

CS 4604: Introduction to Database Management Systems

B. Aditya Prakash

Lecture #16: Transactions 1: Intro. to
ACID

Why Transactions?

- Database systems are normally being accessed by many users or processes at the same time.
 - Both queries and modifications.
- Unlike operating systems, which support interaction of processes, a DMBS needs to keep processes from troublesome interactions.

Transactions - dfn

- = unit of work, eg.
 - move \$10 from savings to checking

Statement of Problem

- Concurrent execution of independent transactions (why do we want that?)

Statement of Problem

- Concurrent execution of independent transactions
 - utilization/throughput (“hide” waiting for I/Os.)
 - response time

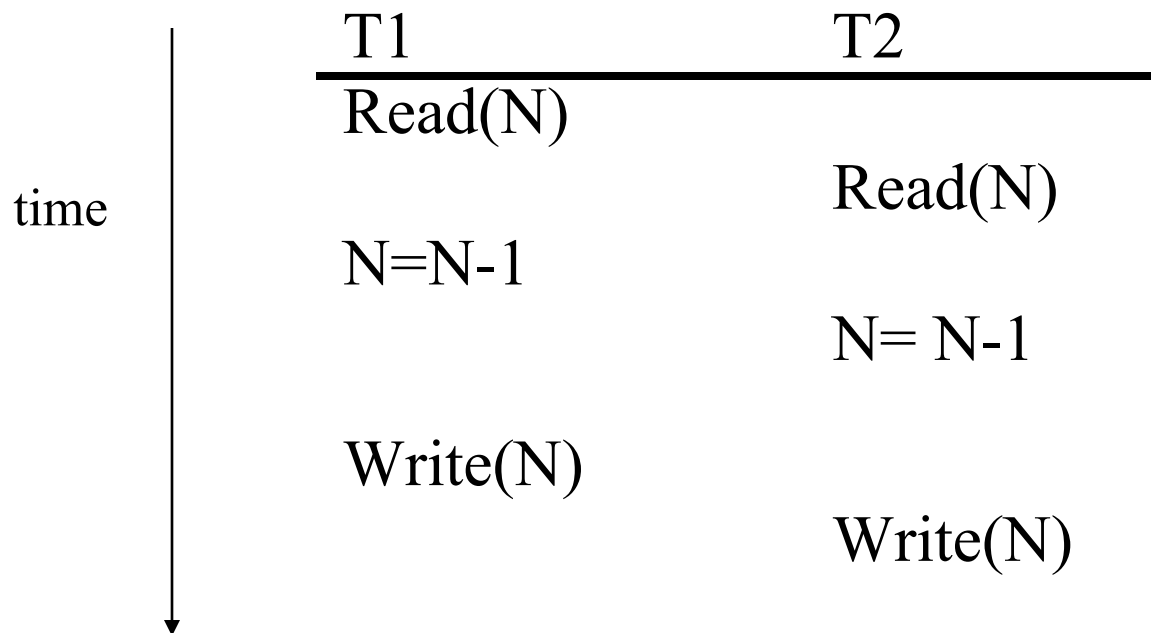
Statement of Problem

- Concurrent execution of independent transactions
 - utilization/throughput (“hide” waiting for I/Os.)
 - response time
- would also like:
 - correctness &
 - fairness
- Example: Book an airplane seat

Definitions

- database - a fixed set of named data objects (A, B, C, \dots)
- transaction - a sequence of read and write operations ($read(A), write(B), \dots$)
 - DBMS' s abstract view of a user program

Example: 'Lost-update' problem



Statement of problem (cont.)

- Arbitrary interleaving can lead to
 - Temporary inconsistency (ok, unavoidable)
 - “Permanent” inconsistency (bad!)
- Need formal correctness criteria.

Example: Bad Interaction

- You and friend each take \$100 from different ATMs at about the same time.
 - The DBMS better make sure one account deduction doesn't get lost.
- **Compare:** An OS allows two people to edit a document at the same time. If both write, one's changes get lost.

ACID Transactions

- *ACID transactions* are:
 - *Atomic* : Whole transaction or none is done.
 - *Consistent* : Database constraints preserved.
 - *Isolated* : It appears to the user as if only one process executes at a time.
 - *Durable* : Effects of a process survive a crash.
- **Optional**: weaker forms of transactions are often supported as well (like Google, Amazon system etc.): Recall NoSQL systems

COMMIT

- The SQL statement COMMIT causes a transaction to complete.
 - It's database modifications are now permanent in the database.

ROLLBACK

- The SQL statement ROLLBACK also causes the transaction to end, but by *aborting*.
 - No effects on the database.
- Failures like division by 0 or a constraint violation can also cause rollback, even if the programmer does not request it.

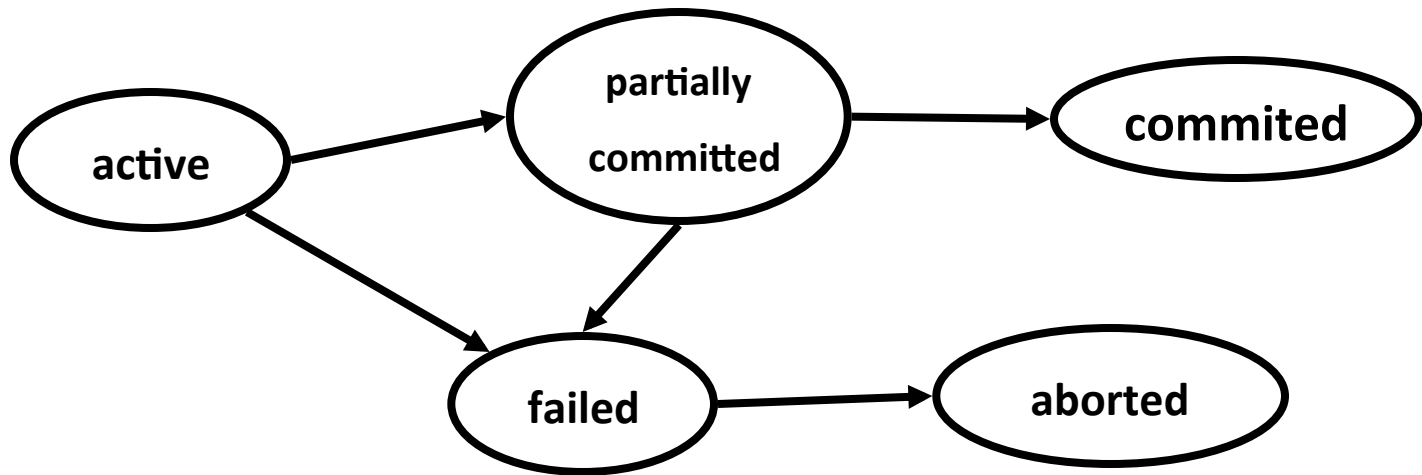
Overview

- *ACID transactions* are:
 - *Atomic* : **Whole transaction or none is done.**
 - *Consistent* : Database constraints preserved.
 - *Isolated* : It appears to the user as if only one process executes at a time.
 - *Durable* : Effects of a process survive a crash.

A Atomicity of Transactions

- Two possible outcomes of executing a transaction:
 - Xact might *commit* after completing all its actions
 - or it could *abort* (or be aborted by the DBMS) after executing some actions.
- DBMS guarantees that Xacts are *atomic*.
 - From user's point of view: Xact always either executes all its actions, or executes no actions at all.

Transaction states



A Mechanisms for Ensuring Atomicity

- What would you do?

A Mechanisms for Ensuring Atomicity

- One approach: LOGGING
 - DBMS logs all actions so that it can undo the actions of aborted transactions.
- ~ like black box in airplanes ...

A Mechanisms for Ensuring Atomicity

- Logging used by all modern systems.
- Q: why?

A Mechanisms for Ensuring Atomicity

- Logging used by all modern systems.
- Q: why?
- A:
 - audit trail &
 - efficiency reasons
- What other mechanism can you think of?

A Mechanisms for Ensuring Atomicity

- Another approach: SHADOW PAGES
 - (not as popular)

Overview

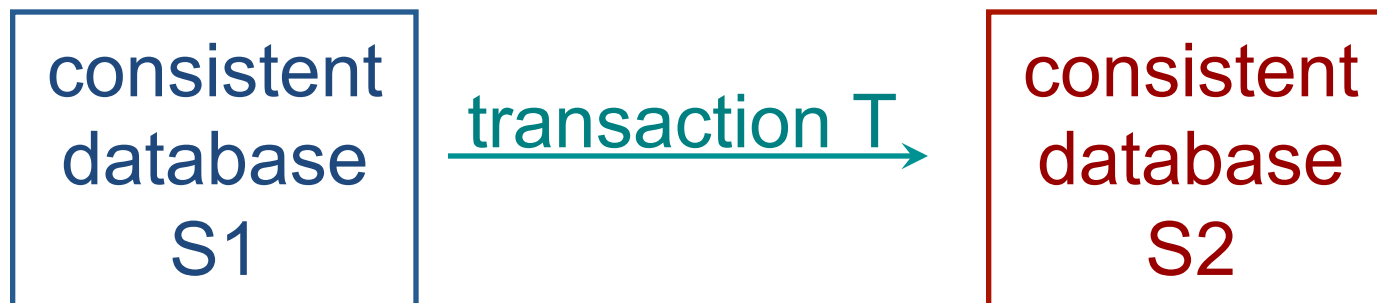
- *ACID transactions* are:
 - *Atomic* : Whole transaction or none is done.
 - *Consistent* : **Database constraints preserved.**
 - *Isolated* : It appears to the user as if only one process executes at a time.
 - *Durable* : Effects of a process survive a crash.

C Transaction Consistency

- “Database consistency” - data in DBMS is accurate in modeling real world and follows integrity constraints

C Transaction Consistency

- “Transaction Consistency”: if DBMS consistent before Xact (running alone), it will be after also
- Transaction consistency: User’s responsibility
 - DBMS just checks IC



C Transaction Consistency (cont.)

- Recall: Integrity constraints
 - must be true for DB to be considered consistent

Examples:

1. FOREIGN KEY R.sid REFERENCES S
2. ACCT-BAL ≥ 0

C Transaction Consistency (cont.)

- System checks ICs and if they fail, the transaction rolls back (i.e., is aborted).
 - Beyond this, DBMS does not understand the semantics of the data.
 - e.g., it does not understand how interest on a bank account is computed
- Since it is the user's responsibility, we don't discuss it further

Overview

- *ACID transactions* are:
 - *Atomic* : Whole transaction or none is done.
 - *Consistent* : Database constraints preserved.
 - *Isolated* : **It appears to the user as if only one process executes at a time.**
 - *Durable* : Effects of a process survive a crash.

I Isolation of Transactions

- Users submit transactions, and
- Each transaction executes as if it was running by itself.
 - Concurrency is achieved by DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
- Q: How would you achieve that?
 - Tough problem!

I Isolation of Transactions

- A: Many methods - two main categories:
- Pessimistic – don't let problems arise in the first place
- Optimistic – assume conflicts are rare, deal with them after they happen.

Example

- Consider two transactions (Xacts):

```
T1: BEGIN  A=A+100,  B=B-100  END  
T2: BEGIN  A=1.06*A, B=1.06*B  END
```

- 1st xact transfers \$100 from B's account to A's
- 2nd credits both accounts with 6% interest.
- Assume at first A and B each have \$1000. What are the **legal outcomes** of running T1 and T2?

I

Example

```
T1: BEGIN  A=A+100, B=B-100  END
T2: BEGIN  A=1.06*A, B=1.06*B  END
```

- many - but $A+B$ should be: $\$2000 * 1.06 = \2120
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. **But, the net effect *must* be equivalent to these two transactions running serially in some order.**

I Example (Contd.)

- Legal outcomes: $A=1166, B=954$ or $A=1160, B=960$
- Consider a possible interleaved schedule:

T1:	$A=A+100,$	$B=B-100$
T2:	$A=1.06*A,$	$B=1.06*B$

- This is OK (same as T1;T2). But what about:

T1:	$A=A+100,$	$B=B-100$
T2:	$A=1.06*A, B=1.06*B$	

I Example (Contd.)

- Legal outcomes: $A=1166, B=954$ or $A=1160, B=960$
- Consider a possible interleaved schedule:

T1:	$A=A+100,$	$B=B-100$
T2:	$A=1.06*A,$	$B=1.06*B$

- This is OK (same as T1;T2). But what about:

T1:	$A=A+100,$	$B=B-100$
T2:	$A=1.06*A,$	$B=1.06*B$

- Result: $A=1166, B=960; A+B = 2126$, bank loses \$6**
- The DBMS' s view of the second schedule:**

T1:	$R(A), W(A),$	$R(B), W(B)$
T2:	$R(A), W(A), R(B), W(B)$	

‘Correctness’ ?

- Q: How would you judge that a schedule is ‘correct’ ?
(‘schedule’ = ‘interleaved execution’)

‘Correctness’ ?

- Q: How would you judge that a schedule is ‘correct’ ?
- A: if it is equivalent to some serial execution

I Formal Properties of Schedules

- Serial schedule: Schedule that does not interleave the actions of different transactions.
- Equivalent schedules: For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule. (*)

(*) no matter what the arithmetic etc. operations are!

I Formal Properties of Schedules

- Serializable schedule: A schedule that is equivalent to some serial execution of the transactions.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)

Anomalies with interleaved execution:

- R-W conflicts
- W-R conflicts
- W-W conflicts

(why not R-R conflicts?)



I

Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, “dirty reads”):

T1: R(A), W(A),	R(B), W(B), Abort
T2: R(A), W(A), C	

I Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, “dirty reads”):

T1: R(A), W(A),	R(B), W(B), Abort
T2: R(A), W(A), C	

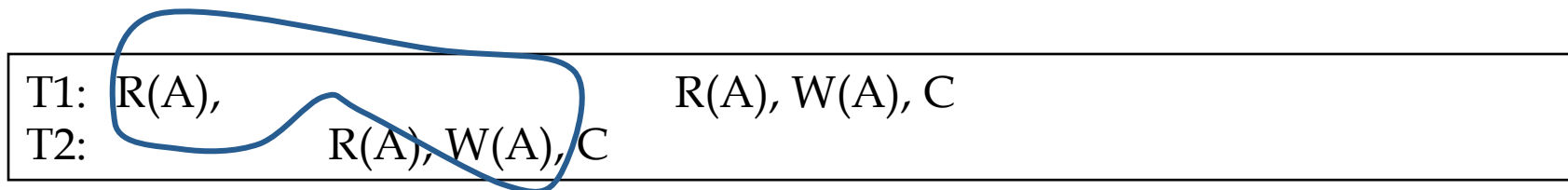
I Anomalies with Interleaved Execution

- Unrepeatable Reads (RW Conflicts):

T1: R(A),	R(A), W(A), C
T2:	R(A), W(A), C

I Anomalies with Interleaved Execution

- Unrepeatable Reads (RW Conflicts):



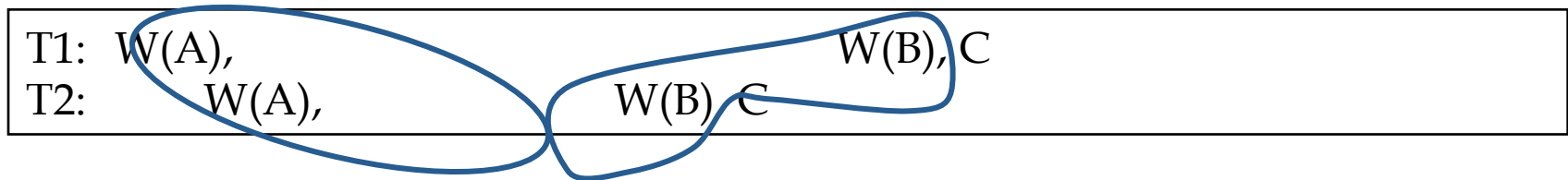
I Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

T1: W(A),		W(B), C
T2: W(A),	W(B), C	

I Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):



Serializability

- Objective: find non-serial schedules, which allow transactions to execute concurrently without interfering, thereby producing a DB state that could be produced by a serial execution
- BUT
 - Trying to find schedules equivalent to serial execution is too slow!

Conflict Serializability

- We need a formal notion of equivalence that can be implemented efficiently...
 - Base it on the notion of “conflicting” operations
- Definition: Two operations conflict if:
 - They are by different transactions,
 - they are on the same object,
 - and at least one of them is a write.

Conflict Serializable Schedules

- Definition: Two schedules are conflict equivalent iff:
 - They involve the same actions of the same transactions, and
 - every pair of conflicting actions is ordered the same way
- Definition: Schedule S is conflict serializable if:
 - S is conflict equivalent to some serial schedule.
- Note, some “serializable” schedules are NOT conflict serializable (See Example 4 later)

CS---Intuition

- A schedule S is conflict serializable if:
 - You are able to transform S into a serial schedule by swapping consecutive non-conflicting operations of different transactions

R(A) W(A)

R(B) W(B)

R(A) W(A)

R(B) W(B)

====

R(A) W(A) R(B) W(B)

R(A) W(A) R(B) W(B)

CS---Intuition

- A schedule S is conflict serializable if:
 - You are able to transform S into a serial schedule by swapping consecutive non-conflicting operations of different transactions

$R(A)$

$W(A)$

$R(A) W(A)$

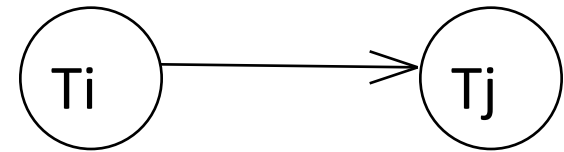
IS NOT SERIALIZABLE!

Serializability

- Q: any faster algorithm? (faster than transposing operations?)

Dependency Graph

- One node per Xact
- Edge from T_i to T_j if:



- An operation O_i of T_i conflicts with an operation O_j of T_j and
- O_i appears earlier in the schedule than O_j .

Dependency Graph: Theorem

- THEOREM: Schedule is conflict serializable iff the dependency graph is acyclic

- Dependency graph is also called the precedence graph
 - different than the waits-for graph we will see later

Example #2 (Lost update)

T1

Read(N)

$N = N - 1$

Write(N)

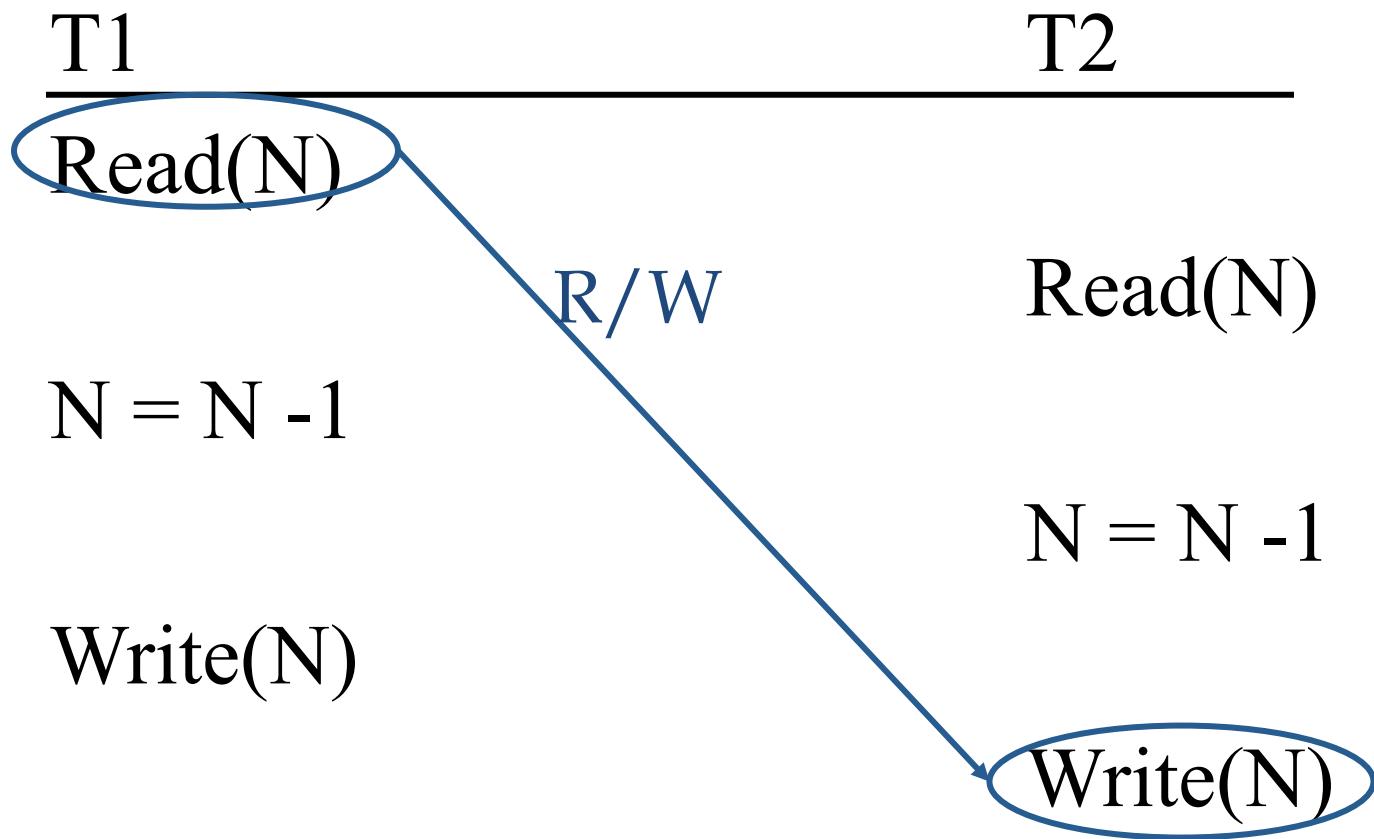
T2

Read(N)

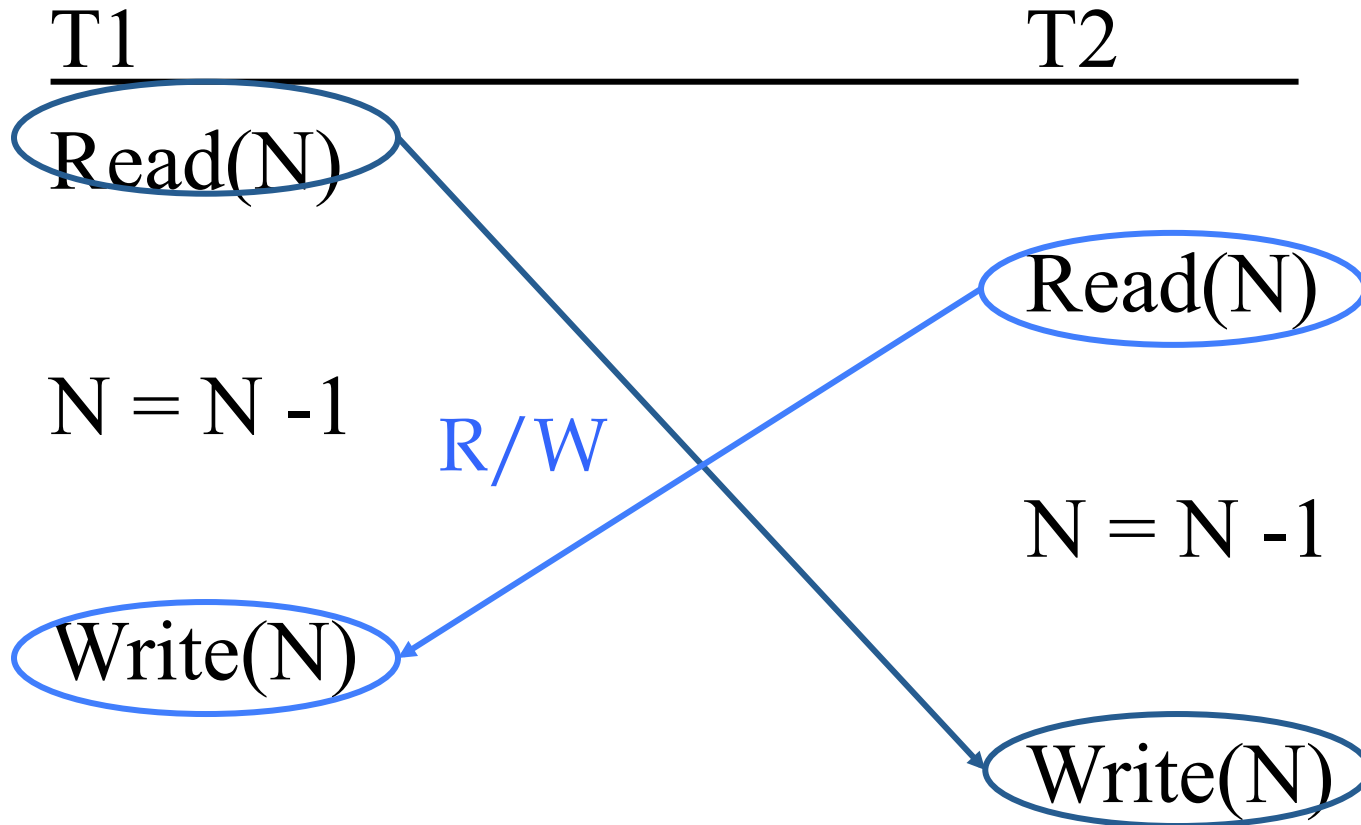
$N = N - 1$

Write(N)

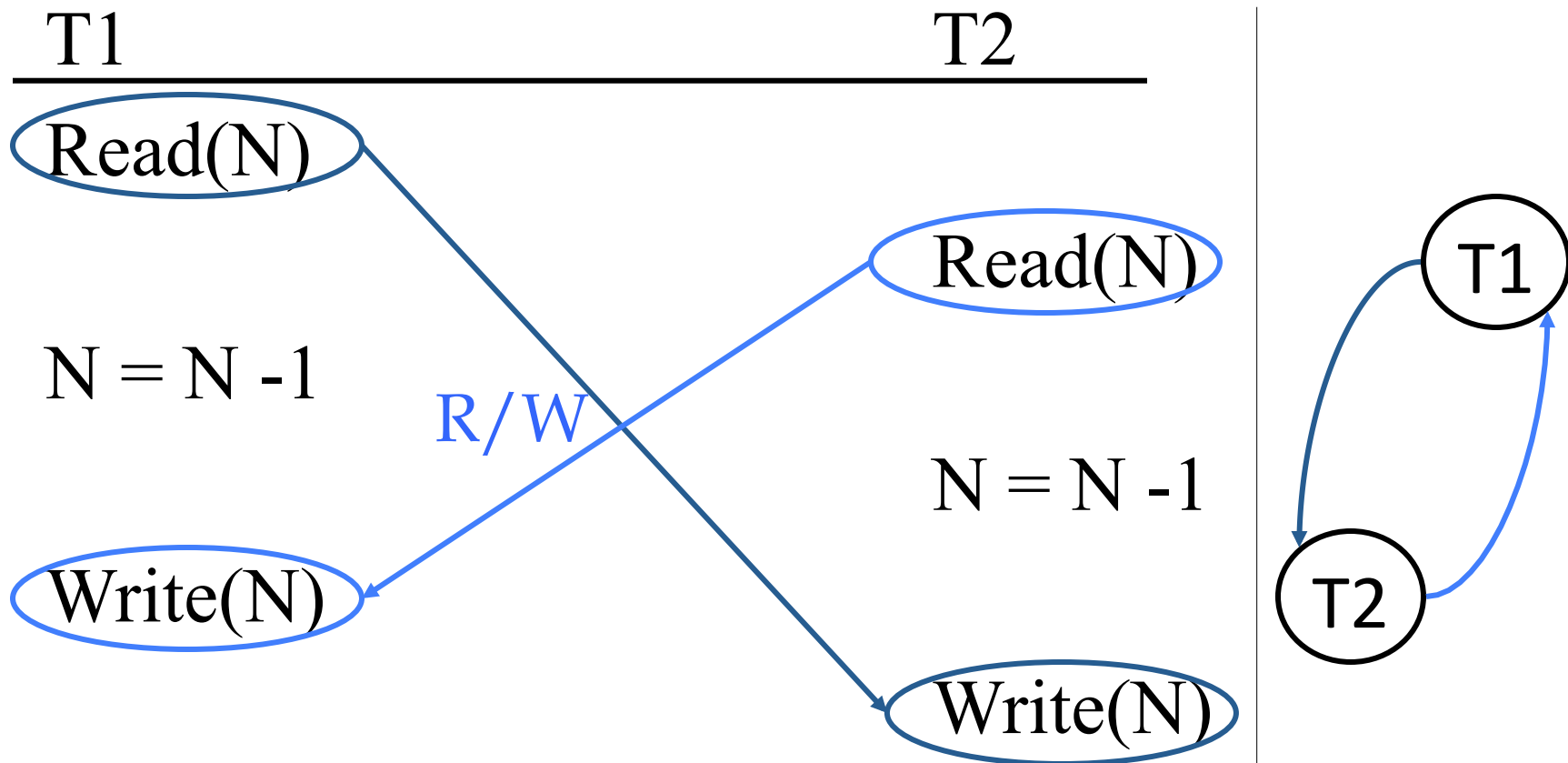
Example #2 (Lost update)



Example #2 (Lost update)



Example #2 (Lost update)

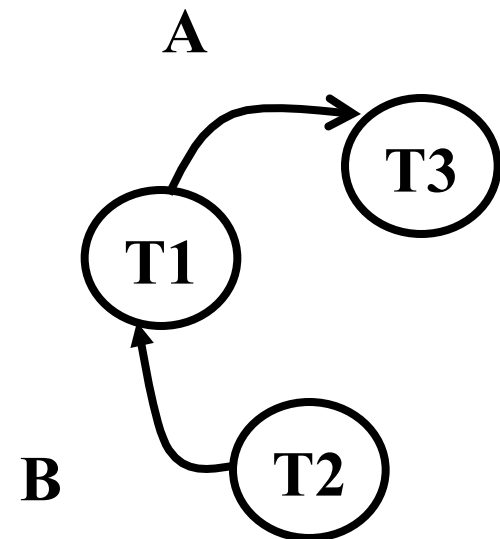


Example #3

T1	T2	T3
Read(A)		
...		
write(A)		
		Read(A)
		...
		Write(A)
	Read(B)	
	...	
	Write(B)	
Read(B)		
...		
Write(B)		

Example #3

T1	T2	T3
Read(A)		
...		
write(A)		
		Read(A)
		...
		Write(A)
	Read(B)	
	...	
	Write(B)	
Read(B)		
...		
Write(B)		



equivalent serial execution?

Example #3

- A: T2, T1, T3

(Notice that T3 should go after T2 in the equivalent serial order, although it starts before it!)

- Q: algo for generating serial execution from (acyclic) dependency graph?

Example #3

- A: T2, T1, T3

(Notice that T3 should go after T2 in the equivalent serial order, although it starts before it!)

- Q: algo for generating serial execution from (acyclic) dependency graph?
- A: Topological sorting

Example #4 (Inconsistent Analysis)

T1

T2

R (A)

A = A-10

W (A)

R(A)

Sum = A

R (B)

Sum += B

R(B)

B = B+10

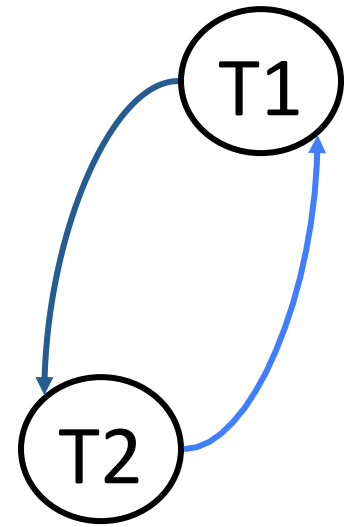
W(B)

dependency
graph?

Example #4 (Inconsistent Analysis)

T1	T2
R (A)	
A = A-10	
W (A)	
	R(A)
	Sum = A
	R (B)
	Sum += B
R(B)	
B = B+10	
W(B)	

dependency graph?



So NOT Conflict
Serializable (and not
serializable)

Example #4 (Inconsistent Analysis)

T1	T2	
R (A)		Q: create a
A = A-10		‘correct’
W (A)		Schedule based on
	R(A)	this one that is not
	Sum = A	conflict-serializable
	R (B)	
	Sum += B	
R(B)		
B = B+10		
W(B)		

Example #4' (Inconsistent Analysis)

T1	T2	
R (A)		A: T2 asks for the count of my active Accounts (assuming $A > 10$, $B > 0$)
$A = A - 10$		
W (A)		
<hr/>		
	R(A)	
	$if (A > 0), count = 1$	
	R (B)	
	$if (B > 0), count++$	
<hr/>		
R(B)		
$B = B + 10$		
W(B)		

NOTES:

1. This schedule is still not CS
2. BUT it is serializable! It is equivalent to either of [T1 T2] or [T2 T1] (both are OK)

Serializability in Practice

- DBMS does not test for conflict serializability of a given schedule
 - Impractical as interleaving of operations from concurrent Xacts could be dictated by the OS
- Approach:
 - Use specific protocols that are known to produce conflict serializable schedules
 - But may reduce concurrency

Solution?

- One solution for “conflict serializable” schedules is Two Phase Locking (2PL)

Answer

- (Full answer:) use locks; keep them until commit (‘strict 2 phase locking’)
- We’ll see the details later (in next class!)

(Review) Goal: ACID Properties

- *ACID transactions* are:
 - *Atomic* : Whole transaction or none is done.
 - *Consistent* : Database constraints preserved.
 - *Isolated* : It appears to the user as if only one process executes at a time.
 - *Durable* : Effects of a process survive a crash.

What happens if system crashes between *commit* and *flushing modified data to disk* ?

D

Durability

- == Recovery
- We'll see it **later** (after concurrency control)

Summary

- **Concurrency control** and **recovery** are among the most important functions provided by a DBMS.
- Concurrency control is automatic
 - System automatically inserts lock/unlock requests and schedules actions of different Xacts
 - Property ensured: resulting execution is equivalent to executing the Xacts one after the other in some order.

ACID properties

Atomicity (all or none)

Consistency

Isolation (as if alone)

Durability

