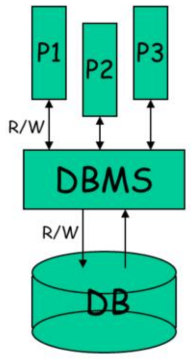
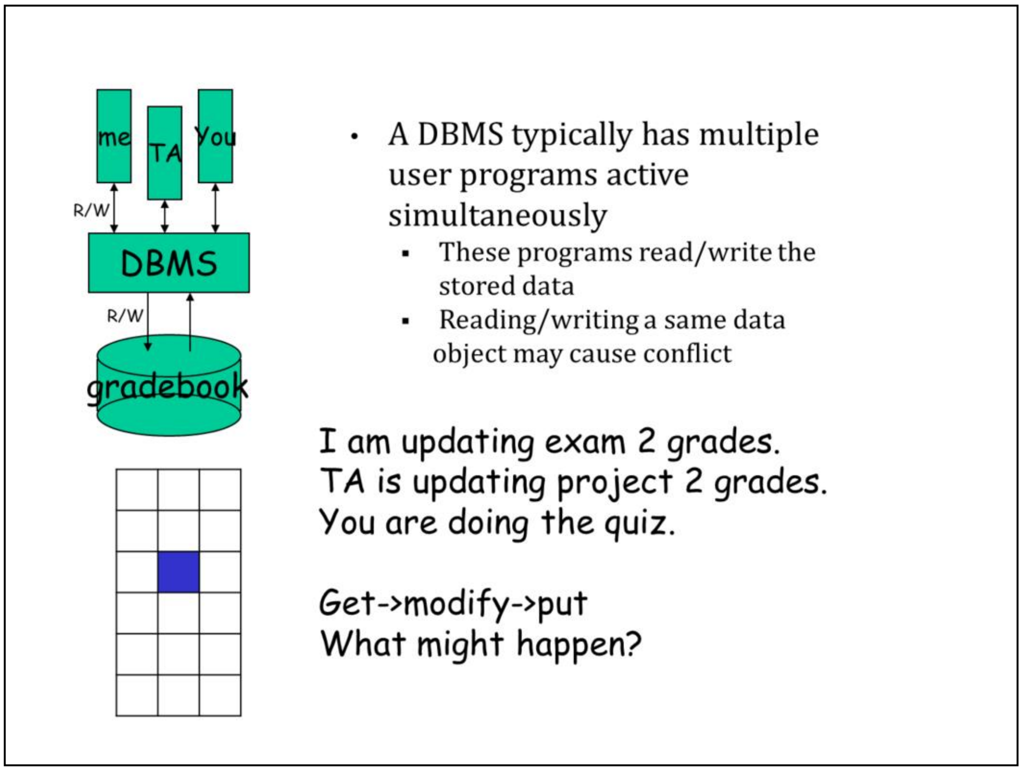
Transaction Management

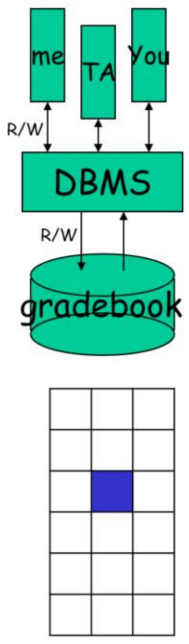
- A DBMS typically has multiple user programs active simultaneously
  - These programs read/write the stored data
  - Reading/writing a same data object may cause conflict
- Simplest solution is to execute programs one by one
  - Disk accesses are frequent, and relatively slow
  - Want to keep CPU working on several user programs concurrently
- Two questions
  - How to handle concurrent program executions?
  - How to handle machine crashes or a program execution being aborted?
- Topics to cover
  - Schedules
  - Concurrency control
  - Multi-Granularity locking



quiz on transaction management 1 & 2

- Access code:
  - ACID
- quiz on transaction management 1
  - 4/7 lecture
- quiz on transaction management 2
  - 4/9 lecture



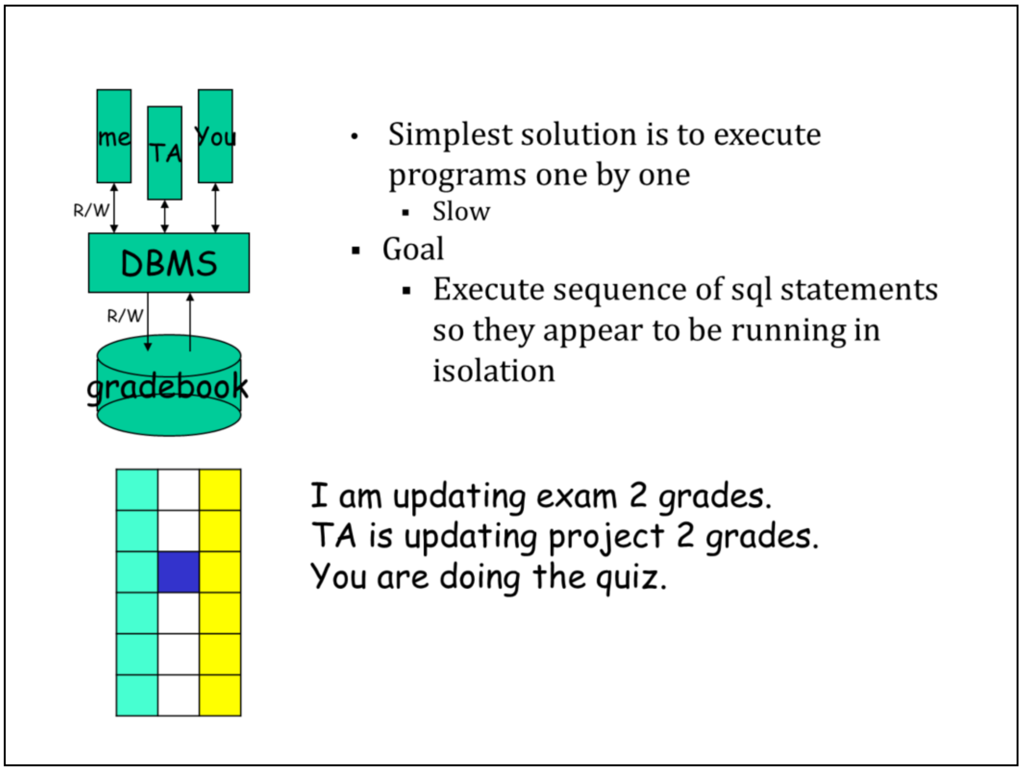


- A DBMS typically has multiple user programs active simultaneously
  - These programs read/write the stored data
  - Reading/writing a same data object may cause conflict

I am updating exam 2 grades.
TA is updating project 2 grades.
You are doing the quiz.

Get->modify->put
What might happen?

In this example, the gradebook is the database. On top of the data, DBMS is managing all executions on the database. Multiple database users are trying to modify the database at the same time. So what should the DBMS do?

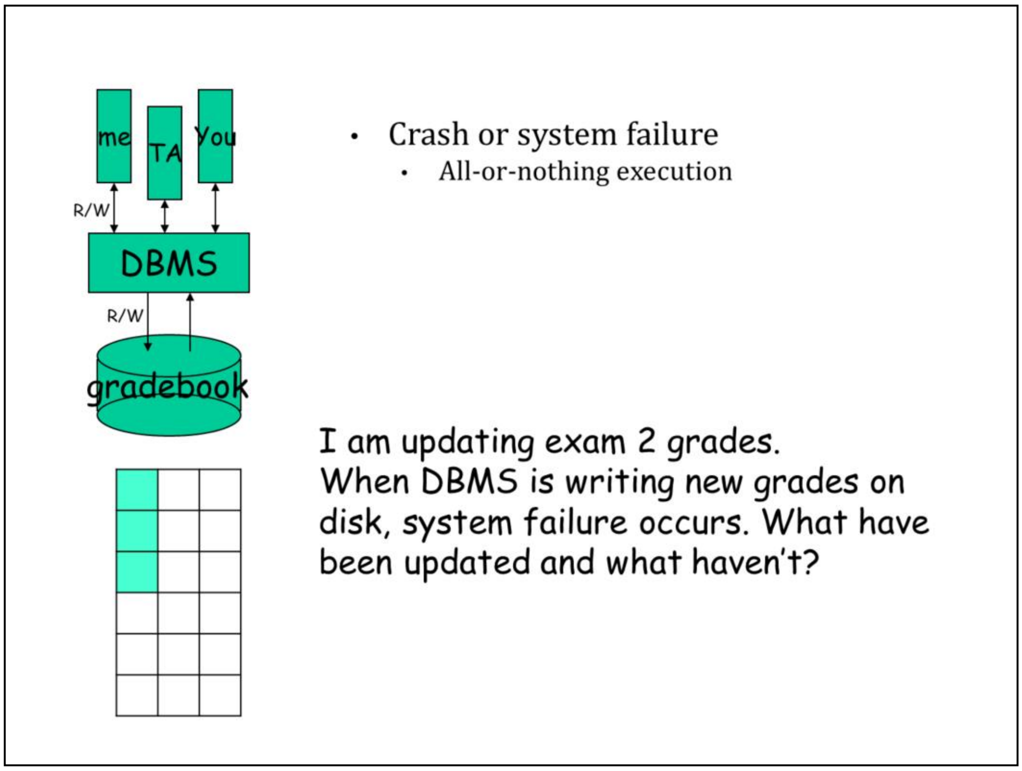


- Simplest solution is to execute programs one by one
  - Slow
- Goal
  - Execute sequence of sql statements so they appear to be running in isolation

I am updating exam 2 grades.
TA is updating project 2 grades.
You are doing the quiz.

If no concurrency is allowed, then when I am updating the gradebook, no one else can access the gradebook. Obviously, this is very inconvenient. So we do want the DBMS to manage concurrent executions. We also want correct results.

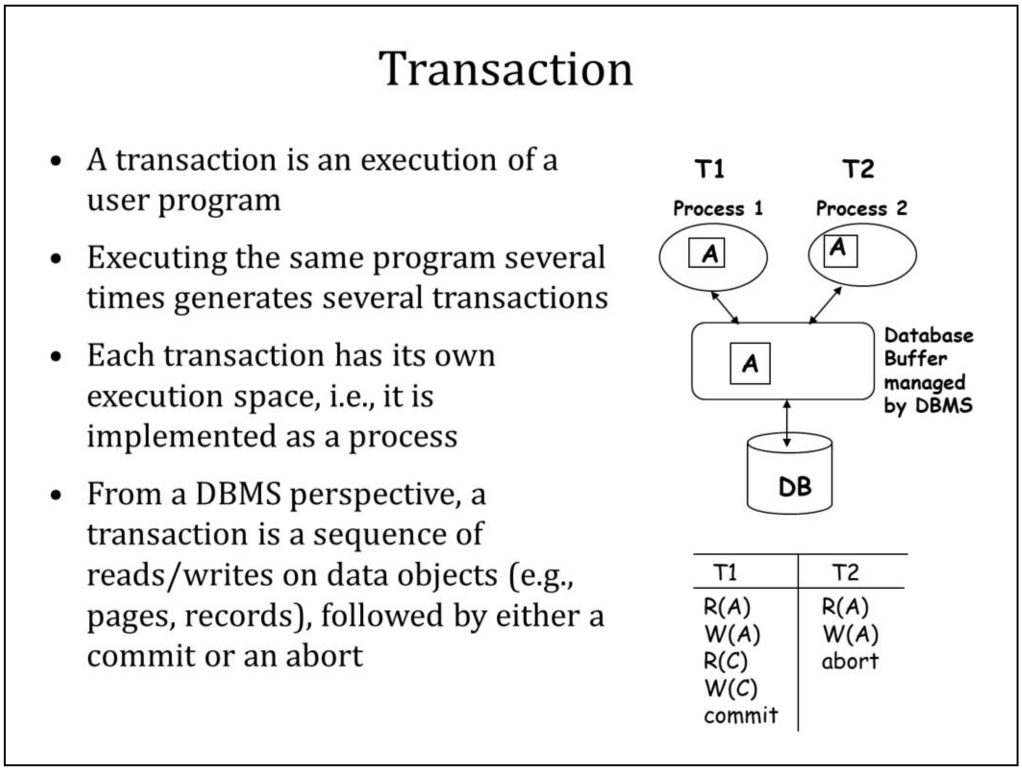
So what can go wrong? Recall how we access the data. We send query to DBMS, then DBMS would find the page, modify the record, and then write it back to disk. Suppose I need to modify the blue part of this page, TA need to modify the yellow part of the page, and you need to modify the navy part of this page. If we query the page at the same time, each of us will get a local copy of the page. When we are done, we may overwrite each other’s update.



- Crash or system failure
  - All-or-nothing execution

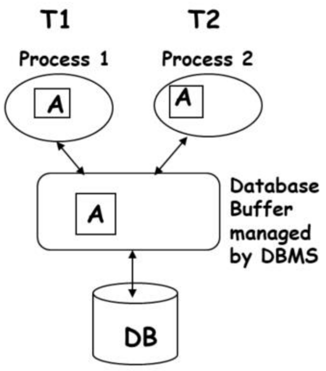
I am updating exam 2 grades.
When DBMS is writing new grades on disk, system failure occurs. What have been updated and what haven't?

Another problem may occur in transaction is system failure. Suppose I am updating exam 2 grades. When DBMS is writing new grades on disk, system failure occurs, let's say, the DBMS server is cut of from power. When it's back on, DBMS needs to know what have been updated and what haven't



Transaction

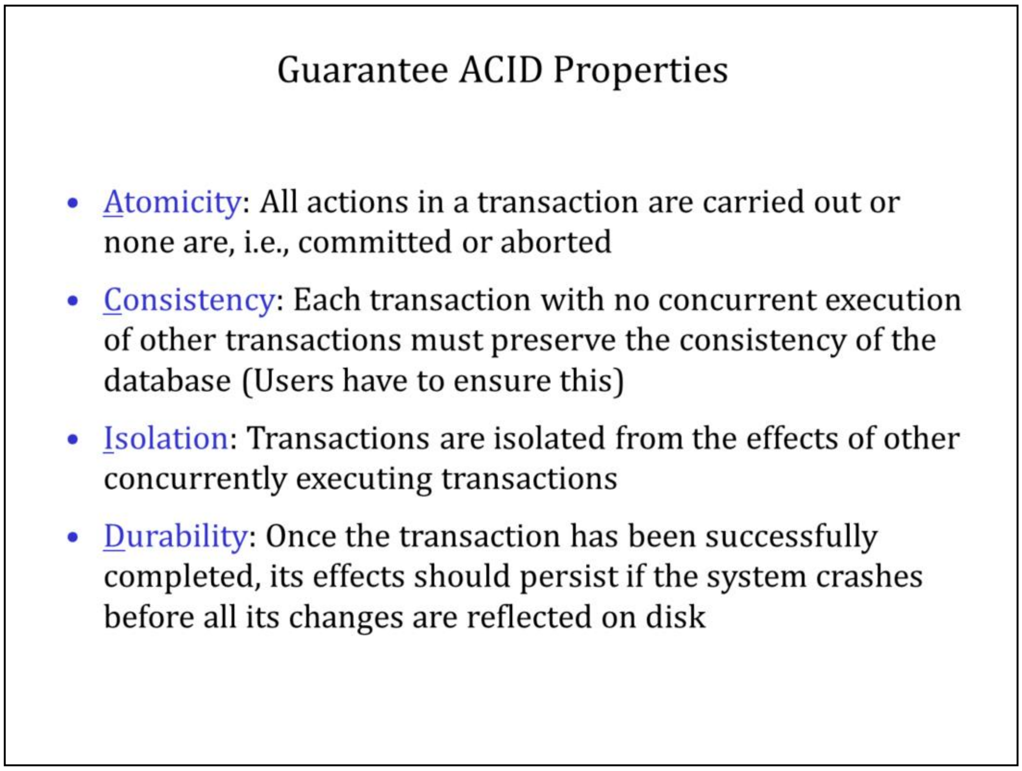
- A transaction is an execution of a user program
- Executing the same program several times generates several transactions
- Each transaction has its own execution space, i.e., it is implemented as a process
- From a DBMS perspective, a transaction is a sequence of reads/writes on data objects (e.g., pages, records), followed by either a commit or an abort



T1	T2
R(*A*)	R(*A*)
W(*A*)	W(*A*)
R(*C*)	abort
W(*C*)	
commit	

Concurrent execution of user programs is essential for good DBMS performance. A user’s program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database. A transaction is the DBMS’s abstract view of a user program: a sequence of reads and write.

In addition to reading and writing, each transaction must specify as its final action either commit, that is complete successfully, or abort, that is terminate and undo all the actions carried out thus far.



Guarantee ACID Properties

- Atomicity: All actions in a transaction are carried out or none are, i.e., committed or aborted
- Consistency: Each transaction with no concurrent execution of other transactions must preserve the consistency of the database (Users have to ensure this)
- Isolation: Transactions are isolated from the effects of other concurrently executing transactions
- Durability: Once the transaction has been successfully completed, its effects should persist if the system crashes before all its changes are reflected on disk

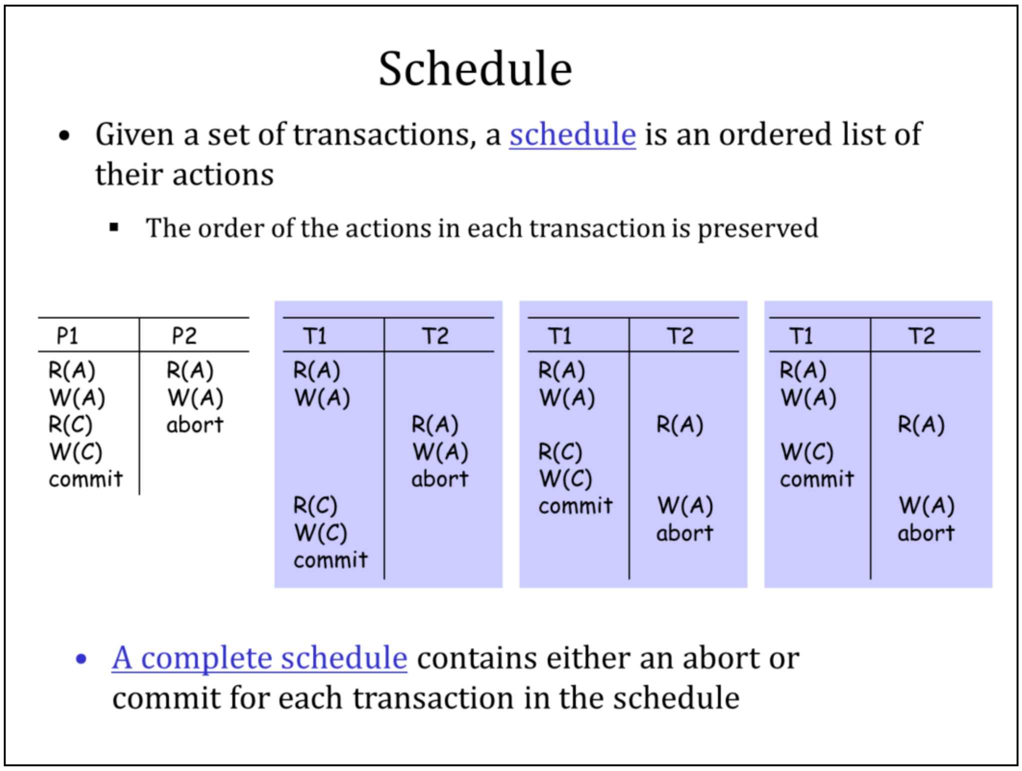
A DBMS must ensure four important properties of transactions to maintain data in the face of concurrent access and system failures.

Atomicity. Either all actions are carried out or none are. Users should not have to worry about the effect of incomplete transactions, say, when a system crash occurs.

Consistency: Each transaction, run by itself with no concurrent execution of other transactions must preserve the consistency of the database. Ensuring this property of a transaction is the responsibility of the user.

Isolation: users should be able to understand a transaction without considering the effect of other concurrently executing transactions, even if the DBMS interleaves the actions of several transactions of performance reasons.

Durability: Once the transaction has been successfully completed, its effects should persist if the system crashes before all its changes are reflected on disk.



Schedule

- Given a set of transactions, a schedule is an ordered list of their actions
  - The order of the actions in each transaction is preserved

P1	P2
R(A)	R(A)
W(A)	W(A)
R(C)	abort
W(C)	
commit	

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	abort
R(C)	
W(C)	
commit	

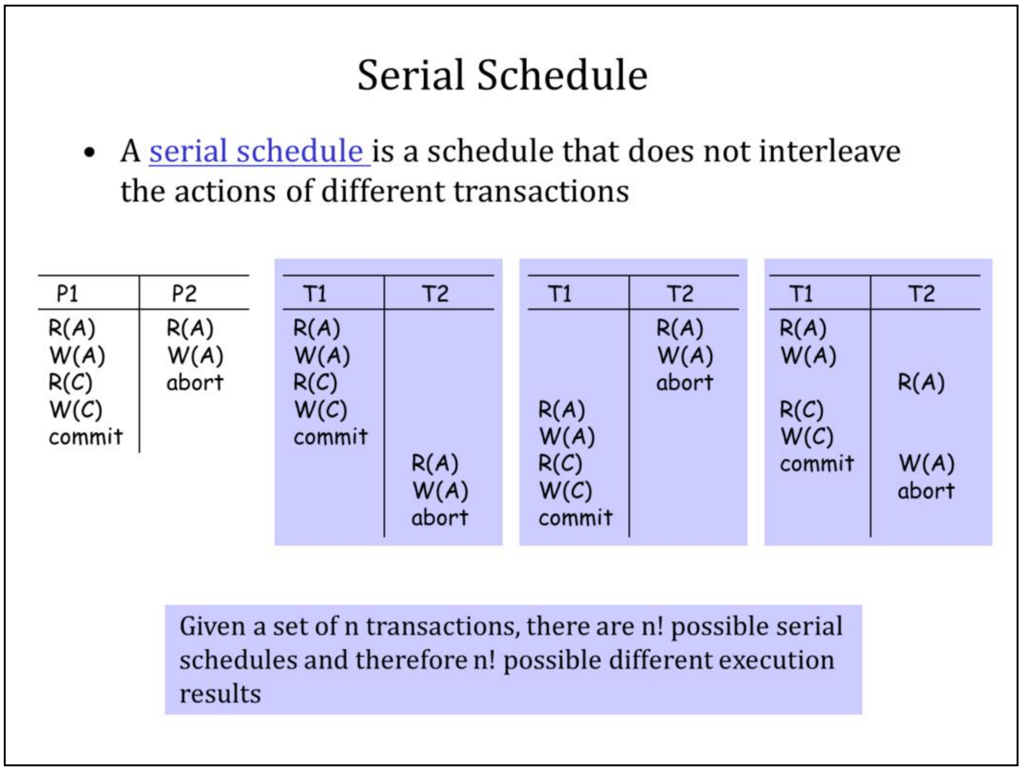
T1	T2
R(A)	
W(A)	
	R(A)
R(C)	
W(C)	
commit	W(A)
	abort

T1	T2
R(A)	
W(A)	
	R(A)
W(C)	
commit	
	W(A)
	abort

- A complete schedule contains either an abort or commit for each transaction in the schedule

A schedule is a list of actions (reading, writing, aborting or committing)

A complete schedule contains either an abort or commit for each transaction in the schedule



Serial Schedule

- A serial schedule is a schedule that does not interleave the actions of different transactions

P1	P2
R(A)	R(A)
W(A)	W(A)
R(C)	abort
W(C)	
commit	

T1	T2
R(A)	
W(A)	
R(C)	
W(C)	
commit	
	R(A)
	W(A)
	abort

T1	T2
	R(A)
	W(A)
	abort
R(A)	
W(A)	
R(C)	
W(C)	
commit	

T1	T2
R(A)	
W(A)	
	R(A)
R(C)	
W(C)	
commit	W(A)
	abort

Given a set of n transactions, there are n! possible serial schedules and therefore n! possible different execution results

If the actions of different transactions are not interleaved, that is, transactions are executed from start to finish, one by one, we call the schedule a serial schedule.

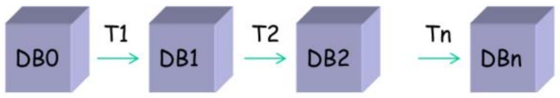
In serial schedules, all the transactions execute serially one after the other.

When one transaction executes, no other transaction is allowed to execute.

1, 2 are serial schedule. 3 is not

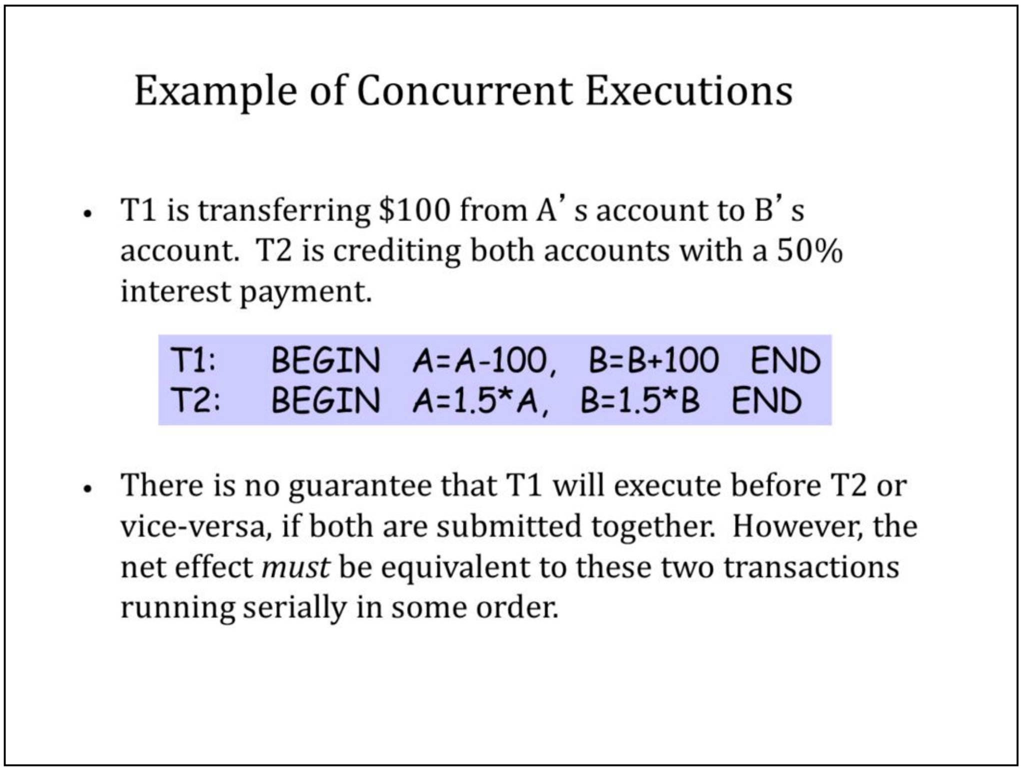
Serializable Schedule

- Given a set of n transactions, there are n! possible serial schedules and therefore n! different execution results



- A schedule is a serializable schedule if its effect on the database is identical to that of some serial schedule (will be refined later on)

The problem now is how to ensure that the order of reads/writes in a given set of transactions forms a serializable schedule!



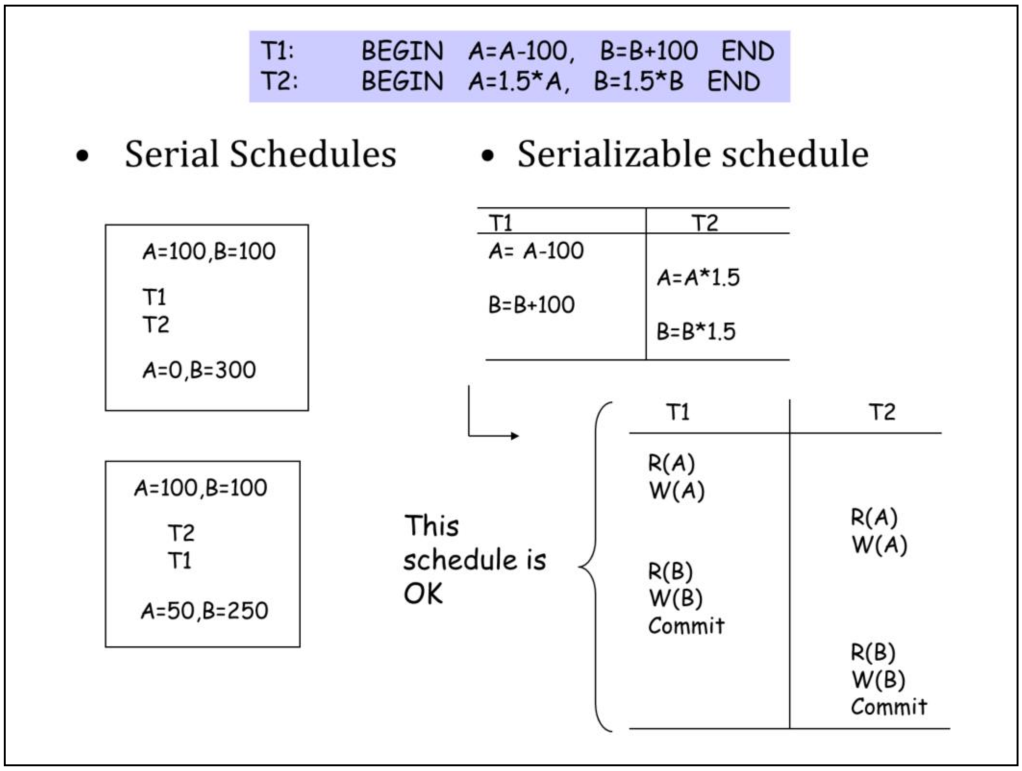
Example of Concurrent Executions

- T1 is transferring \$100 from A's account to B's account. T2 is crediting both accounts with a 50% interest payment.

```
T1:   BEGIN  A=A-100,  B=B+100  END
T2:   BEGIN  A=1.5*A,  B=1.5*B  END
```

- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect *must* be equivalent to these two transactions running serially in some order.

Executing transactions serially in different orders may produce different results, but all are presumed to be acceptable. The DBMS makes no guarantees about which of them will be the outcome of an interleaved execution.



```
T1:    BEGIN  A=A-100,  B=B+100  END
T2:    BEGIN  A=1.5*A,  B=1.5*B  END
```

- Serial Schedules
- Serializable schedule

A=100,B=100

T1
T2

A=0,B=300

A=100,B=100

T2
T1

A=50,B=250

T1	T2
A= A-100	
	A=A*1.5
B=B+100	
	B=B*1.5

This schedule is OK

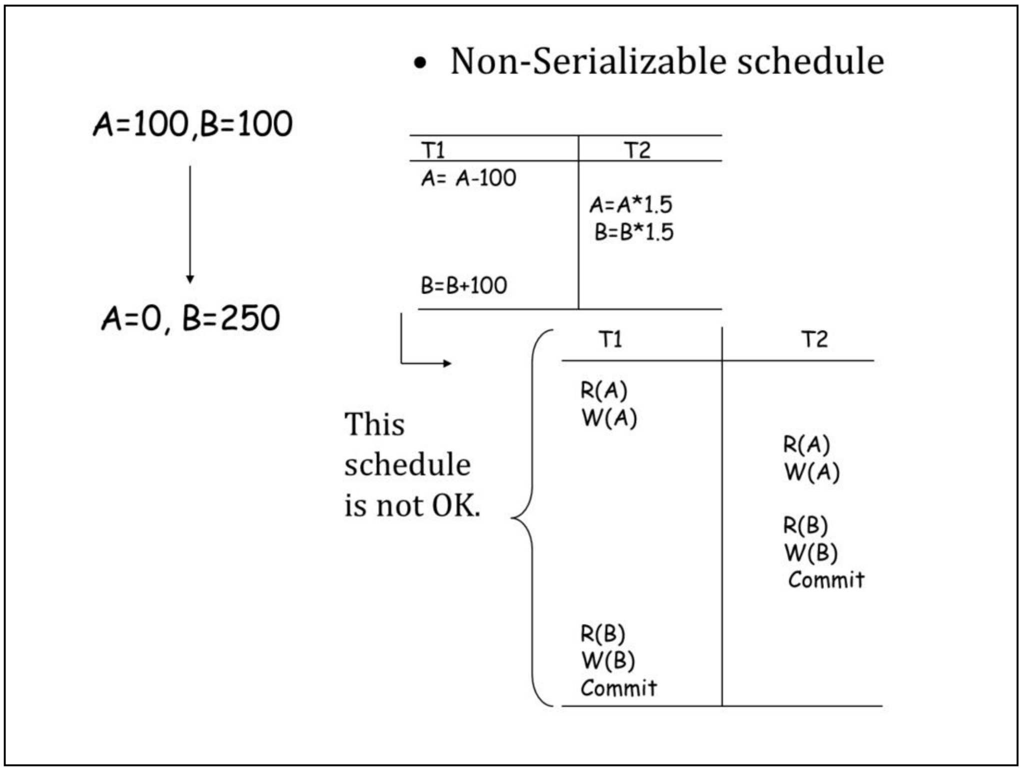
T1	T2

R(A)
W(A)	

	R(A)
	W(A)

R(B)
W(B)
Commit	

	R(B)
	W(B)
	Commit



- Non-Serializable schedule

A=100,B=100

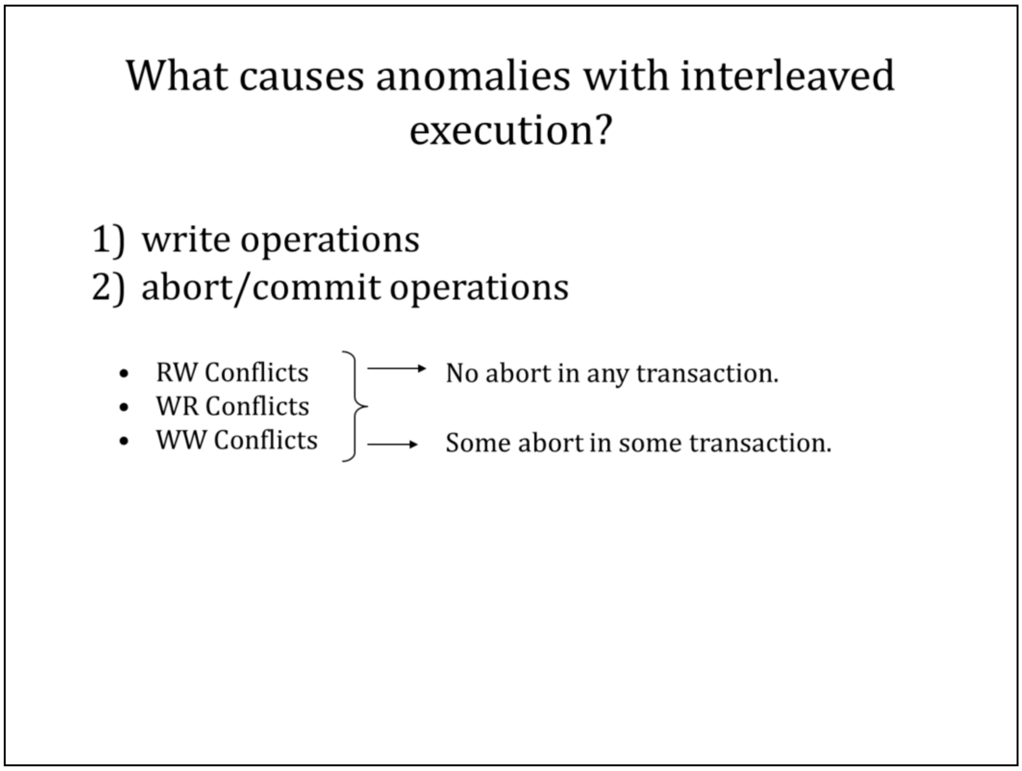
$\downarrow$

A=0, B=250

T1	T2
A= A-100	
	A=A*1.5
	B=B*1.5
B=B+100	

This schedule is not OK.

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
R(B)	
W(B)	
Commit	



What causes anomalies with interleaved execution?

1) write operations
2) abort/commit operations

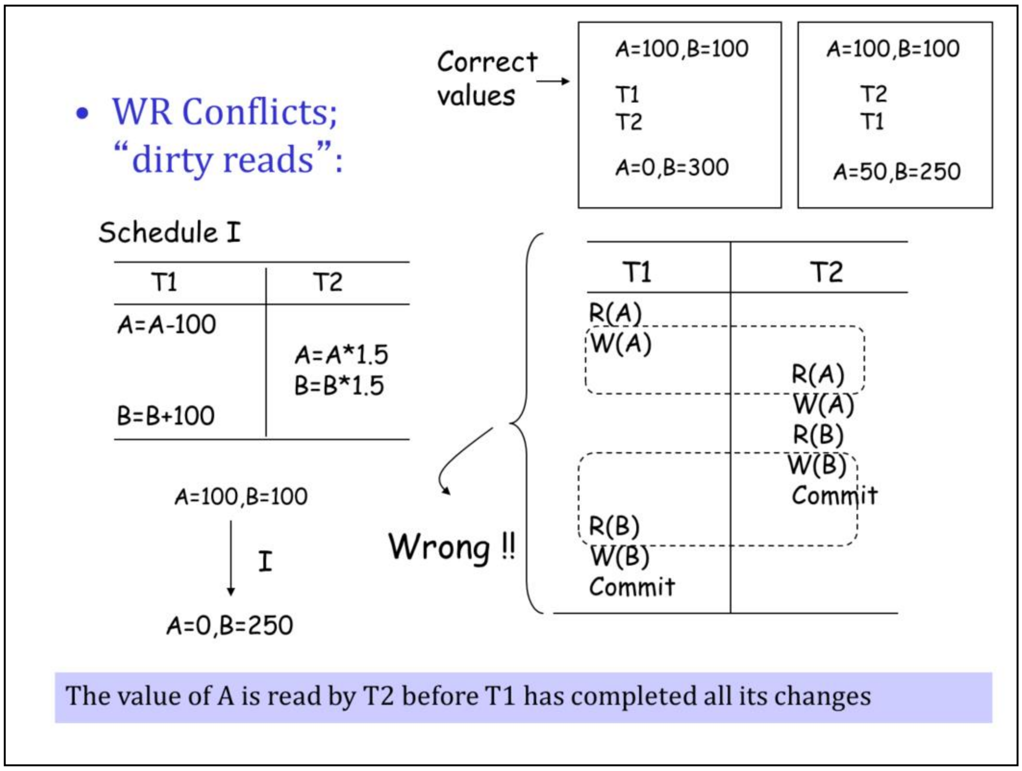
- RW Conflicts
- WR Conflicts
- WW Conflicts

$\rightarrow$ No abort in any transaction.

$\rightarrow$ Some abort in some transaction.

Two actions on the same data object conflict if at least one of them is a write.

All actions of aborted transactions are to be undone, and we can therefore imagine that they were never carried out to begin with.



- WR Conflicts; “dirty reads”:

Correct values →

A=100,B=100	A=100,B=100
T1 T2	T2 T1
A=0,B=300	A=50,B=250

Schedule I

T1	T2
A=A-100	
	A=A*1.5
	B=B*1.5
B=B+100	

$$A=100,B=100 \xrightarrow{I} A=0,B=250$$

Wrong !!

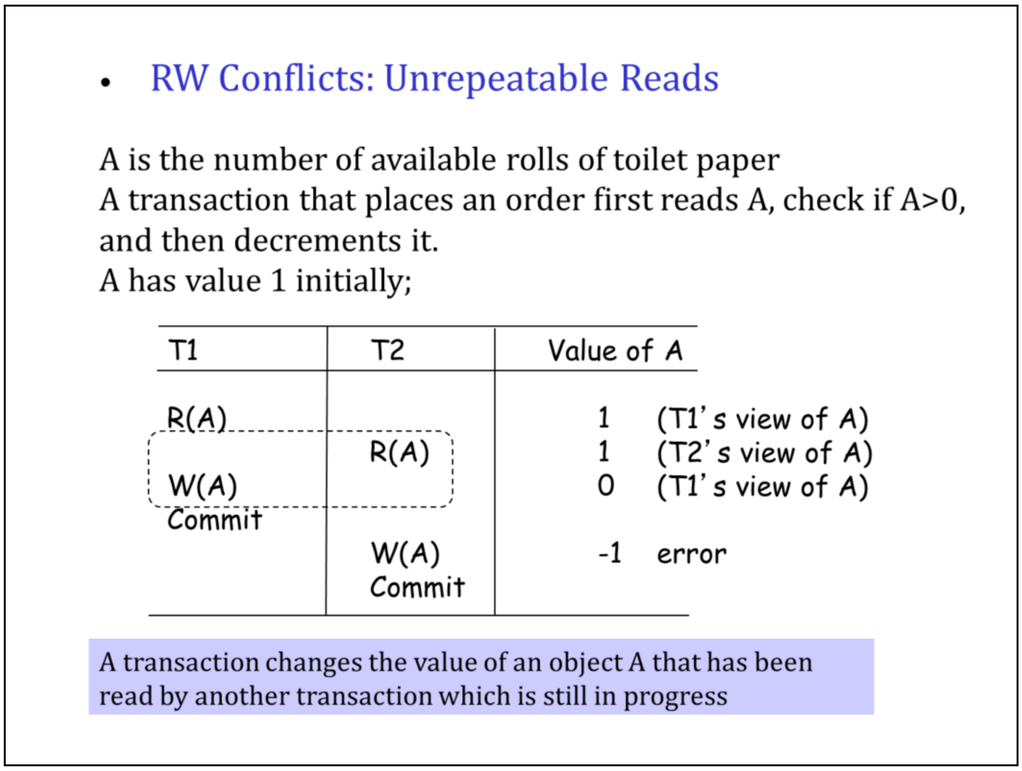
T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
R(B)	
W(B)	
Commit	

The value of A is read by T2 before T1 has completed all its changes

Reading uncommitted data

T1: transfer $100 from A to B

T2: interest is deposited into these two accounts.



• RW Conflicts: Unrepeatable Reads

A is the number of available rolls of toilet paper
A transaction that places an order first reads A, check if A>0, and then decrements it.
A has value 1 initially;

T1	T2	Value of A	
R(A)		1	(T1's view of A)
	R(A)	1	(T2's view of A)
W(A)		0	(T1's view of A)
Commit			
	W(A)	-1	error
	Commit		

A transaction changes the value of an object A that has been read by another transaction which is still in progress

- WW Conflicts: Overwriting uncommitted data

T1 sets A and B to 10; T2 sets A and B to 20.

-Consistency constraint: A and B must have the same value.

T1	T2	Value of A
W(A)		10
	W(A)	20
	W(B)	20
	Commit	10
W(B)		
Commit		

A =20 while B=10.

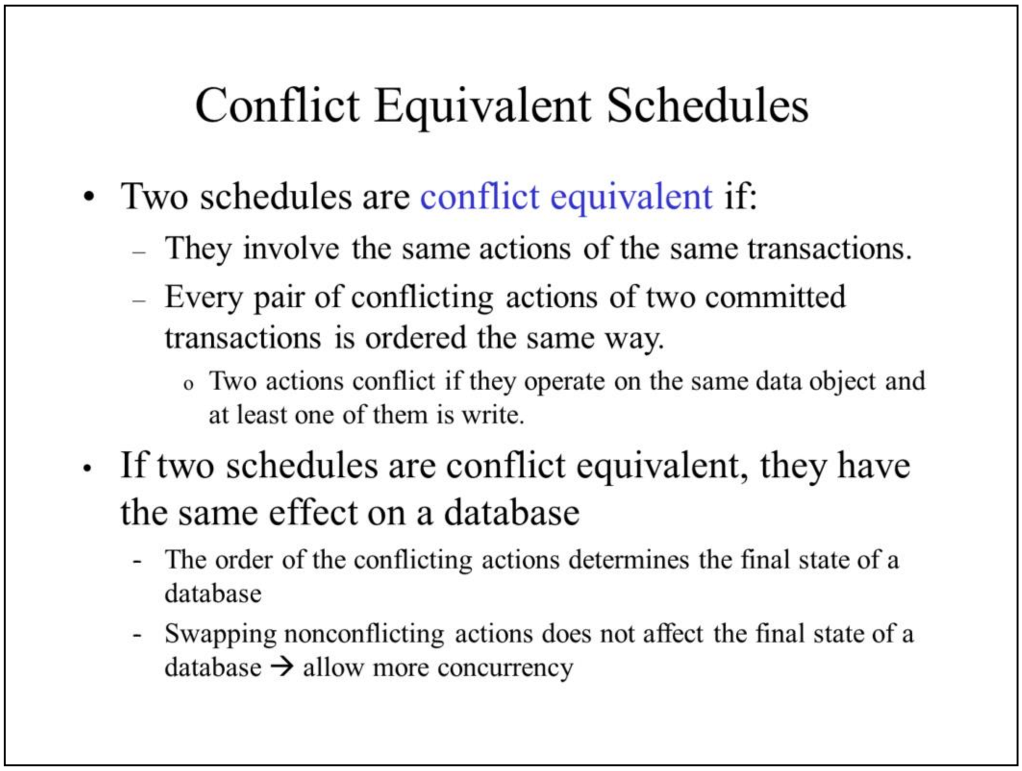
A transaction overwrites the value of an object A, which has already modified by another trasaction which is still in progress

Conflict Equivalent Schedules

- Two schedules are conflict equivalent if:
  - They involve the same actions of the same transactions.
  - Every pair of conflicting actions of two committed transactions is ordered the same way.
    - Two actions conflict if they operate on the same data object and at least one of them is write.

T1	T2
R1(A)	
W1(A)	
R1(B)	
W1(B)	
	R2(A)
	W2(A)

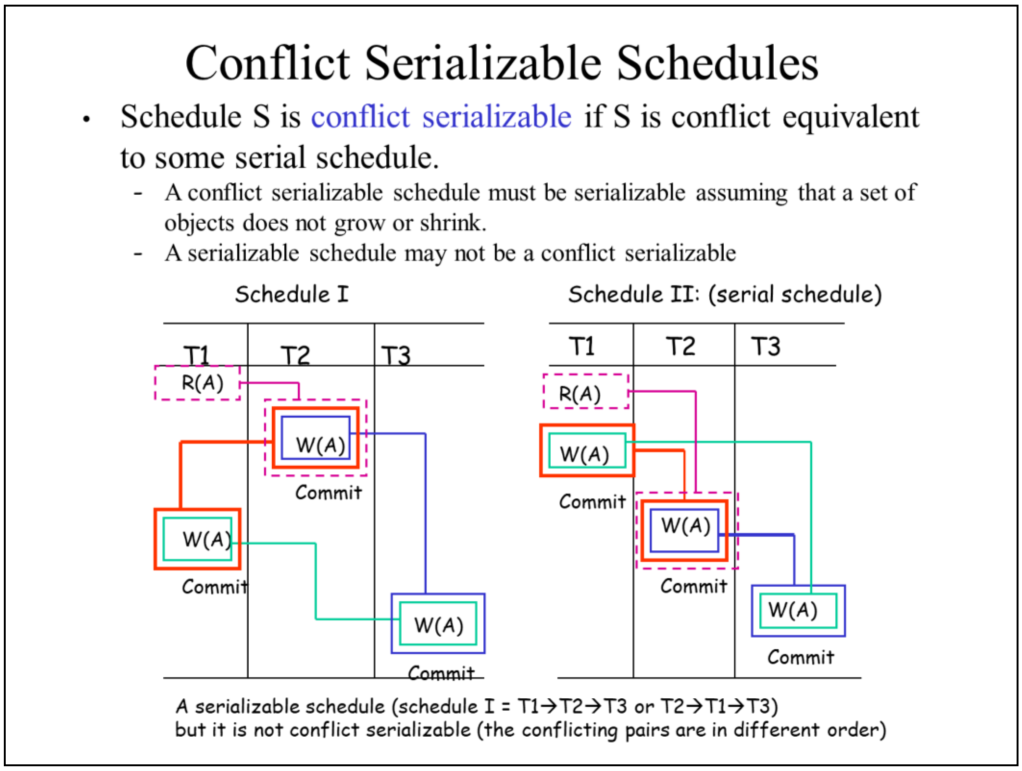
T1	T2
R1(A)	
W1(A)	
	R2(A)
	W2(A)
R1(B)	
W1(B)	



Conflict Equivalent Schedules

- Two schedules are conflict equivalent if:
  - They involve the same actions of the same transactions.
  - Every pair of conflicting actions of two committed transactions is ordered the same way.
    - Two actions conflict if they operate on the same data object and at least one of them is write.
- If two schedules are conflict equivalent, they have the same effect on a database
  - The order of the conflicting actions determines the final state of a database
  - Swapping nonconflicting actions does not affect the final state of a database → allow more concurrency

Two schedules are said to be conflict equivalent if they involve the same set of actions of the same transactions and they order every pair of conflicting actions of two committed transactions in the same way.



Conflict Serializable Schedules

- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule.
  - A conflict serializable schedule must be serializable assuming that a set of objects does not grow or shrink.
  - A serializable schedule may not be a conflict serializable

Schedule I
T1
T2
T3
R(A)
W(A)
Commit
W(A)
Commit
W(A)
Commit
Schedule II: (serial schedule)
T1
T2
T3
R(A)
W(A)
Commit
W(A)
Commit
W(A)
Commit

A serializable schedule (schedule I = T1→T2→T3 or T2→T1→T3)
but it is not conflict serializable (the conflicting pairs are in different order)

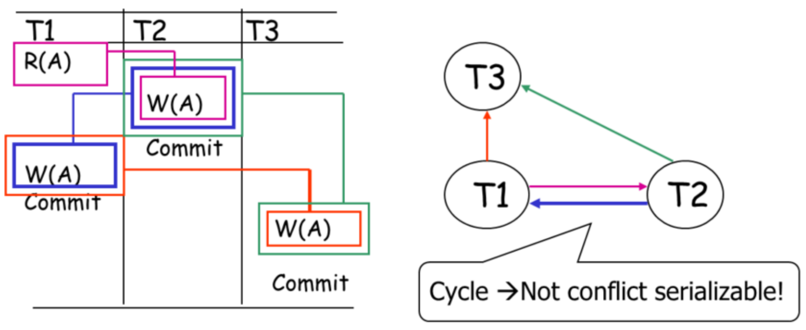
Schedule on the left is not conflict serializable. There are two pairs of conflicting actions: T1:R(A), T2: W(A) and T2:W(A), T1:W(A). They are not in the same order.

- To determine if a schedule does not result in anomaly, we just need to make sure it is conflict equivalent to some serial schedule
- How can we know if a schedule is conflict equivalent to some serial schedule?
  - Using precedence graph or serializability graph.

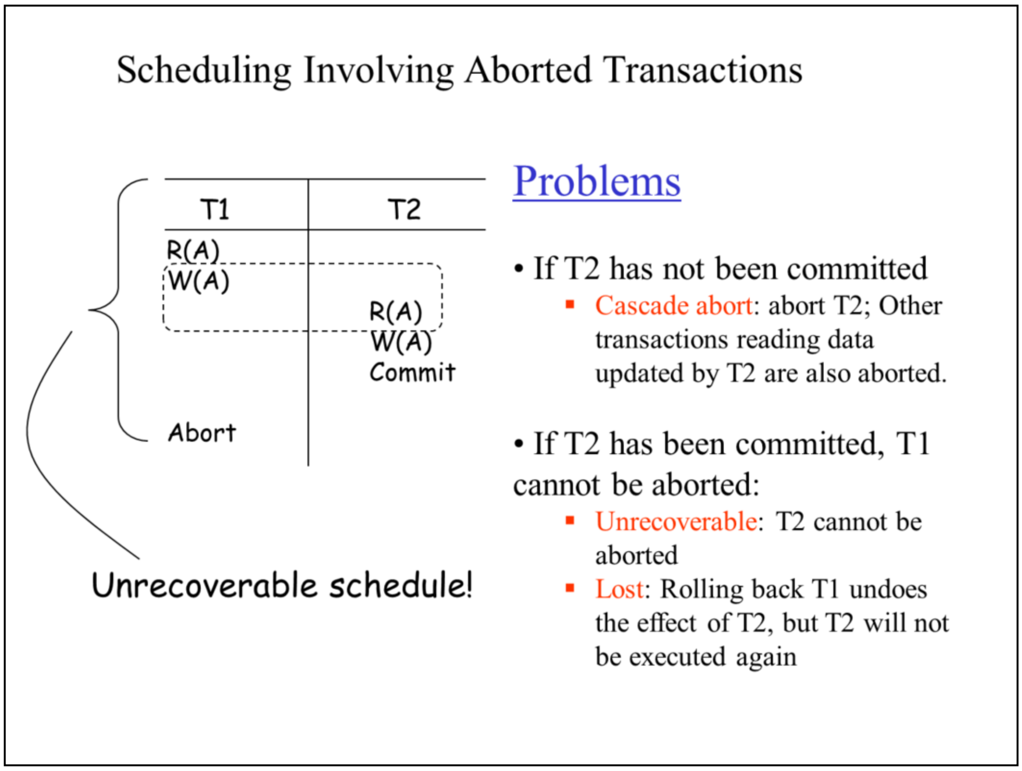
Precedence Graph (Serializability Graph)

The precedence graph for a schedule S contains:

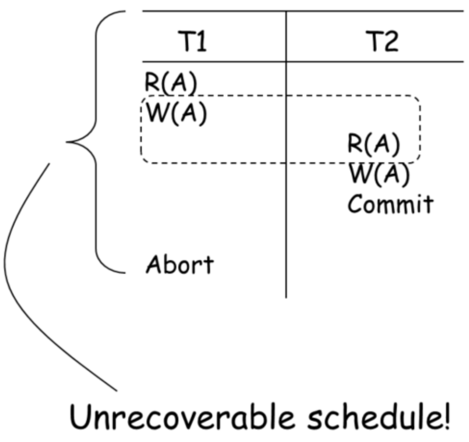
- A node for each committed transaction in S
- An arc from *Ti* to *Tj* if an action of *Ti* precedes and conflicts with one of *Tj*'s actions



Theorem: A schedule is conflict serializable if and only if its dependency graph is acyclic



Scheduling Involving Aborted Transactions



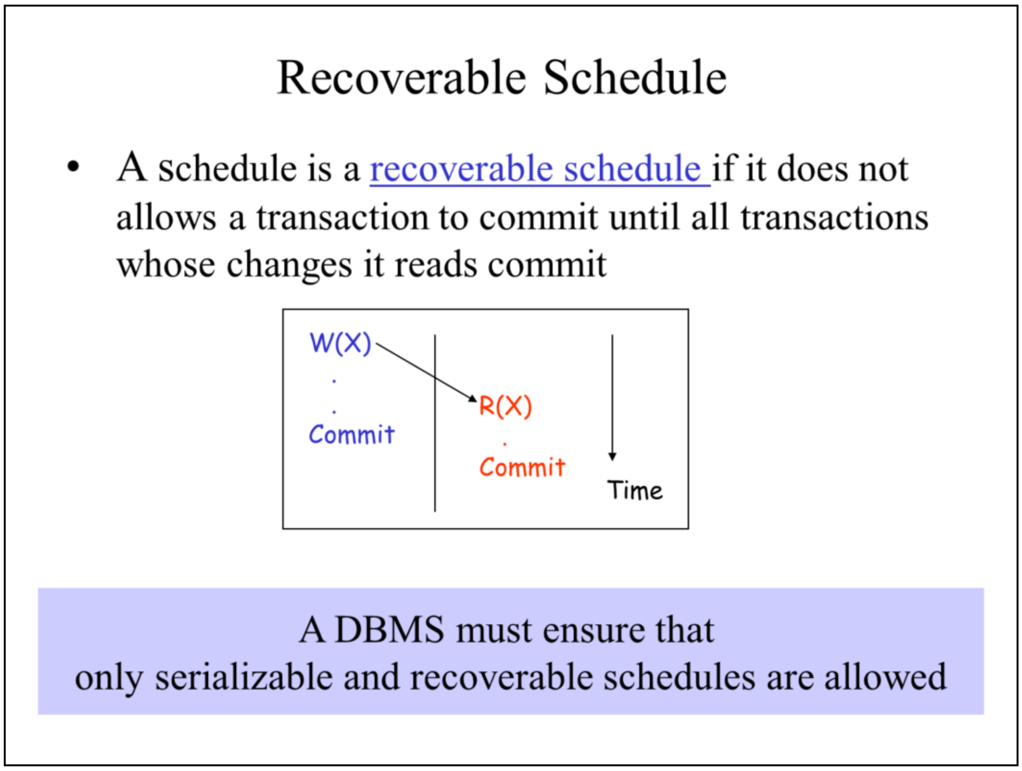
Problems

- If T2 has not been committed
  - Cascade abort: abort T2; Other transactions reading data updated by T2 are also aborted.
- If T2 has been committed, T1 cannot be aborted:
  - Unrecoverable: T2 cannot be aborted
  - Lost: Rolling back T1 undoes the effect of T2, but T2 will not be executed again

Suppose (1) an account transfer program T1 deducts $100 from account A. Then (2) an interest deposit program T2 reads the current values of account A and add 6% interest to it, then commits. And then (3) T1 is aborted. Now T2 has read a value for A that should never have been there. If T2 had not yet committed, we could deal with the situation by cascading the abort of T1 and also aborting T2. This process recursively aborts any transaction that read data written by T2 and so on.

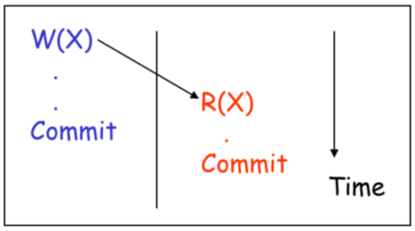
But T2 has already committed, so we can't undo its actions.

There is another potential problem in undoing the actions of a transaction. When T1 is aborted and its changes are undone, T2's changes are lost as well, even if T2 decides to commit.



Recoverable Schedule

- A schedule is a recoverable schedule if it does not allows a transaction to commit until all transactions whose changes it reads commit

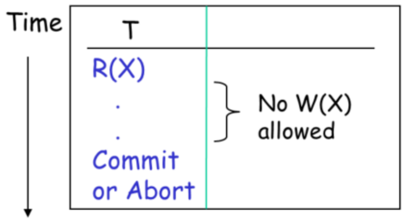


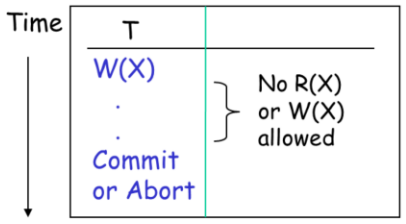
A DBMS must ensure that
only serializable and recoverable schedules are allowed

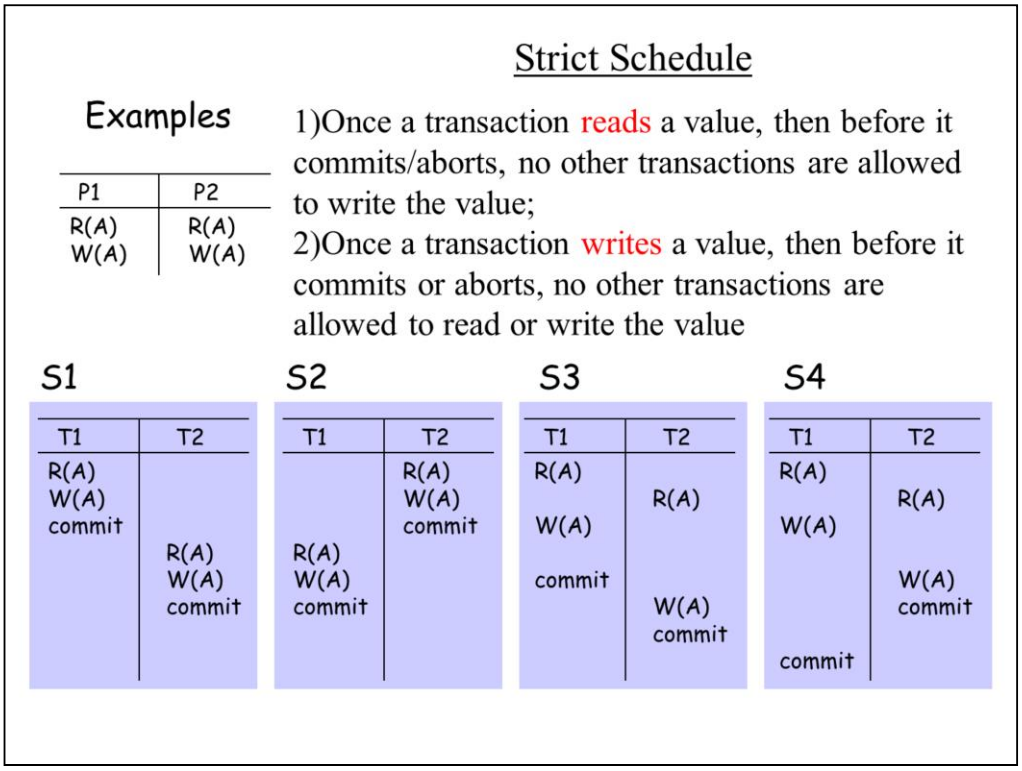
In a recoverable schedule, transactions commit only after (and if!) all transactions whose changes they read commit.

Strict Schedule

- A schedule is a strict schedule if it satisfies two conditions
  1) Once a transaction reads a value, then before it commits/aborts, no other transactions are allowed to write the value;
  2) Once a transaction writes a value, then before it commits or aborts, no other transactions are allowed to read or write the value







Strict Schedule

Examples

P1	P2
R(A)	R(A)
W(A)	W(A)

1)Once a transaction reads a value, then before it commits/aborts, no other transactions are allowed to write the value;
2)Once a transaction writes a value, then before it commits or aborts, no other transactions are allowed to read or write the value

S1

T1	T2
R(A)	
W(A)	
commit	
	R(A)
	W(A)
	commit

S2

T1	T2
	R(A)
	W(A)
	commit
R(A)	
W(A)	
commit	

S3

T1	T2
R(A)	
	R(A)
W(A)	
commit	
	W(A)
	commit

S4

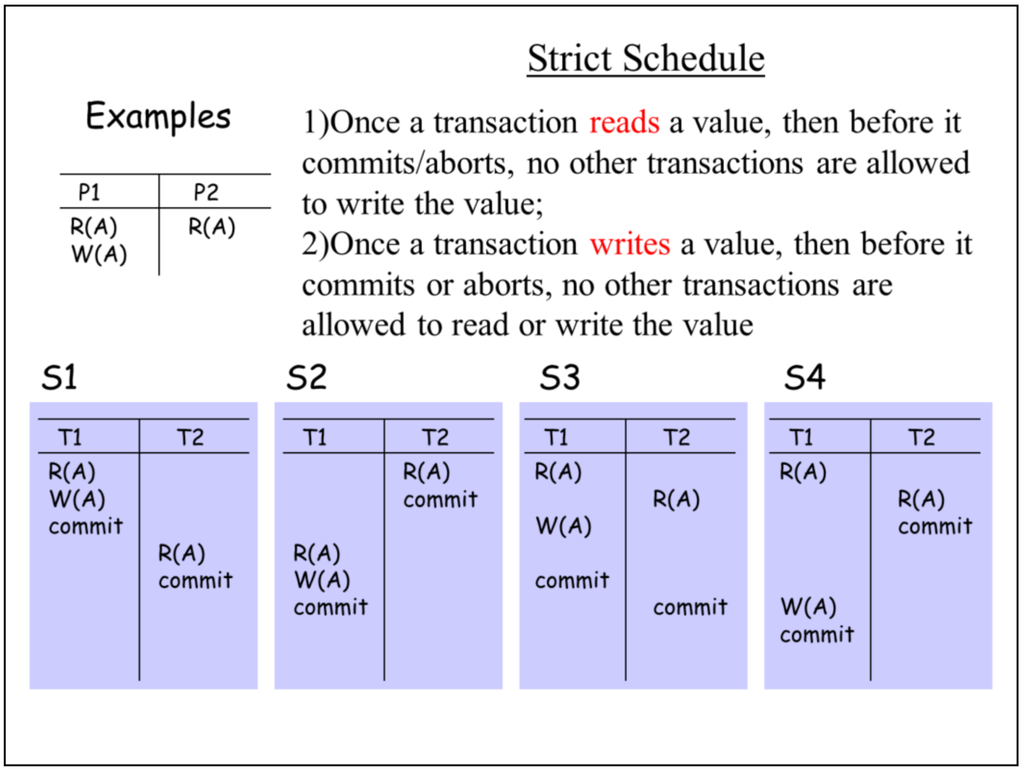
T1	T2
R(A)	
	R(A)
W(A)	
	W(A)
	commit
commit	

S1,2 are strict schedule.

S3,4 are not

S3: after T2 R(A), T1 W(A)

S4: after T2 R(A), T1 W(A); after T1 W(A), T2 W(A),



Strict Schedule

Examples

P1	P2
R(A)	R(A)
W(A)	

1)Once a transaction reads a value, then before it commits/aborts, no other transactions are allowed to write the value;
2)Once a transaction writes a value, then before it commits or aborts, no other transactions are allowed to read or write the value

S1

T1	T2
R(A)	
W(A)	
commit	
	R(A)
	commit

S2

T1	T2
	R(A)
	commit
R(A)	
W(A)	
commit	

S3

T1	T2
R(A)	
	R(A)
W(A)	
commit	
	commit

S4

T1	T2
R(A)	
	R(A)
	commit
W(A)	
commit	

S1,2,4 are strict schedule.

S3: after T2 R(A), T1 W(A)

A serial schedule must be a strict schedule, but not vice versa.

S1

T1	T2
R(A)	
W(A)	
commit	
	R(A)
	commit

S2

T1	T2
	R(A)
	commit
R(A)	
W(A)	
commit	

S4

T1	T2
R(A)	
	R(A)
	commit
W(A)	
commit	

Not a serial schedule!

A Strict schedule must be a serializable and recoverable schedule

1. It avoids RW, WR, WW conflicts, and
2. It does not require cascading aborts, and actions of an aborted transaction can be undone.

Quick Review

- Transaction: One execution of a program and seen as a sequence of read/write operations followed by either a commit or an abort
- Schedule: An ordered sequence of reads/writes
- Serial schedule: No interleaving of actions from different transactions
- Serializable schedule: The effect/result is the same as that from a serial schedule
- Conflict-equivalent schedules: Two schedules are conflict-equivalent if the order of their conflicting pairs
- Conflict-serializable schedule: The order of its conflicting pairs is the same as that of a serial schedule
- Recoverable schedule: A transaction is not allowed to commit until all transations whose changes it reads commit
- Strict schedule
  - When an object is read by a transaction, no write from another until it finishes
  - When an object is written by a transation, no read or write from another until it finishes