Transactions

Overview

- *Transaction*: A sequence of database actions enclosed within special tags

T1: transfers $50 from A to B

```
begin T1
read(A)
A = A -50
write(A)
read(B)
B=B+50
write(B)
end T1
```

Overview (cont'd)

- Properties:
  - ***Atomicity***: Entire transaction or nothing
  - ***Consistency***: Transaction, executed completely, takes database from one consistent state to another
  - ***Isolation***: Concurrent transactions *appear* to run in isolation
  - ***Durability***: Effects of committed transactions are not lost
- Consistency: Transaction programmer needs to guarantee that
    - DBMS can do a few things, e.g., enforce constraints on the data
- Rest: DBMS guarantees

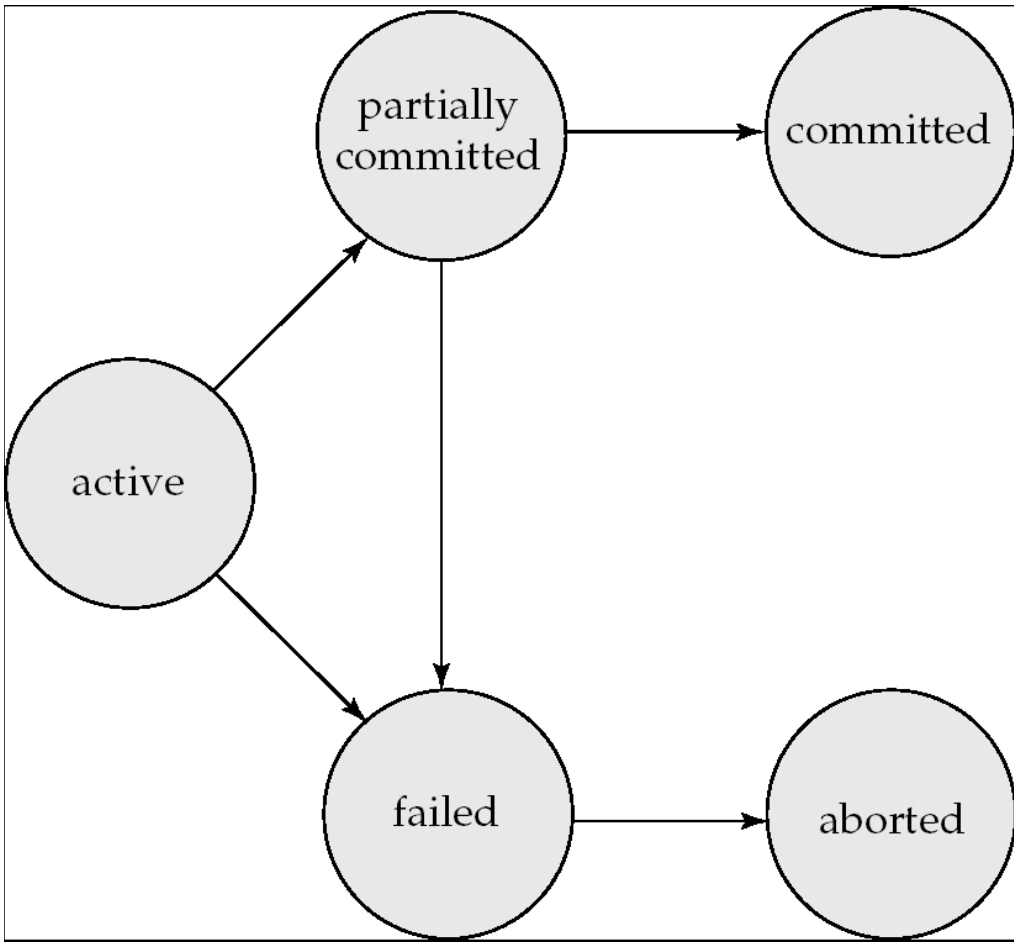
Assumptions and Goals

- Assumptions:
  - The system can crash at any time
  - Similarly, the power can go out at any point
    - Contents of the main memory won't survive a crash, or power outage
  - **Disks are durable. They might stop, but data is not lost.**
  - Disks only guarantee *atomic sector writes,* nothing more
  - Transactions are by themselves consistent
- Goals:
  - Guaranteed durability, atomicity
  - As much concurrency as possible, while not compromising isolation and/or consistency
    - Two transactions updating the same account balance… NO
    - Two transactions updating different account balances… YES

Next

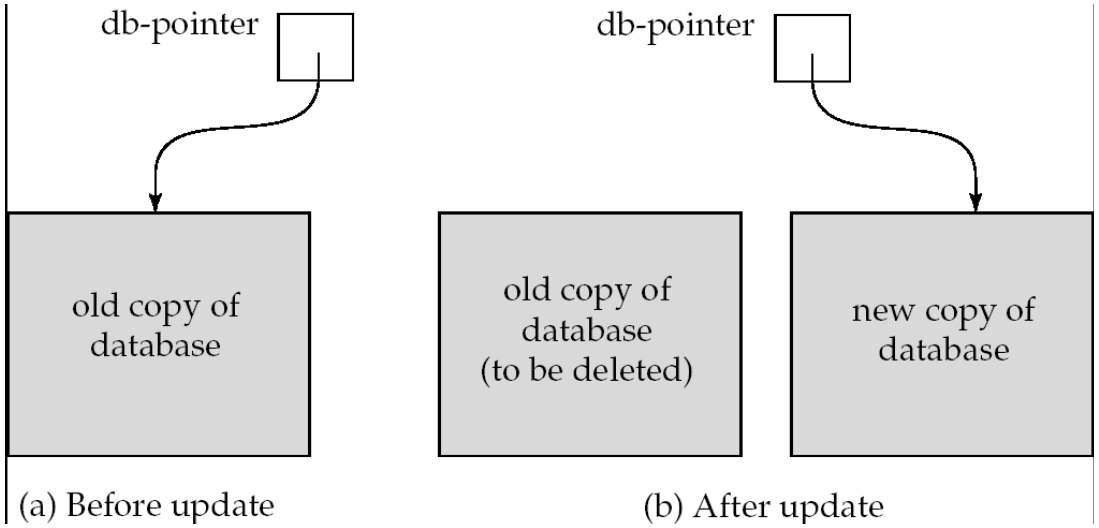
- States of a transaction
- A simple solution called *shadow copy*
  - Satisfies Atomicity, Durability, and Consistency, but no Concurrency
  - Very inefficient

Transaction states



Shadow Copy

- Make updates on a copy of the database.
- Switch pointers atomically after done.
  - Some *text editors* work this way



Shadow Copy

Atomicity:	• As long as the DB pointer switch is atomic. • Okay if DB pointer is in a single block
Concurrency:	• No.
Isolation:	• No concurrency, so isolation is guaranteed.
Durability:	• Assuming disk is durable (we will assume this for now).
Very inefficient:	• Databases tend to be very large. Making extra copies not feasible. Further, no concurrency.

Next

- Concurrency control schemes
  - A CC scheme is used to guarantee that concurrency does not lead to problems
  - For now, we will assume durability is not a problem
    - No crashes
    - Though transactions may still abort
- Schedules
- When is concurrency okay ?
  - Serial schedules
  - Serializability

A Schedule

Transactions:

T1: transfers $50 from A to B

T2: transfers 10% of A to B

Database constraint: A + B is constant (*checking+saving accts)*

T1	T2
read(A)	
A = A -50	
write(A)	
read(B)	
B=B+50	
write(B)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	read(B)
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	45
B	50	105

Each transaction obeys the constraint.

This schedule does too.

Schedules

- A *schedule* is simply a (possibly interleaved) execution sequence of transaction instructions
- *Serial Schedule:* A schedule in which transactions appear one after the other
  - No interleaving
- Serial schedules satisfy isolation and consistency
  - Since each transaction by itself does not introduce inconsistency

Example Schedule

- Another "serial" schedule:

T1	T2
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	read(B)
	B = B+ tmp
	write(B)
read(A)	
A = A -50	
write(A)	
read(B)	
B=B+50	
write(B)	

Effect:	Before	After
A	100	40
B	50	110

Consistent ?

Constraint is satisfied.

Another schedule

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B+ tmp
	write(B)

Is this schedule okay ?

Lets look at the final effect

Effect:	Before	After
A	100	45
B	50	105

Consistent.
So this schedule is okay too.

Another schedule

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B+ tmp
	write(B)

Is this schedule okay ?

Let's look at the final effect…

Effect:	Before	After
A	100	45
B	50	105

Further, the effect same as the serial schedule 1.

Called *serializable*

Example Schedules (Cont.)

A "bad" schedule

T1	T2
read(A)	
A = A -50	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	read(B)
write(A)	
read(B)	
B=B+50	
write(B)	
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	50
B	50	60

Not consistent

Serializability

- A schedule is called *serializable* if its final effect is the same as that of a *serial schedule*
- Serializability → schedule is fine and does not result in inconsistent database
  - Since serial schedules are fine
- Non-serializable schedules are likely to result in inconsistent databases
- We will ensure serializability
  - Typically relaxed in real high-throughput environments

Serializability

- Not possible to look at all *n!* serial schedules to check if the effect is the same
  - Instead we ensure serializability by allowing or not allowing certain schedules
- Conflict serializability
- View serializability
  - View serializability allows more schedules

Conflict Serializability

Two read/write instructions "conflict" if

- They are by different transactions
- They operate on the same data item
- **At least one** is a "write" instruction

Why do we care ?

- If two read/write instructions don't conflict, they can be "swapped" without any change in the final effect
- However, if they conflict they CAN'T be swapped without changing the final effect

Conflicting Instructions

- Instructions I_i and I_j of transactions T_i and T_j respectively, **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.

Equivalence by Swapping

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B+ tmp
	write(B)

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
read(B)	
	write(A)
B=B+50	
write(B)	
	read(B)
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	45
B	50	105

==

Effect:	Before	After
A	100	45
B	50	105

Equivalence by Swapping

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	45
B	50	105

! ==

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
	read(B)
write(B)	
	B = B+ tmp
	write(B)

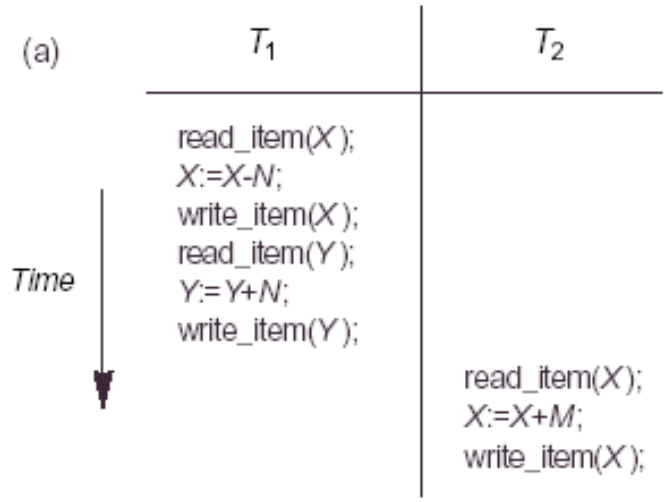
Effect:	Before	After
A	100	45
B	50	55

Conflict Serializability

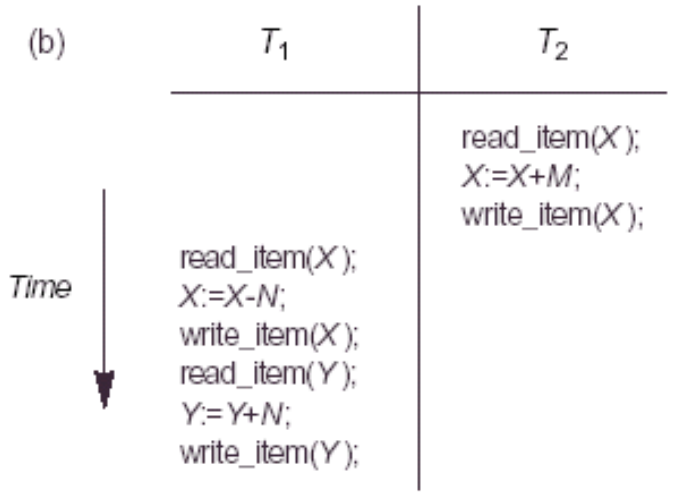
Conflict-equivalent schedules:

- If **S can be transformed into S'** through a series of swaps, S and S' are called *conflict-equivalent*
- *conflict-equivalent guarantees same final effect on the database*

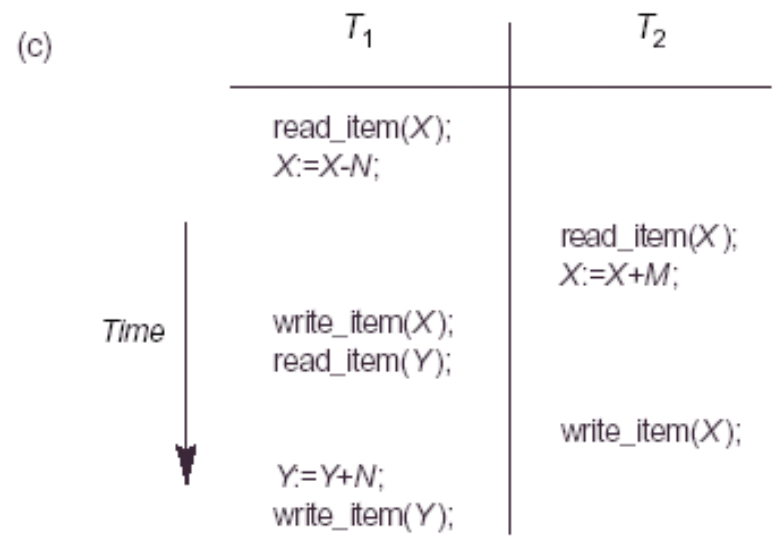
A schedule S is conflict-serializable if it is conflict-equivalent to a serial schedule



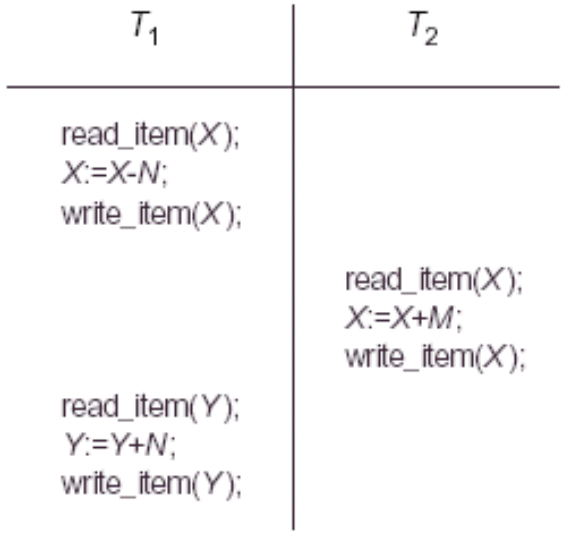
(serial) schedule A



(serial) schedule B



(non-serializable) schedule C



(serializable) schedule D

Equivalence by Swapping

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B+ tmp
	write(B)

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
read(B)	
B=B+50	
	write(A)
write(B)	
	read(B)
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	45
B	50	105

==

Effect:	Before	After
A	100	45
B	50	105

Equivalence by Swapping

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	45
B	50	105

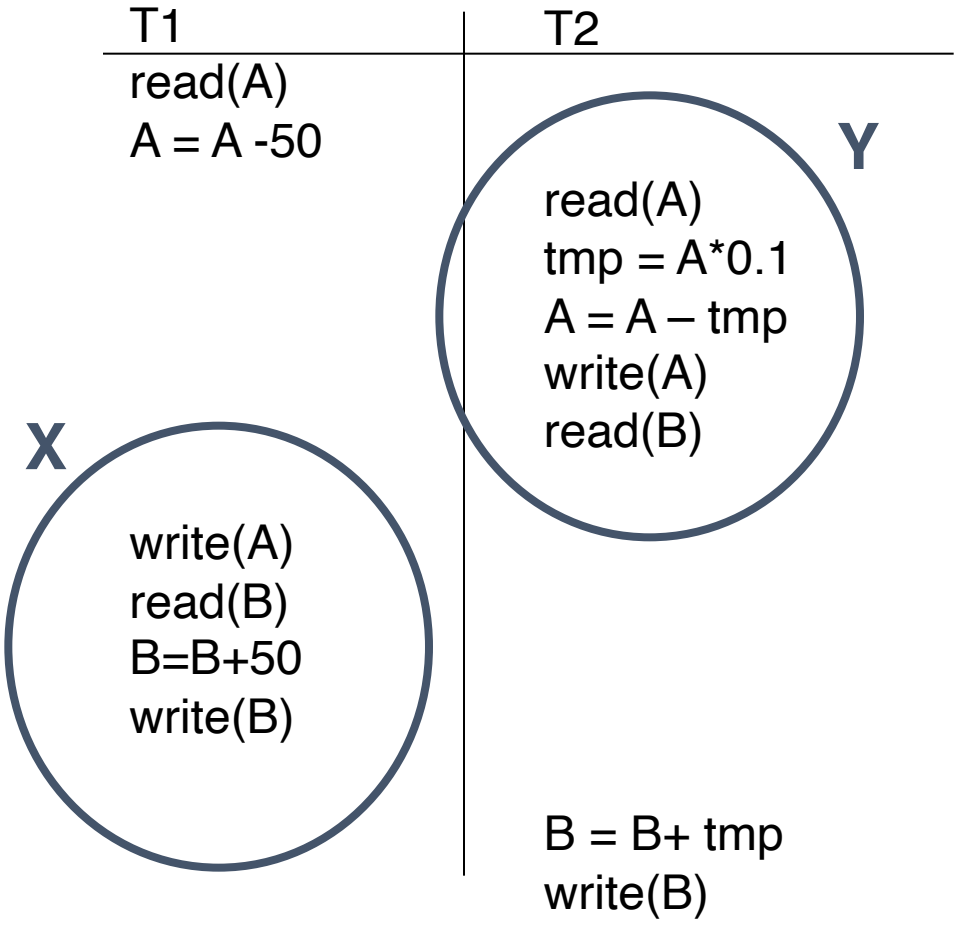
==

T1	T2
read(A)	
A = A -50	
write(A)	
read(B)	
B=B+50	
write(B)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	read(B)
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	45
B	50	105

Example Schedules (Cont.)

A “bad” schedule



Can’t move Y below X
read(B) and write(B) conflict

Other options don’t work either

Not Conflict Serializable

Serializability

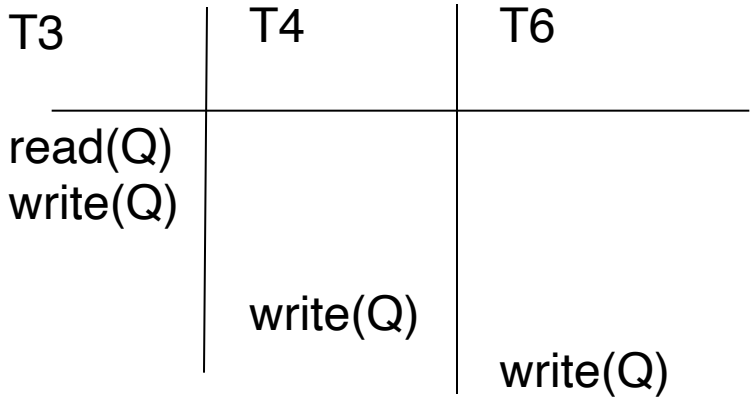
- In essence, following set of instructions is not conflict-serializable:

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

View-Serializability

- Similarly, the following schedule is not conflict-serializable

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)



- It is serializable in terms of the final results
  - Intuitively, this is because the *conflicting write instructions* don't matter
  - The final write is the only one that matters

Schedules S1 and S2 are view equivalent if:

If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2

If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2

If Ti writes final value of A in S1, then Ti also writes final value of A in S2

Other notions of serializability

The schedule below produces same outcome as the serial schedule < T1, T5 >, yet is not conflict equivalent or view equivalent to it.

T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

- Not conflict-serializable or view-serializable, but serializable
- Mainly because of the +/- only operations
  - Requires analysis of the actual operations, not just read/write operations
- Most high-performance transaction systems will allow these

Testing for conflict-serializability

Given a schedule, determine if it is conflict-serializable

Draw a *precedence-graph* over the transactions

- A directed edge from T1 and T2, if they have conflicting instructions, and T1's conflicting instruction comes first

If there is a cycle in the graph, not conflict-serializable

- Can be checked in at most $O(n+e)$ time, where n is the number of vertices, and e is the number of edges

If there is none, conflict-serializable

Testing for view-serializability is NP-hard.

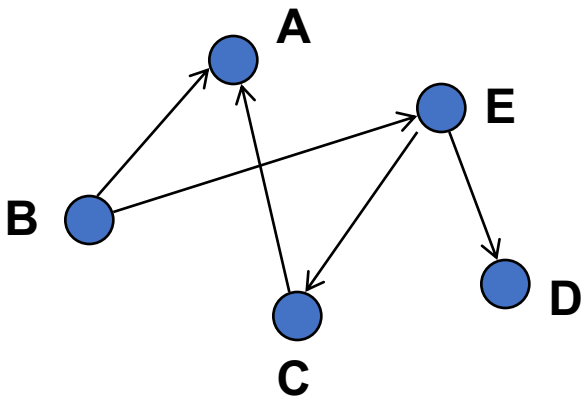
Precedence Graph

- Testing for conflict serializability
- T = {T_1,….T_n}, a set of transactions.

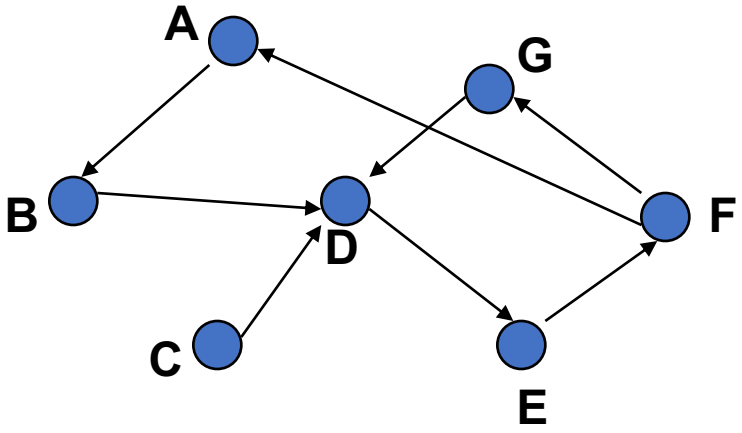
- The Preceduce graph of a schedule S is a directed graph G(S) = (V,E), where
  - V = {T_1, …T_n} is a set of vertices;
  - E consists of edges (T_i,T_j) if
  - one of T_i's operations precedes and conflicts with one of T_j's operations in S.

- Cycle in a graph
  - A cycle is a path that starts and terminates at the same node

- Examples



Does not contain cycle

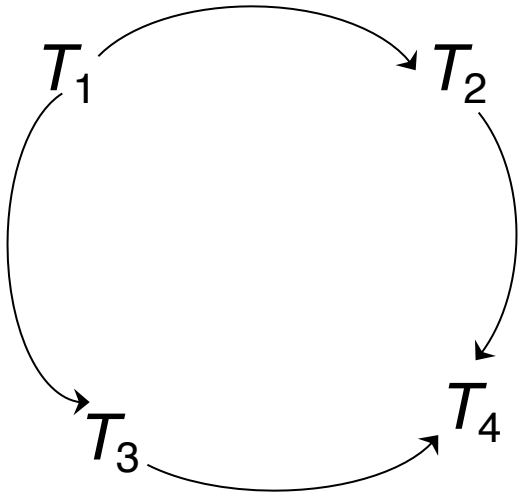


Contains cycles

- **Theorem**: Schedule is conflict serializable if and only if its dependence graph is acyclic

Example Schedule (Schedule A) + Precedence Graph

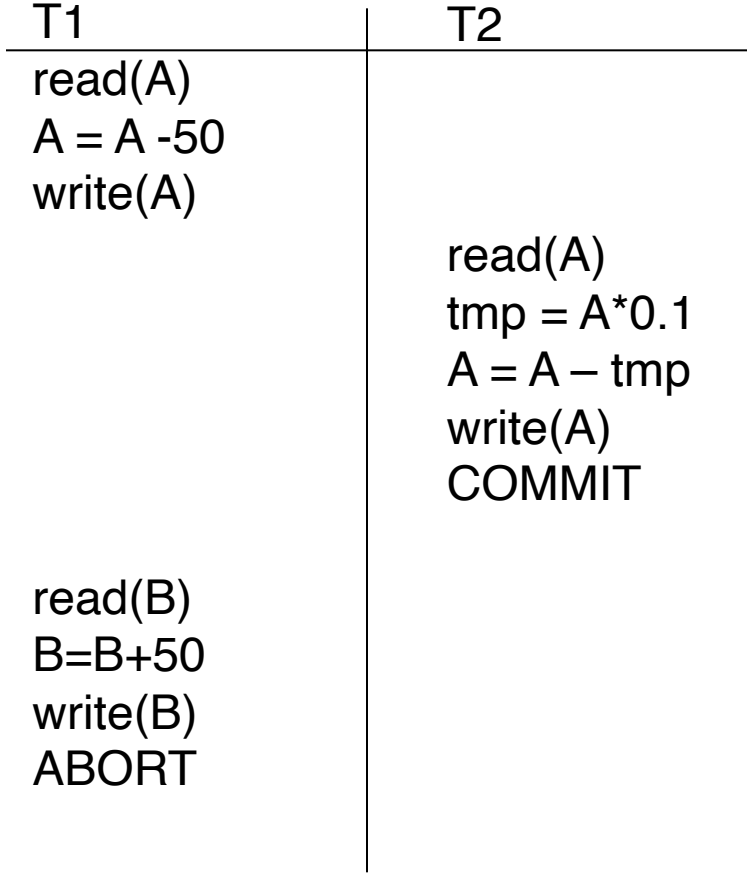
T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				



Recoverability

- Serializability is good for consistency

- But what if transactions fail ?
  - T2 has already committed
    - A user might have been notified
  - Now T1 abort creates a problem
    - T2 has seen its effect, so just aborting T1 is not enough. T2 must be aborted as well (and possibly restarted)
    - But T2 is *committed*



■**Database must ensure that schedules are recoverable**.

Recoverability

- Recoverable schedule: If T1 has read something that T2 has written, T2 must commit before T1
  - Otherwise, if T1 commits, and T2 aborts, we have a problem
- Cascading rollbacks: If T10 aborts, T11 must abort, and hence T12 must abort and so on.

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

Cascadeless Schedules

- **Cascadeless schedules** — cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Notes

- We discussed:
  - Serial schedules, serializability
  - Conflict-serializability, view-serializability
  - How to check for conflict-serializability
  - Recoverability, cascade-less schedules

- We haven't discussed:
  - How to guarantee serializability ?
    - Allowing transactions to run, and then aborting them if the schedules wasn't serializable is clearly not the way to go
  - We instead use schemes to guarantee that the schedule will be conflict-serializable