
- Must *preserve the order* in which the instructions appear in each individual transaction.

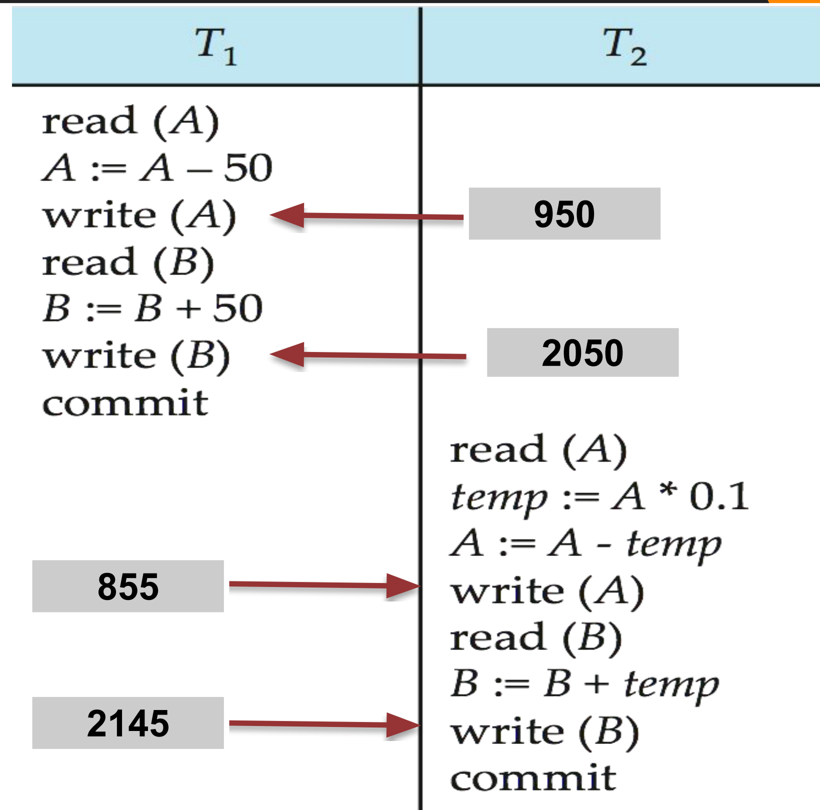
Serial Schedules

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

For a set of *n* transactions there exists *n!* different valid serial schedules but we can't predict how many schedules are possible in total (it depends on the OS and hardware).

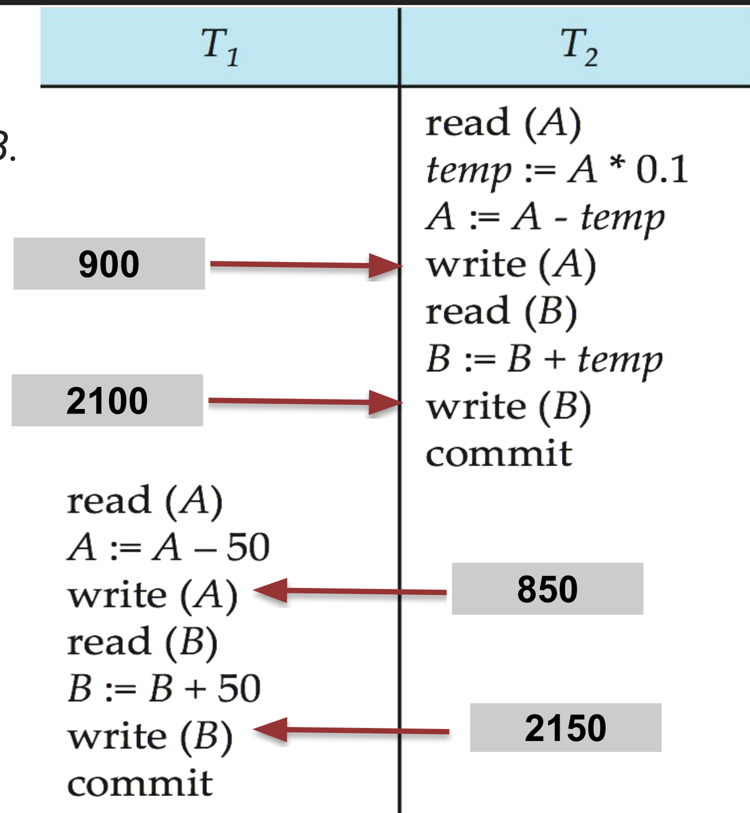
Schedule 1

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- Initial balance:
  A = 1000$ and
  B = 2000$
- An example of a **serial** schedule in which T_1 is followed by T_2 :



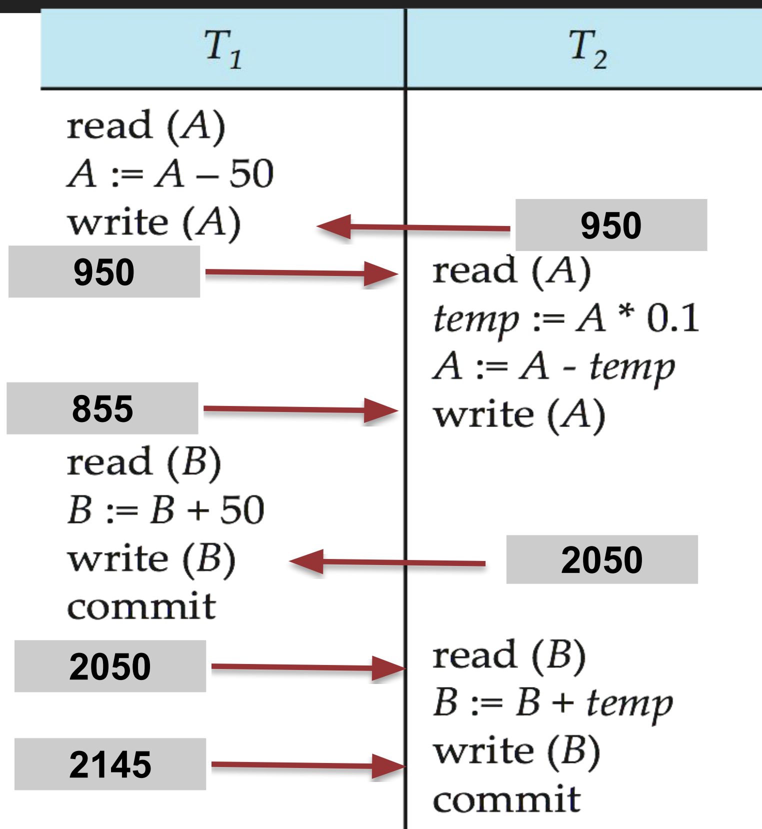
Schedule 2

- Let T_1 transfer $50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B*.
- Initial balance:
  A = 1000$ and
  B = 2000$
- An example of a **serial** schedule in which T_2 is followed by T_1 :



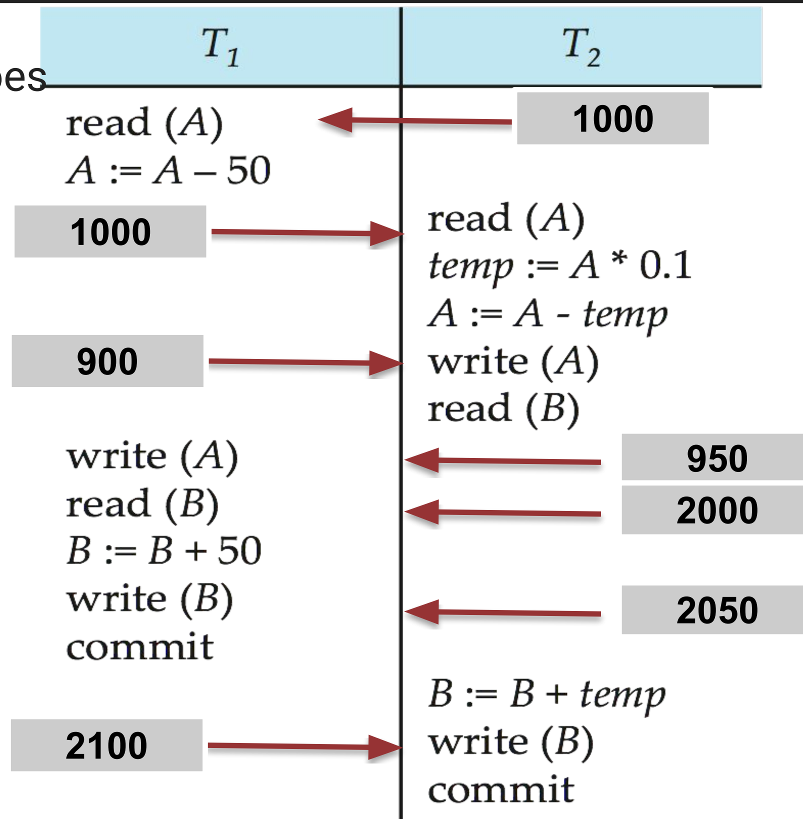
Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is **equivalent** to Schedule 1.
- In schedule 1, 2 and 3 the sum 'A+B' is preserved.



Schedule 4

- The following concurrent schedule does not preserve the sum of "*A* + *B*"



Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus, serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.

Different forms of schedule equivalence give rise to the notions of:

1. **conflict serializability**
2. **view serializability**

Simplified view of Transactions

- We ignore operations other than **read** and **write** instructions.
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

- Let I_i and I_j be two Instructions of transactions T_i and T_j respectively. Instructions I_i and I_j **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.
  1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j don't conflict.
  2. I_i = **read**(Q), I_j = **write**(Q). They conflict.
  3. I_i = **write**(Q), I_j = **read**(Q). They conflict
  4. I_i = **write**(Q), I_j = **write**(Q). They conflict
- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them.
  - If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

- If a schedule *S* can be transformed into a schedule *S´* by a series of swaps of non-conflicting instructions, we say that *S* and *S´* are **conflict equivalent**.
- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability Example 1

- Schedule 3 can be transformed into Schedule 6 -- a serial schedule where T_2 follows T_1, by a series of swaps of non-conflicting instructions. Therefore, Schedule 3 is conflict serializable.

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

Schedule 3

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

Schedule 6

Conflict Serializability Example 2

- Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule $< T_3, T_4 >$, or the serial schedule $< T_4, T_3 >$.

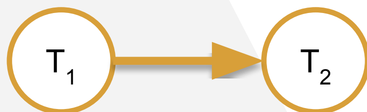
Precedence Graph

- Consider a schedule S. We construct a directed graph, called a precedence graph from S.
- This graph consists of a pair *G=(V, E)*, where V is a set of vertices and E is a set of edges.

  Here,
  V consists of all the transactions participating in the schedule.

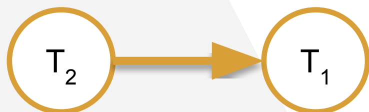
  E consists of all edges T_i□T_j for which one of 3 conditions holds:
  - T_i executes write(Q) before T_j executes read(Q)
  - T_i executes read(Q) before T_j executes write(Q)
  - T_i executes write(Q) before T_j executes write(Q)
- If an edge T_i □ T_j exists in the precedence graph, then in any serial schedule S' equivalent to S, T_i must appear before T_j.

Precedence Graph Example 1



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

Precedence Graph Example 2



T_1	T_2
	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	

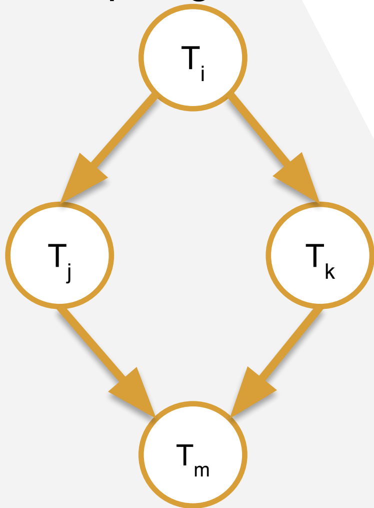
Precedence Graph Example 3

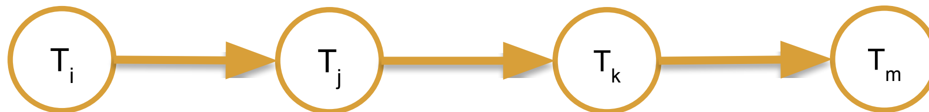
- If the precedence graph for S has a cycle, then the schedule S is not conflict serializable.
- If the graph contains no cycles, then the schedule S is conflict serializable.

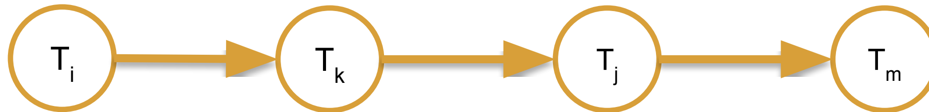
T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Serializability Order

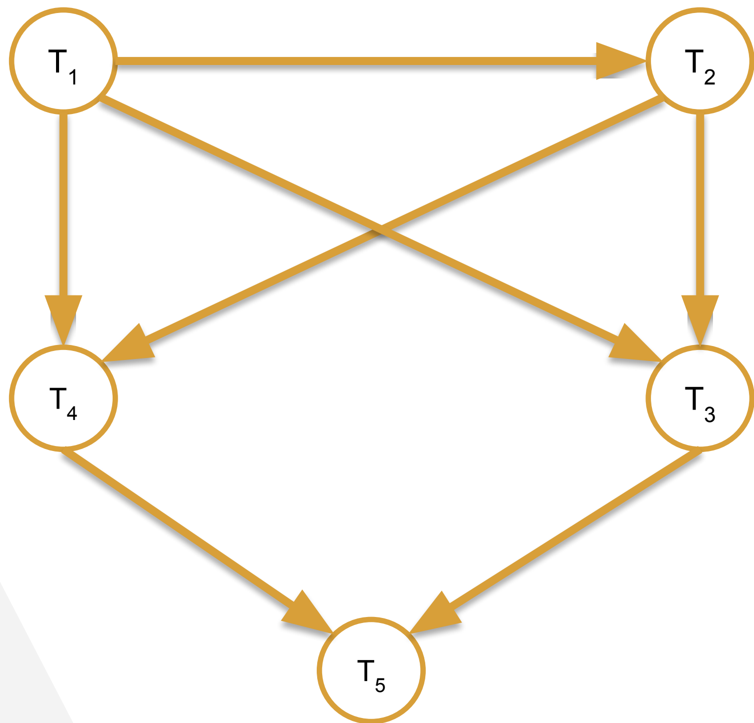
- A serializability order of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph. This process is called topological sorting.
- There are, in general several possible linear orders that can be obtained through a topological sort.







Conflict Serializability Checking 1

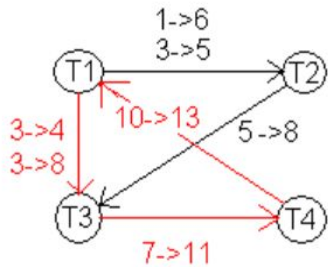


Conflict Serializability Checking 2

T1	T2	T3	T4
read(X) (1)			
	read(X) (2)		
write(Y) (3)			
		read(Y) (4)	
	read(Y) (5)		
	write(X) (6)		
		read(W) (7)	
		write(Y) (8)	
			read(W) (9)
			read(Z) (10)
			write(W) (11)
read(Z) (12)			
write(Z) (13)			

Solution

Here is the precedence graph:



As you can see, there are cycles in the graph: T1->T3->T4->T1 and T1->T2->T3->T4->T1. This means that the schedule is not conflict serializable.

References:

Database System Concepts by Korth, 6th edition, sections: 14.1 – 14.6

THANKS!

Any questions?

You can find me at imam@cse.uiu.ac.bd