Transaction Management
&
Concurrency Control

Transactions

- A transaction is a collection of operations that form a single logical unit of work.

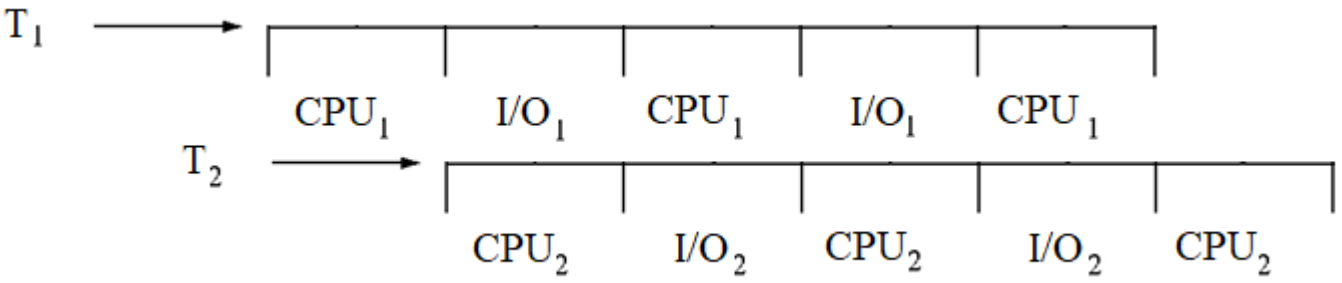
e.g. Someone wants to transfer INR 500 from one bank account (maintained by variable A) to another bank account (maintained by variable B)

T_1		
r(A)	// I/O burst operation	LOGICALLY A SINGLE OPERATION
A=A-500;	// CPU burst operation	
w(A)	// I/O burst operation	
r(B)	// I/O burst operation	
B=B+500;	// CPU burst operation	
w(B)	// I/O burst operation	

- At any instance of moment, there can be many such transactions T_1, T_2, T_3, …, T_n submitted by same or many users.
- If these transactions are executed sequentially:

  (a) waiting time will be more for some transactions than expected

  (b) Many CPU cycles will be idle when I/O operations are going on.
- So, it is advisable to interlace the concurrent transactions for better CPU utilization.



- Users submit transactions, and can think of each transaction as executing by itself.
- Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.

- However simultaneously executing transactions T_1, T_2, T_3, …, T_n may act on same variables residing in database, and therefore their order of execution (which adheres to business logic) must be satisfied by such concurrent execution.

- Hence it is termed as '**controlled concurrency**'.

- Let us assume T_1, T_2, T_3, T_4, T_5, T_6 are submitted to DBMS simultaneously for execution, with constraint that it should execute in the following order: T_2, T_1, T_5, T_6, T_3, T_4. Hence, these transactions may be executed concurrently if possible, but should yield the same result as it would have been generated had they been executed sequentially in the given order.

Properties of transaction

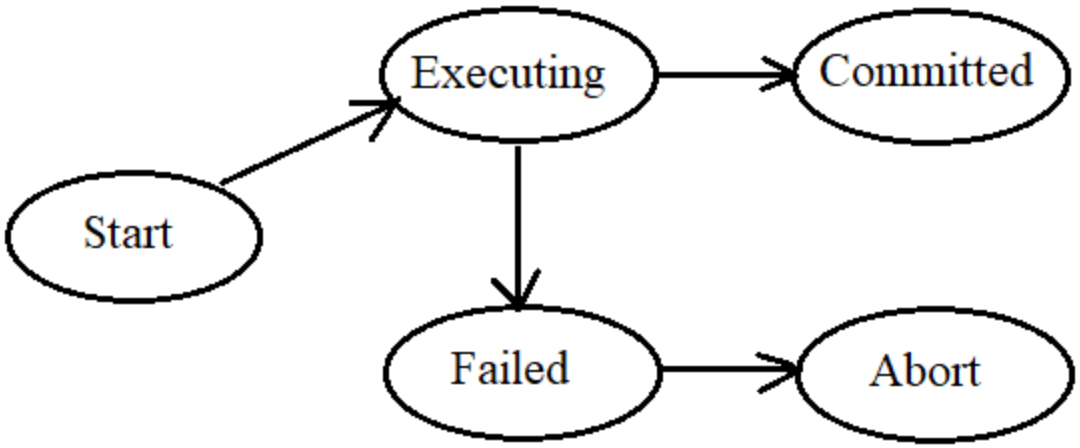
- **Atomicity:** all or nothing
- **Consistency:** takes DB from one consistent state to another, without necessarily preserving interim consistency.
- **Isolation:** Concealed from each other. The changes of a variable by one transaction are not realizable by another transaction until written back.
- **Durability:** Once a transaction commits, its updates survive in the database.

Transactions

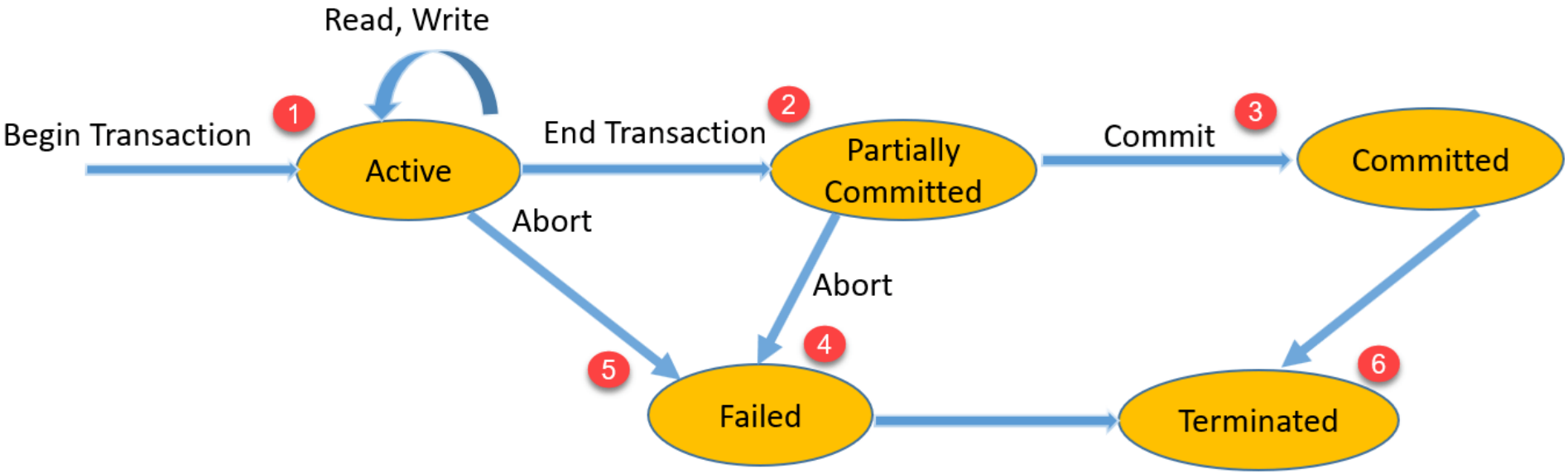
- A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.

- **A *transaction* is the DBMS's abstract view of a user program: a sequence of reads and writes.**

Lifecycle of a transaction



Lifecycle of a transaction



Concurrency of transactions

T_1 (to withdraw INR 50 from account A)	T_2 (to withdraw INR 100 from account A)
r(A)	
	r(A)
A=A-50;	
	A=A-100;
	w(A)
w(A)	

Let us assume initial value of A is 1000 before submission of both transactions.

Desired answer after T_1 and T_2 executed:
A=850

After execution of T_1 and T_2 (be in any serial order), the answer should not be what is produced by this concurrent execution!!!

Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts: dirty read problem):

```
T1:     R(A), W(A),                    R(B), W(B), Fail, Rollback
T2:                   R(A), W(A), C
```

- Unrepeatable Reads or non-repeatable read problem (RW Conflicts):

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

- Overwriting Uncommitted Data (WW Conflicts: lost update problem):

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

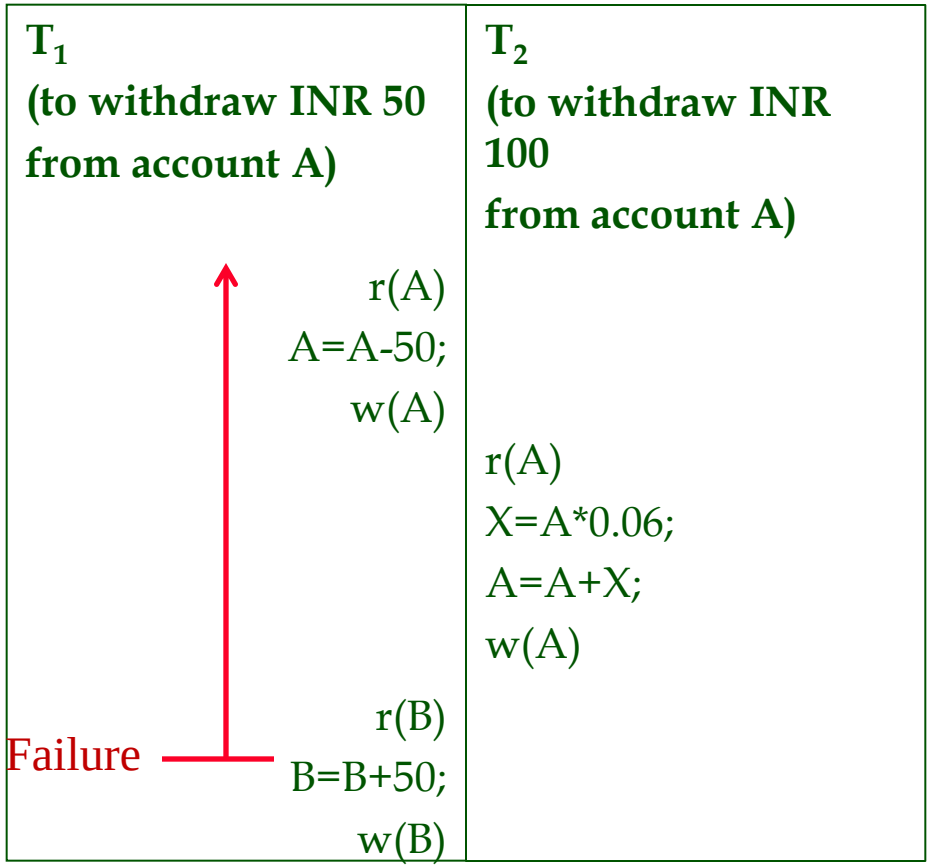
Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts: dirty read problem):

```
T1:        R(A), W(A),                    R(B), W(B), Fail, Rollback
T2:                      R(A), W(A), C
```

- Reading uncommitted data

Dirty Read: Example



r(A) reads the data modified by T1 through w(A) but T1 has not committed yet (value of A read by T2 is not yet saved permanently by T1).

As T1 fails, all operations rollback and value of A is not updated in database, but T2 reads and works on it!!

Anomalies with Interleaved Execution

- Unrepeatable Reads or non-repeatable read problem (RW Conflicts):

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

- When a transaction reads a data item twice and meanwhile another transaction operates on the same data, leading to reading two different values of same data item by the first transaction.

Unrepeatable read: Example

T_1 (to withdraw INR 50 from account A)	T_2 (to withdraw INR 100 from account A)
r(A)	
	r(A)
	A=A-100;
	w(A)
r(A)	
A=A-50;	
w(A)	

Two subsequent reads of item A in T1 finds two different values!!!

Anomalies with Interleaved Execution

- Overwriting Uncommitted Data (WW Conflicts: lost update problem):

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

- The second WRITE overwrites the first WRITE.
- If there are two write operation on same variable from two different transactions, and there is no read operation in between, it will cause loss of result of first WRITE operation, termed as lost update.

Lost update : Example

T_1	T_2
r(A)	
A=A-50;	
	r(A)
	X=A*0.06;
	A=A+X;
w(A)	
	w(A)
r(B)	
B=B+50;	
w(B)	

The effect of value update of A by w(A) in T1 is lost due to w(A) in T2!!!

- *Incorrect summary problem* is discussed in some books, but that is a kind of the above discussed problems.

Scheduling Transactions

- *Serial schedule:* Schedule that does not interleave the actions of different transactions.

- *Equivalent schedules*: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- *Non-serial schedule:* Schedule that interleaves the actions of different transactions.

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

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

Scheduling Transactions

- A schedule is a defined order of execution of the transactions.
- Let us assume k concurrent transactions. T_1 has n_1 operations, T_1 has n_1 operations, …, T_k has n_k operations.
- Number of possible ways of serial execution (i.e. number of serial schedules) is: k!
- One or few of these serial executions are actually intended by business logic.

Scheduling Transactions

- Let us assume k concurrent transactions. T_1 has n_1 operations, T_2 has n_2 operations, …, T_k has n_k operations.

- Number of possible ways of interlaced execution (i.e. number of non-serial schedules) is:

$$\frac{(n_1 + n_2 + \ldots + nk)_!}{n_1!\, n_2!\, \ldots\, n_k!} - k!$$

- Few of these interlaced executions will actually produce the same output as required by the business logic (equivalent to some decided serial execution).

Types of Schedules

- Serial schedule
- Complete schedule
- Recoverable and non-recoverable schedule
- Cascading and Cascadeless schedule
- Strict schedule
- Equivalent and non-equivalent schedule

Serial Schedules

- Serial schedule means no interleaving.
- Ensures to take database to consistent state.

T_1	T_2
r(A)	
A=A-50;	
w(A)	
	r(A)
	A=A+500;
	w(A)

Non-serial Schedules

- Non-serial schedule means there will be interleaving.
- May take database to consistent state.
- May generate unwanted result.

T_1	T_2
r(A)	
A=A-50;	
	r(A)
	A=A+500;
	w(A)
w(A)	

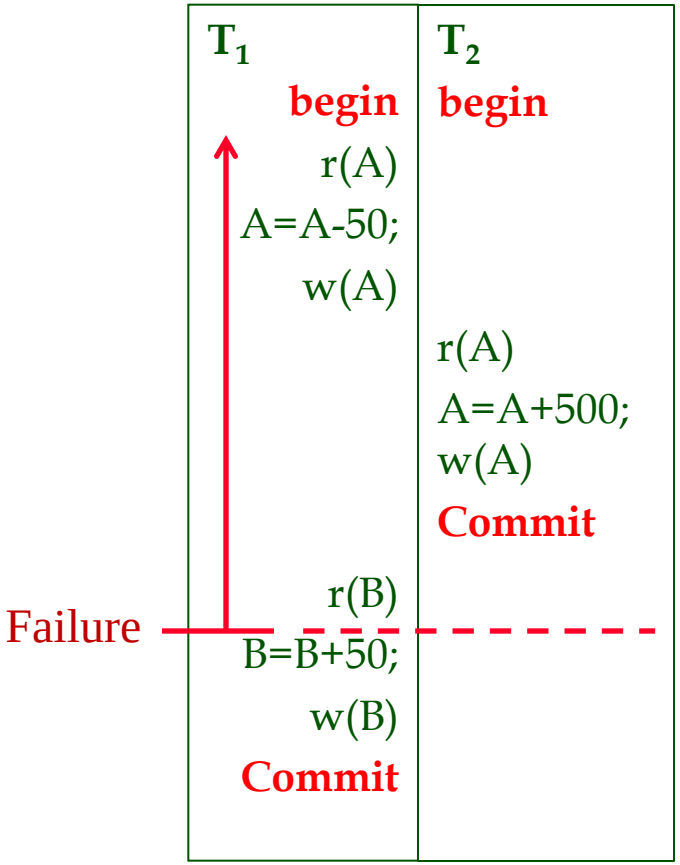
Complete Schedules

- Commit or Abort statement is used at the end of the transaction to permanently record the values of the data items in database.
- Unless particularly needed, we usually do not explicitly write it in our notation, but we should!

T_1	T_2
begin	**begin**
r(A)	
A=A-50;	
	r(A)
	A=A+500;
	w(A)
w(A)	**abort**
commit	

Recoverable Schedules

- T2 has read a data written interim by T1.
- T2 should be rolled back due to rolling back of T1, as T2 has read a data written by T1.
- But T2 cannot be rolled back after T1 fails at it has already committed.
- This schedule is not recoverable as one participating transaction cannot be rolled back if one of the transactions abort.
- If the transaction reading data from another transaction **commits ONLY after the commit** of the transaction from which it reads data, the schedule can be recoverable.



Cascadeless Schedules

- A bit stricter condition!
- One transaction reads only those data ALREADY committed by another transaction.
- No need of cascading of rollback in this case.
- Cascadeless -> Recoverable

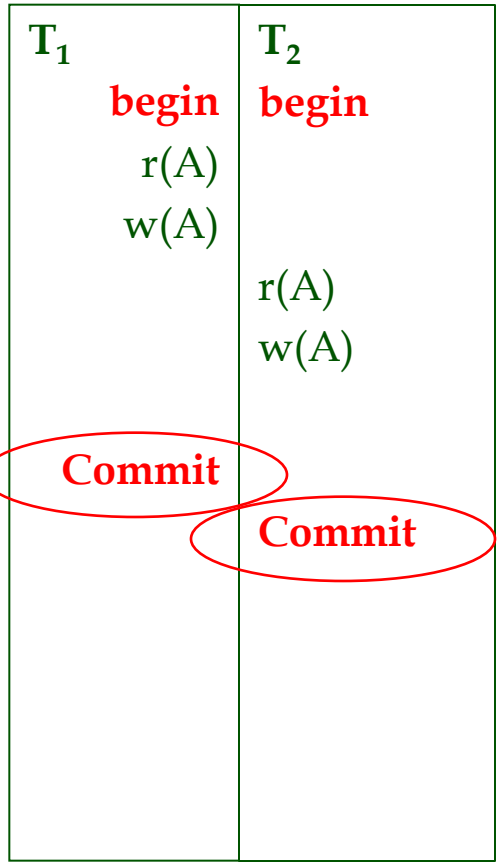
T_1	T_2	T_3
begin	**begin**	**begin**
r(A)		
w(A)		
Commit		
	r(A)	
	w(A)	
	Commit	
		r(A)
		w(A)
		Commit

Strict Schedules

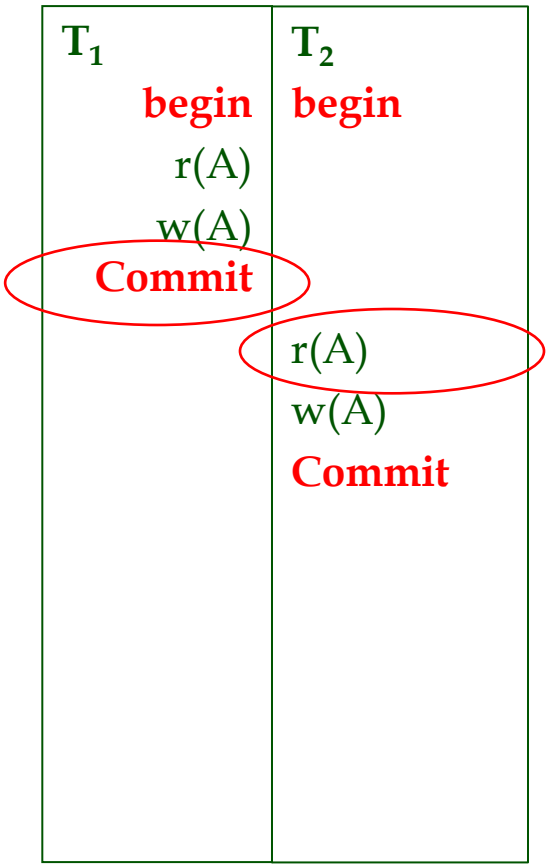
- A further bit stricter condition!
- One transaction reads or writes only those data ALREADY committed by another transaction.
- No need of cascading of rollback in this case too!
- Strict -> Cascadeless

T_1	T_2	T_3
begin	**begin**	**begin**
r(A)		
w(A)		
Commit		
	r(A)	
	w(A)	
	Commit	
		r(A)
		w(A)
		Commit

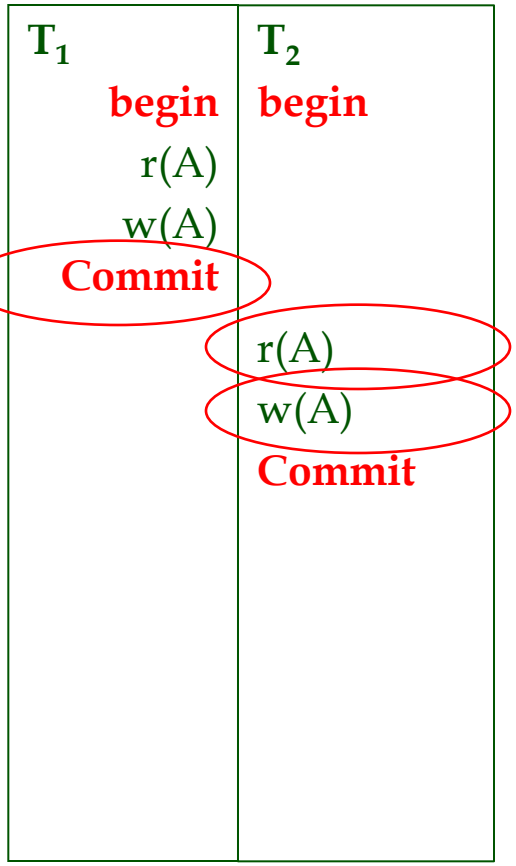
Recoverable

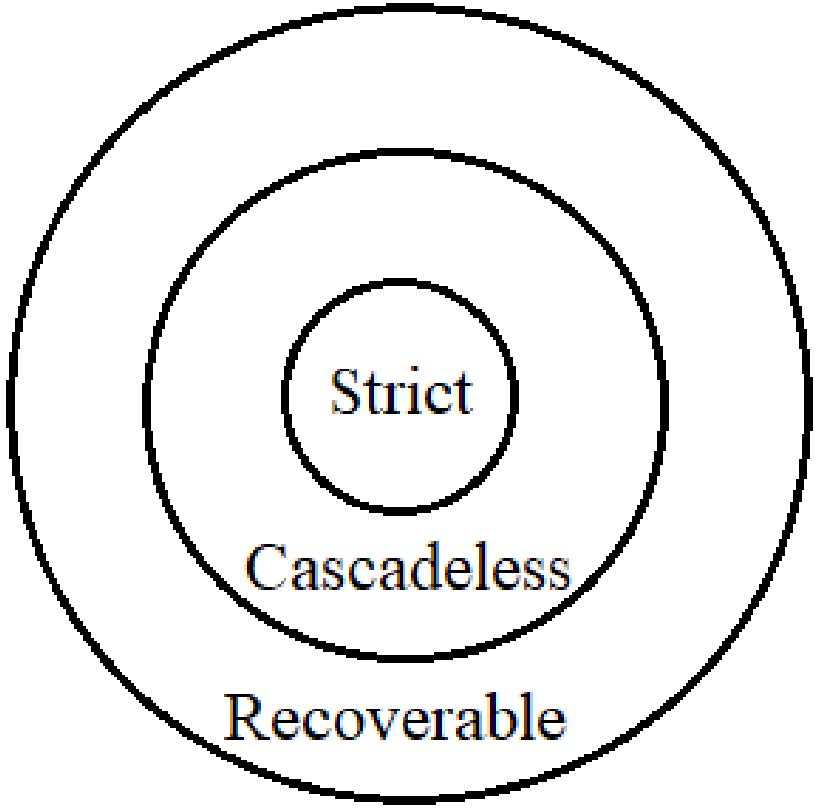


Cascadeless



Strict




Strict
Cascadeless
Recoverable

Equivalent Schedules

- Two schedules can be compared and they can be said 'equivalent' if they follow certain rules.
  - Result equivalent
  - Conflict equivalent

Result Equivalent Schedules

- Two schedules are said to be result equivalent if they produce same database state for a given database state as input.
- The two schedules are result equivalent if the initial state is A=100.

Initially A = 100

S_1	S_2
r(A)	r(A)
A=A+10;	A=A*1.1;
w(A)	w(A)

Result Equivalent Schedules

Initially X = 2, Y=5

S_1

T_1	T_2
r(X)	
X=X+5;	
w(X)	
	r(X)
	X=X*3;
	w(X)
r(Y)	
Y=Y+5;	
w(Y)	

Finally X = 21, Y=10

S_2

T_1	T_2
	r(X)
	X=X*3;
	w(X)
r(X)	
X=X+5;	
w(X)	
r(Y)	
Y=Y+5;	
w(Y)	

Finally X = 11, Y=10

S_1 and S_2 are not result equivalent with respect to the given input state.

Conflicts

- Conflicting actions are any combination of the following:
- r(A) by T_i and w(A) by T_j
- w(A) by T_i and r(A) by T_j
- w(A) by T_i and w(A) by T_j

- Non-conflicting actions are:
  - r(A) by T_i and r(A) by T_j
  - Operation on two different data items

T_i	T_j
r(A)	r(A)
A=A+10;	A=A*1.1;
w(A)	w(A)

Conflict Equivalent Schedules

For two schedules S_1 and S_2 to be conflict equivalent,
they should have **same conflicts** and in **same order**.

Conflict Equivalent Schedules

S_1

T_1	T_2
r(A)	
w(A)	
	r(A)
	w(A)
r(B)	
w(B)	

S_2

T_1	T_2
r(A)	
w(A)	
	r(A)
r(B)	
	w(A)
w(B)	

Are S_1 and S_2 conflict equivalent?

Conflict Equivalent Schedules

S_1

T_1	T_2
r(A)	
w(A)	
	r(A)
	w(A)
r(B)	
w(B)	

Conflicts from S_1 are:

$r(A)|_{T1}$ - $w(A)|_{T2}$

$w(A)|_{T1}$ - $r(A)|_{T2}$

$w(A)|_{T1}$ - $w(A)|_{T2}$

Conflict Equivalent Schedules

S_2

T_1	T_2
r(A)	
w(A)	
	r(A)
r(B)	
	w(A)
w(B)	

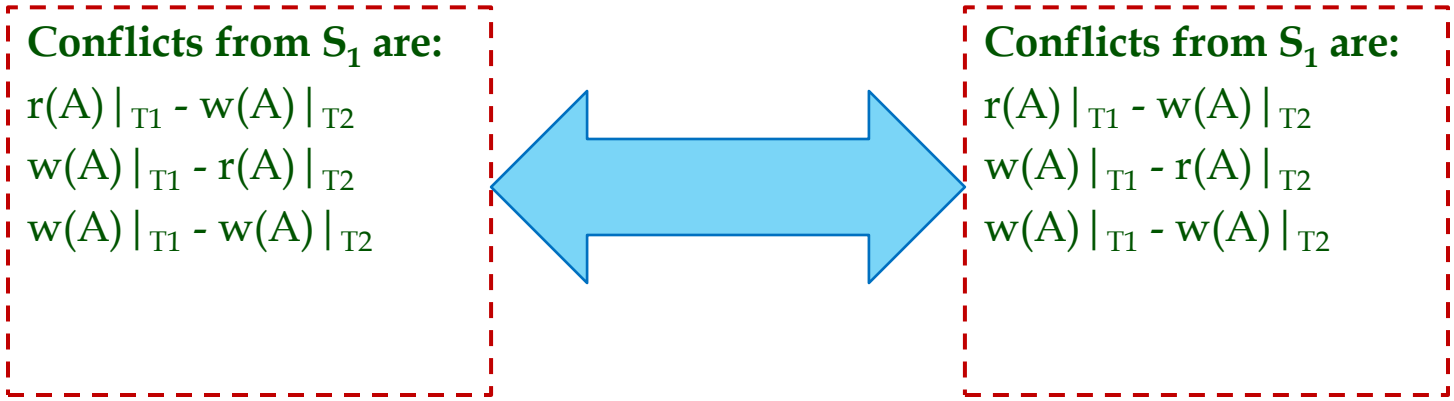
Conflicts from S_1 are:

$r(A)|_{T1}$ - $w(A)|_{T2}$

$w(A)|_{T1}$ - $r(A)|_{T2}$

$w(A)|_{T1}$ - $w(A)|_{T2}$

Conflict Equivalent Schedules



As conflicts in S_1 and S_2 are same and in same order, they are conflict equivalent.

Conflict Equivalent Schedules

Are S_1 and S_2 conflict equivalent?
Transaction numbers are written as subscripts

Is $S_1 \overset{C}{=} S_2$?

S_1: $R_1(A)$, $R_2(B)$,$W_1(A)$, $W_2(B)$
S_2: $R_2(B)$, $R_1(A)$, $W_2(B)$, $W_1(A)$

Conflict Equivalent Schedules

Are S_1 and S_2 conflict equivalent?
Transaction numbers are written as subscripts

Is $S_1 \overset{C}{=} S_2$? YES

S_1: $R_1(A)$, $R_2(B)$,$W_1(A)$, $W_2(B)$
S_2: $R_2(B)$, $R_1(A)$, $W_2(B)$, $W_1(A)$

Conflict Equivalent Schedules

Are S_1 and S_2 conflict equivalent?
Transaction numbers are written as subscripts

Is $S_1 \overset{C}{=} S_2$?

S_1: $R_1(A)$, $W_1(A)$, $R_2(B)$, $W_2(B)$, $R_1(B)$
S_2: $R_1(A)$, $W_1(A)$, $R_1(B)$, $R_2(B)$, $W_2(B)$

Conflict Equivalent Schedules

Are S_1 and S_2 conflict equivalent?
Transaction numbers are written as subscripts

Is $S_1 \overset{C}{=} S_2$? NO

S_1: $R_1(A)$, $W_1(A)$, $R_2(B)$, $W_2(B)$, $R_1(B)$
S_2: $R_1(A)$, $W_1(A)$, $R_1(B)$, $R_2(B)$, $W_2(B)$

Conflict Equivalent Schedules

Are S_1 and S_2 conflict equivalent?
Transaction numbers are written as subscripts

Is $S_1 \overset{C}{=} S_2$?

S_1: $R_2(A)$, $R_1(A)$, $W_2(B)$, $W_1(B)$
S_2: $R_1(A)$, $R_1(A)$, $W_2(B)$, $W_1(B)$

Conflict Equivalent Schedules

Are S_1 and S_2 conflict equivalent?
Transaction numbers are written as subscripts

Is $S_1 \overset{C}{=} S_2$? YES

S_1: $R_2(A)$, $R_1(A)$, $W_2(B)$, $W_1(B)$
S_2: $R_1(A)$, $R_1(A)$, $W_2(B)$, $W_1(B)$

Conflict Serializable Schedules

If a non-serial schedule S_1 is conflict equivalent to a serial schedule S_2, then S1 is termed as *conflict serializable*.

Conflict Serializable Schedules

- **To summarize what we learnt till now,**
- Two schedules are conflict equivalent if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way
- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

Testing conflict serializability

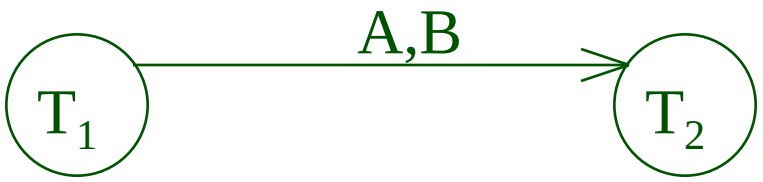
T1:	R(A),	W(A),												R(B),	W(B),				
T2:						R(A),	W(A),				R(B),		W(B),						W(B)

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

- **Step 1:**
- Start drawing a graph, which is called as precedence graph. Create a node T_i for each participating transaction in the schedule S under test.

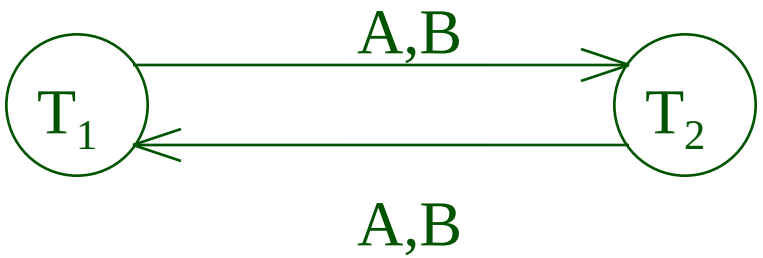
T1:	R(A),	W(A),						R(B),	W(B),		
T2:			R(A),	W(A),	R(B),	W(B),				W(B)	

- **Step 2:**
- For each case where T_j executes R(X) after T_i executing W(X), we draw an edge from T_i to T_j.



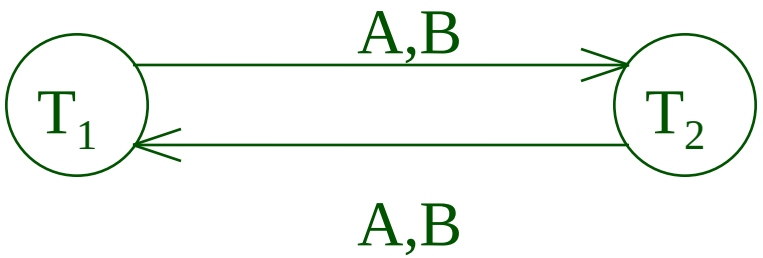
T1:	R(A),	W(A),						R(B),	W(B),		
T2:			R(A),	W(A),	R(B),	W(B),				W(B)	

- **Step 3:**
- For each case where T_j executes W(X) after T_i executing R(X), we draw an edge from T_i to T_j.



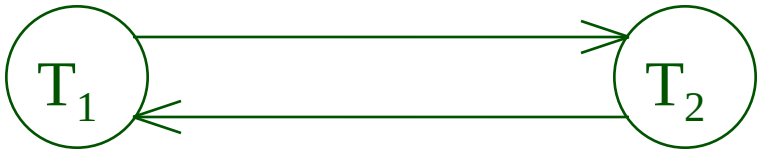
T1:	R(A),	W(A),					R(B),	W(B),	
T2:			R(A),	W(A),	R(B),	W(B),			W(B)

- **Step 4:**
- For each case where T_j executes W(X) after T_i executing W(X), we draw an edge from T_i to T_j.



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

- **Step 5:**
- Cycle in precedence graph -> NOT conflict serializable
- NO cycle in precedence graph -> conflict serializable



NOT conflict serializable