Transactions

Transaction

- A transaction is an action, or a series of actions, carried out by a single user or an application program, which reads or updates (writes) the contents of a database.

- Any action that reads from and/or writes to a database may consist of
  - Simple SELECT statement to generate a list of table contents
  - A series of related UPDATE statements to change the values of attributes in various tables
  - A series of INSERT statements to add rows to one or more tables
  - A combination of SELECT, UPDATE, and INSERT statements

Transaction

- A *logical* unit of work that must be either entirely completed or aborted
- Successful transaction changes the database from one *consistent* state to another
  - One in which all data integrity constraints are satisfied
- Most real-world database transactions are formed by two or more database requests
  - The equivalent of a single SQL statement in an application program or transaction

  - Some examples of transactions are
    - Money transactions
    - Ticket booking
    - Online admission
    - Remote gaming
    - …

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g. transaction to transfer \$50 from account A to account B:
  1. **read**(A)
  2. $A := A - 50$
  3. **write**(A)
  4. **read**(B)
  5. $B := B + 50$
  6. **write**(B)
- Two main issues to deal with:
  - **Failures** of various kinds, such as hardware failures and system crashes
  - **Concurrent** execution of multiple transactions

Example of multiple Transactions

- Let T_1 transfers $50 from A to B, and T_2 transfers 10% of the balance from A to B.
- Here T_1 and T_2 are two transactions and T_1 is followed by 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

ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either **all** operations of the transaction are properly reflected in the database **or none are**.
- **Consistency.** Execution of a transaction in isolation **preserves the consistency** of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each **transaction** must be **unaware of** other **concurrently executing transactions**. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, 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.
- **Durability.** After a transaction completes successfully, the changes it has made to the database **persist**, even if there are system failures.

Transaction Management with SQL

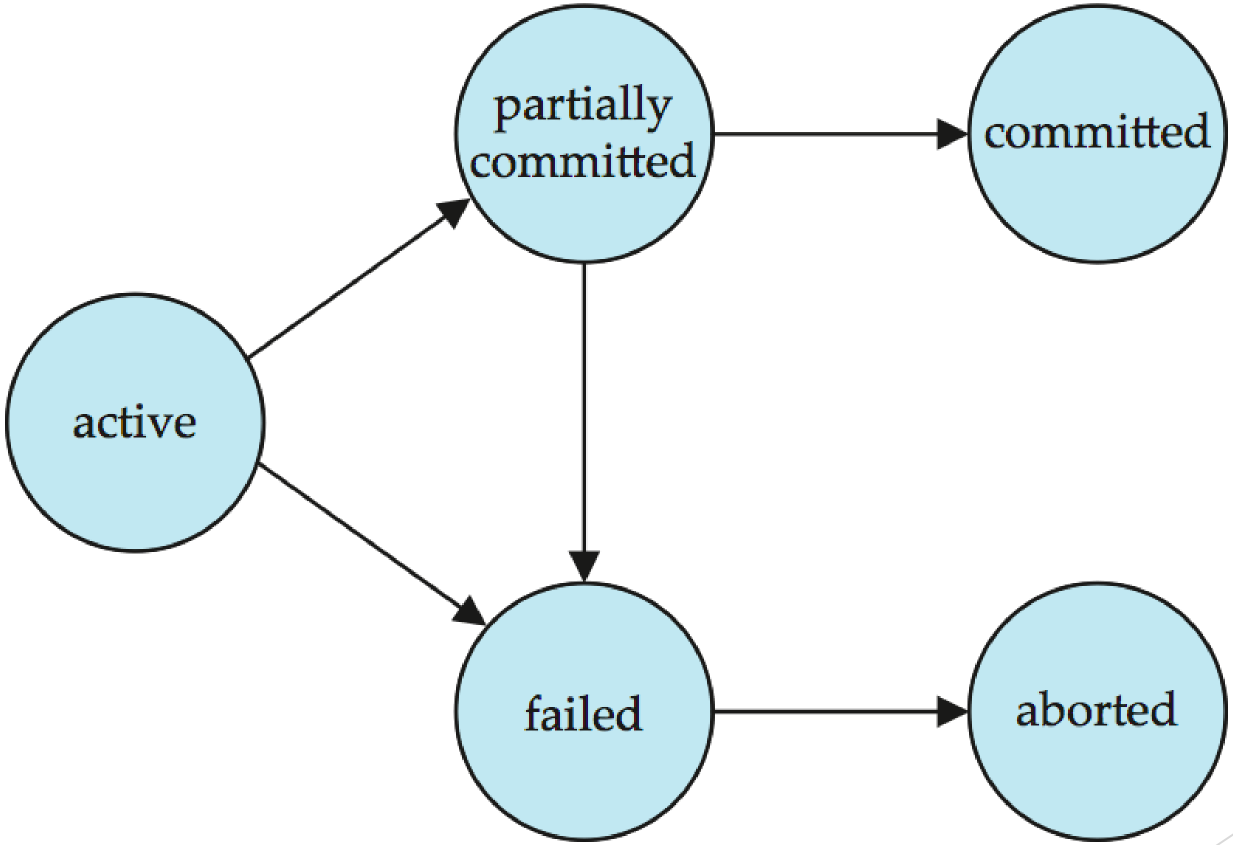
- ANSI has defined standards that govern SQL database transactions
- Transaction support is provided by two SQL statements: COMMIT and ROLLBACK
- ANSI standards require that, when a transaction sequence is initiated by a user or an application program, it must continue through all succeeding SQL statements until one of four events occurs

1. A COMMIT statement is reached- all changes are permanently recorded within the database
2. A ROLLBACK is reached – all changes are aborted and the database is restored to a previous consistent state
3. The end of the program is successfully reached – equivalent to a COMMIT
4. The program abnormally terminates and a rollback occurs

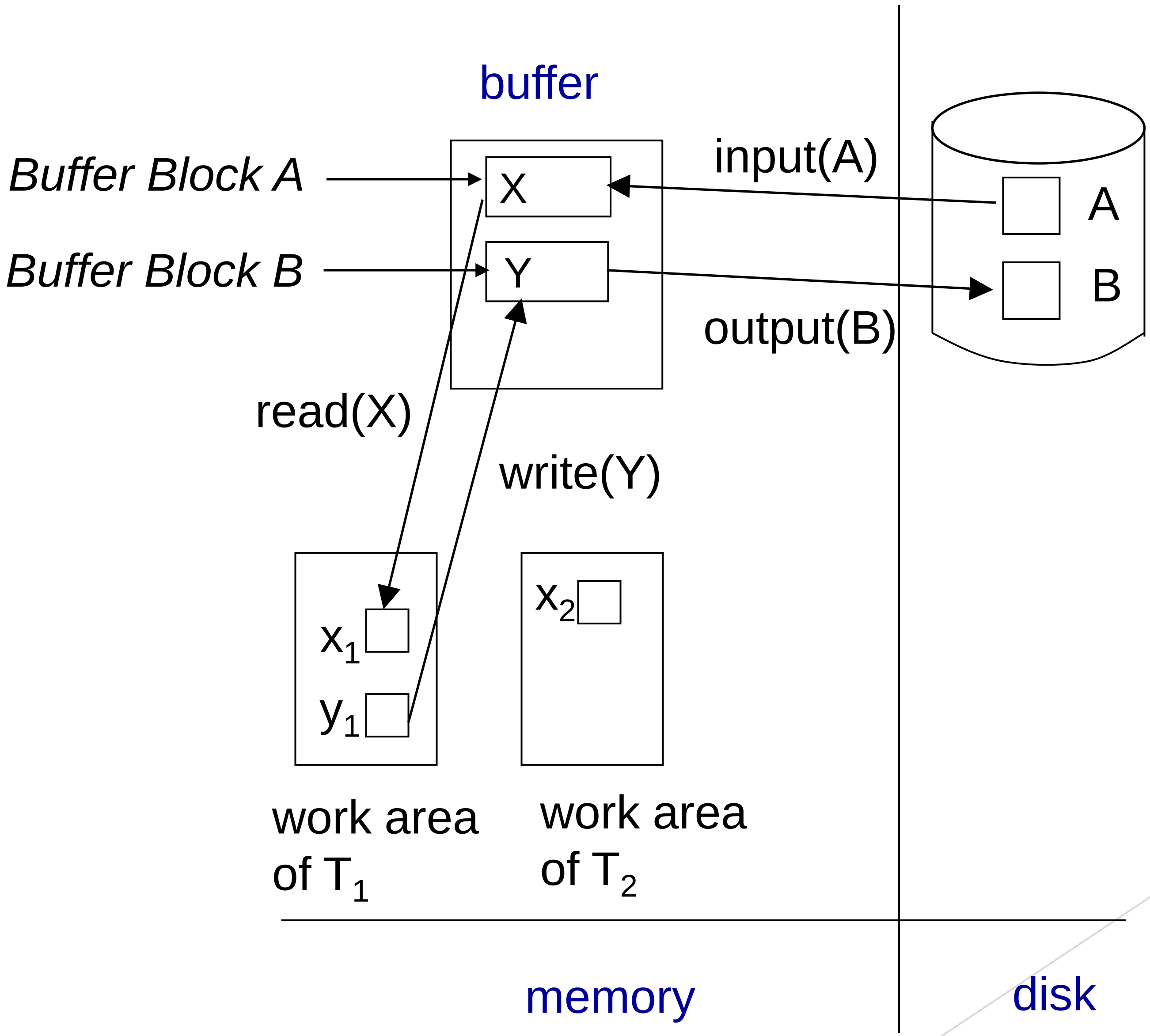
Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** -- after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction
    - can be done only if no internal logical error
  - kill the transaction
- **Committed** – after successful completion.

Transaction State (Cont.)



Example of Data Access



Example of Fund Transfer

- Transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B)*
- Atomicity requirement
  - **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**
- Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the **updates to the database by the transaction must persist** even if there are software or hardware failures.

Example of Fund Transfer (Cont.)

- Transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B)*
- **Consistency requirement** in above example:
  - the **sum of A and B is unchanged** by the execution of the transaction
- In general, consistency requirements include
  - **Explicitly** specified **integrity constraints** such as primary keys and foreign keys
  - **Implicit** integrity constraints
    - e.g. **sum of balances of all accounts**, minus sum of loan amounts **must equal value of cash-in-hand**
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer (Cont.)

- **Isolation requirement** — if between steps 3 and 6, 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.
- **However**, executing multiple transactions concurrently has **significant benefits**, like it will reduce time, increase modularity, give the advantage of remote access…

The Transaction Log

- Keeps track of all transactions that update the database. It contains:
  - A record for the beginning of transaction
  - For each transaction component (SQL statement)
    - Type of operation being performed (update, delete, insert)
    - Names of objects affected by the transaction (the name of the table)
    - “Before” and “after” values for updated fields
    - Pointers to previous and next transaction log entries for the same transaction
  - The ending (COMMIT) of the transaction
- Increases processing overhead but the ability to restore a corrupted database is worth the price

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedules

- **Schedule** – 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.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedule 1

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- A serial schedule in which T_1 is followed by T_2 :

A=500
A=450

B=600
B=650

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

Before transaction: A=500
B=600 Total=1100

After Transaction : A=405
B=695 Total=1100

A=450
temp=45
A=405

B=650
B=695

Schedule 2

- A serial schedule where T_2 is followed by T_1

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	

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.

Before transaction: A=500
B=600 Total=1100

After Transaction : A=405
B=695 Total=1100

	T_1	T_2	
A=500 A=450	read (A) $A := A - 50$ write (A)		
		read (A) $temp := A * 0.1$ $A := A - temp$ write (A)	A=450 Temp=45 A=405
B=600 B=650	read (B) $B := B + 50$ write (B) commit		
		read (B) $B := B + temp$ write (B) commit	B=650 B=695

In Schedules 1, 2 and 3, the sum **A + B is preserved**.

Schedule 4

Before transaction: A=600
B=500 Total=1100

After Transaction : A=550
B=560 Total=1110

*D=Disk *M=Memory

- The following concurrent schedule **does not preserve** the value of ***(A + B)***.

A(D)=600, B(D)=500	T_1	T_2	A(D)=600, B(D)=500
A(M)=600	read (A)		
A(M)=550	$A := A - 50$		
		read (A)	A(M)=600
		$temp := A * 0.1$	temp=60
		$A := A - temp$	A(M)=540
		write (A)	A(D)=540
		read (B)	B(M)=500
A(D)=550	write (A)		
B(M)=500	read (B)		
B(M)=550	$B := B + 50$		
B(D)=550	write (B)		
	commit		
		$B := B + temp$	B(M)=560
		write (B)	B(D)=560
		commit	

Conflicting Instructions

- Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.

  1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j **don't conflict**.
  2. l_i = **read**(Q), l_j = **write**(Q). They **conflict**.
  3. l_i = **write**(Q), l_j = **read**(Q). They **conflict**
  4. l_i = **write**(Q), l_j = **write**(Q). They **conflict**

- Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them.
  - If l_i and l_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.

Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, “**dirty reads**”):
- Take an Example of railway seat booking system. Don’t have enough tickets to book, but still showing wrong data. Remember, until commit, no changes is visible in database.

A=10; b=15	T_1	T_2	A=10; b=15
A=10	read (A)		
A=10-2=8	write (A)		
		read (A)	A=8
		write (A)	Write=8-8=0
		C	
B=15	read (B)		
B=15-2=13	write (B)		
	Abbort		

Anomalies with Interleaved Execution

- Unrepeatable Reads (RW Conflicts):

A=10; b=15

T_1	T_2
read (A)	
	read (A)
	write (A)
	C
read (A)	
write (A)	
C	

Different Data

A=10

A=2
A=2-2=0

A=10
A=10-8=2

Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

T_1	T_2
write (A)	
	write (A)
	write (B)
	C
write (B)	
C	

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**

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 (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by 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

But how to test conflict serializability?

Testing Conflict Serializability

- Construct precedence graph G for given schedule S
- S is conflict-serializable iff G is acyclic

Precedence Graph

- Precedence graph for schedule S:
  - Nodes: Transactions in S
  - Edges: Ti → Tj whenever
    - S: … ri (X) … wj (X) …
    - S: … wi (X) … rj (X) …
    - S: … wi(X) … wj (X) …

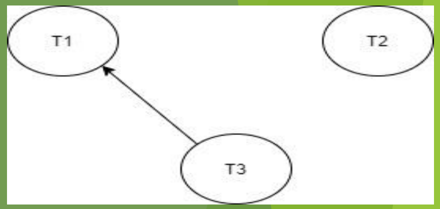
Note: not necessarily consecutive

Conflict Serializability (Cont.)

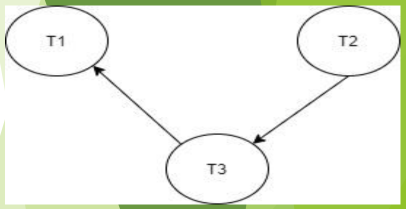
- Check conflict (RW, WR, WW) pairs in other transactions and draw edges

T1	T2	T3
R(x)		
		R(y)
		R(x)
	R(y)	
	R(z)	
		W(y)
	W(z)	
R(z)		
W(x)		
W(z)		

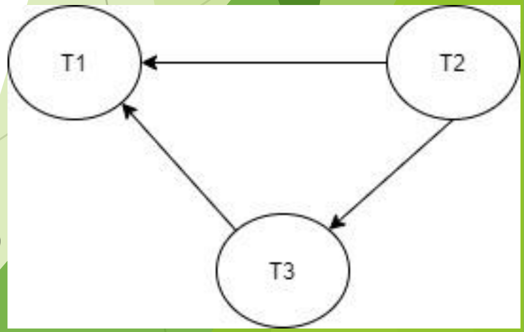
1. R(x) is okay, as no W(x) present in T2, T3
2. R(y) is okay, as no W(y) present in T1, T2
3. R(x) is not okay, as W(x) present in T1. So draw line from T3 to T1
4. R(y) is not okay as W(y) present in T3. So draw line from T2 to T3.
5. R(z) is not okay as W(z) present in T1. So draw line from T2 to T1
6. For W(y), we have to check W(y) and R(y) both. It is okay.
7. W(z) is not okay as R(z) and W(z) also present in T1. So draw a line from T2 to T1 (repeat of step 5. Don't mark again)
8. We don't have to check for the other Transactions as T2 and T3 is empty.



Step 3



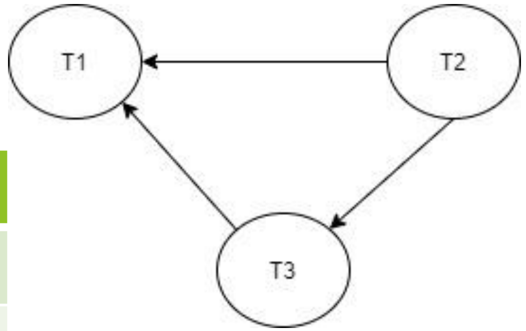
Step 4

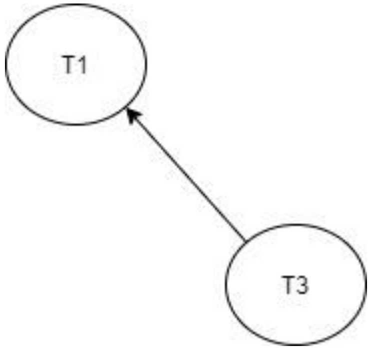


Step 5

Conflict Serializability (Cont.)

T1	T2	T3
R(x)		
		R(y)
		R(x)
	R(y)	
	R(z)	
		W(y)
	W(z)	
R(z)		
W(x)		
W(z)		





1. Now check is there any cycle or loop present or not in the precedence graph?
2. Here, we can't find any loop or cycle. So the transaction is conflict serializable. So we can make serializable transaction from it. It is also consistent transaction.
3. Now find the vertex whose indegree is zero and disconnect it from the precedence graph. (T2)
4. Now again check the other vertices to find the indegree 0. (T3).
5. So the transaction is serializable as T2->T3->T1

Conflict Serializability (Cont.)

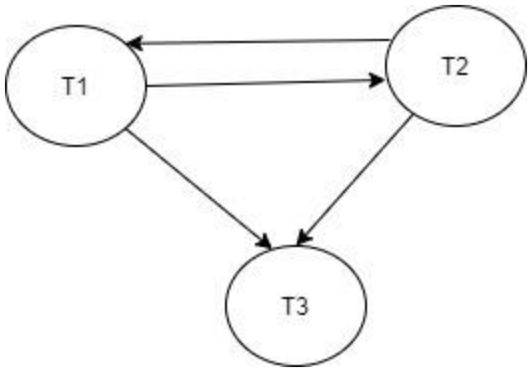
So the transaction is serializable as T2->T3->T1

T1	T2	T3
R(x)		
		R(y)
		R(x)
	R(y)	
	R(z)	
		W(y)
	W(z)	
R(z)		
W(x)		
W(z)		

T1	T2	T3
	R(y)	
	R(z)	
	W(z)	
		R(y)
		R(x)
		W(y)
R(x)		
R(z)		
W(x)		
W(z)		

View Serializability

T1	T2	T3
R(A) A=100		
	W(A) A-20; A=80	
W(A) A+10; A=90		
		W(A) A-50; A=40



1. From conflicting serializability method, we have drawn the precedence graph and we got a loop.
2. Now view serializability method will be used.
3. Conflicting write operations are modified and precedence graph is drawn again.

T1	T2	T3
R(A) A=100		
W(A) A+10;A=110		
	W(A) A-20; A=90	
		W(A) A-50; A=40

1. In this graph, we have not find any loop.
2. So the transactions are serializable.
3. This tables are not equal, but view equivalent.