Transactions – Concurrent Executions

Dr. L.M. Jenila Livingston
VIT Chennai

Transactions

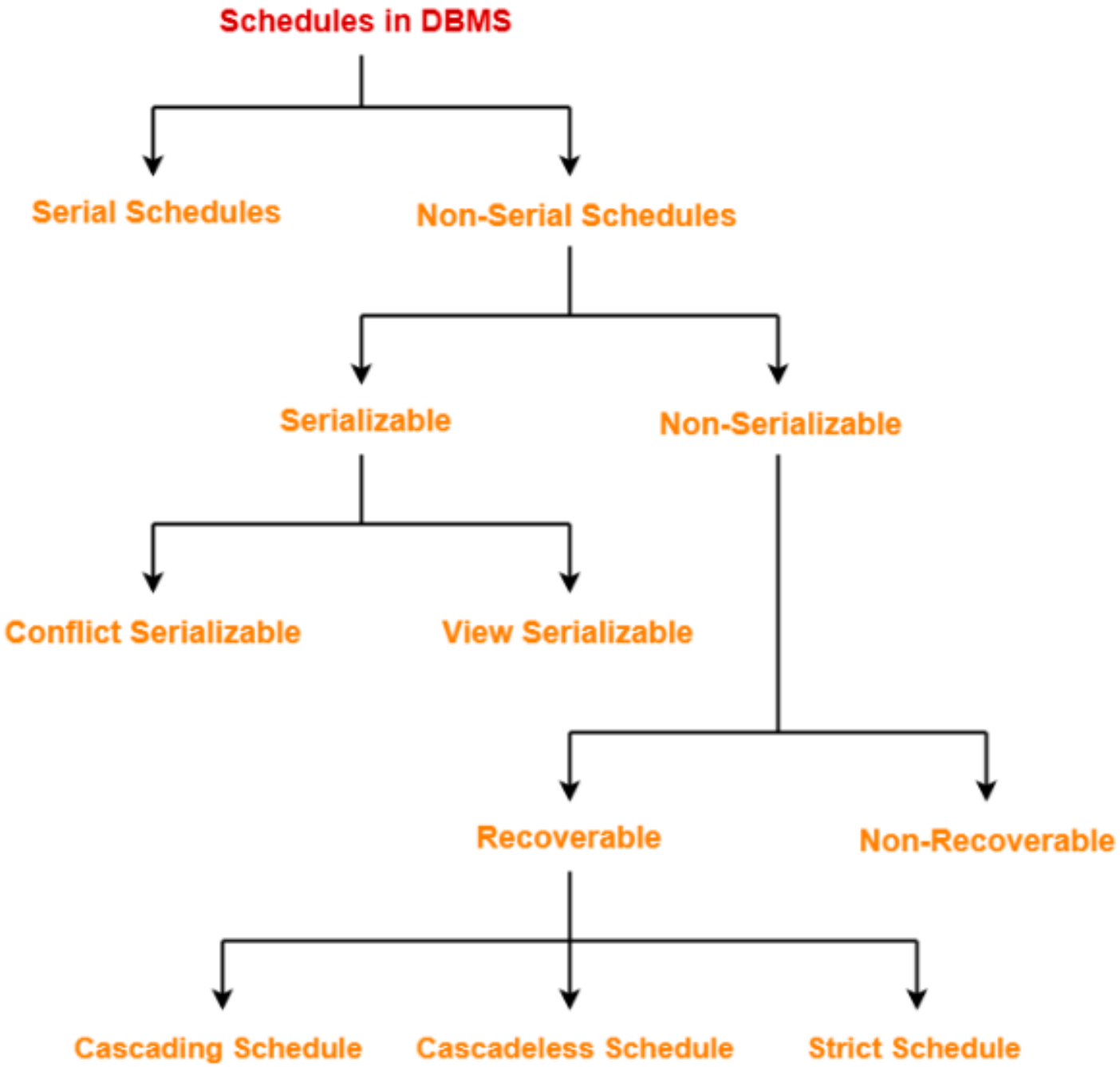
- Concurrent Executions
- Serializability
- Conflict Serializable
- Testing for Serializability
- View Serializable
- Recoverability
- Transaction Definition in SQL
- Recoverable Schedules

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

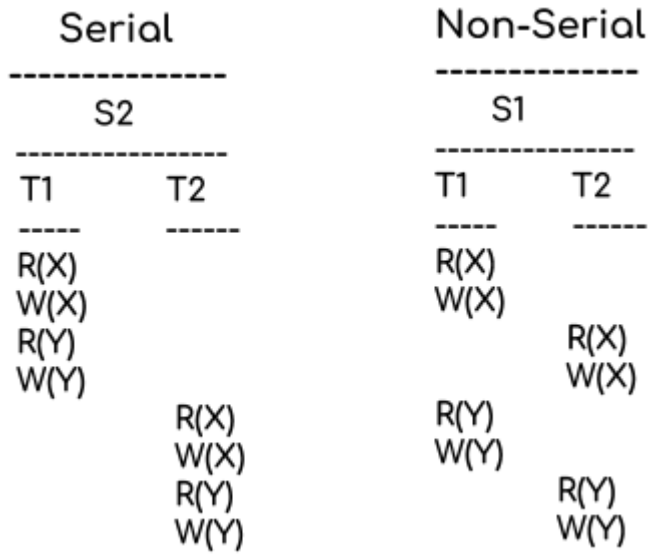
Schedules

- **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


Schedules in DBMS
Serial Schedules
Non-Serial Schedules
Serializable
Non-Serializable
Conflict Serializable
View Serializable
Recoverable
Non-Recoverable
Cascading Schedule
Cascadeless Schedule
Strict Schedule

Serial Vs Serializable

Serial Schedules	Serializable Schedules
No concurrency is allowed. Thus, all the transactions necessarily execute serially one after the other.	**Concurrency** is allowed. Thus, multiple transactions can execute concurrently.
Serial schedules lead to less resource utilization and CPU throughput.	Serializable schedules improve both resource utilization and CPU throughput.
Serial Schedules are less efficient as compared to serializable schedules.	Serializable Schedules are always better than serial schedules.



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 :

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

Let A=150, B=50
A=150-50=**100**
B=50+50=**100**

temp=100*0.1=10
A=100-10=**90**
B=100+10=**110**

Sum(A+B)=200

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	

Let A=150, B=50
Temp=150*0.1=15
A=150-15=**135**
B=50+15=**65**

A=135-50=**85**
B=65+50=**115**

Sum(A+B)=200

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.

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

Let A=150, B=50
A=150-50=100

temp=100*0.1=10
A=100-10=**90**

B=50+50=100

B=100+10=**110**

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

Sum(A+B)=200

Schedule 4

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

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Let A=150, B=50
A=150-50

temp=150*0.1=15
A=150-15=**135**

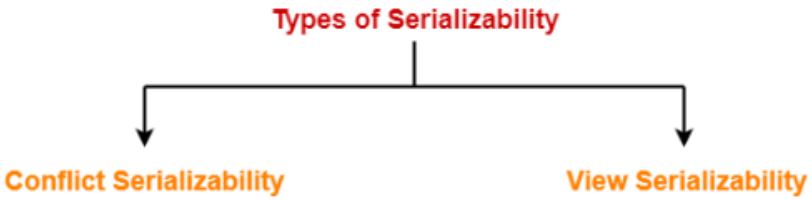
A=100
B=50+50=**100**

B=50+15=**65**

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.

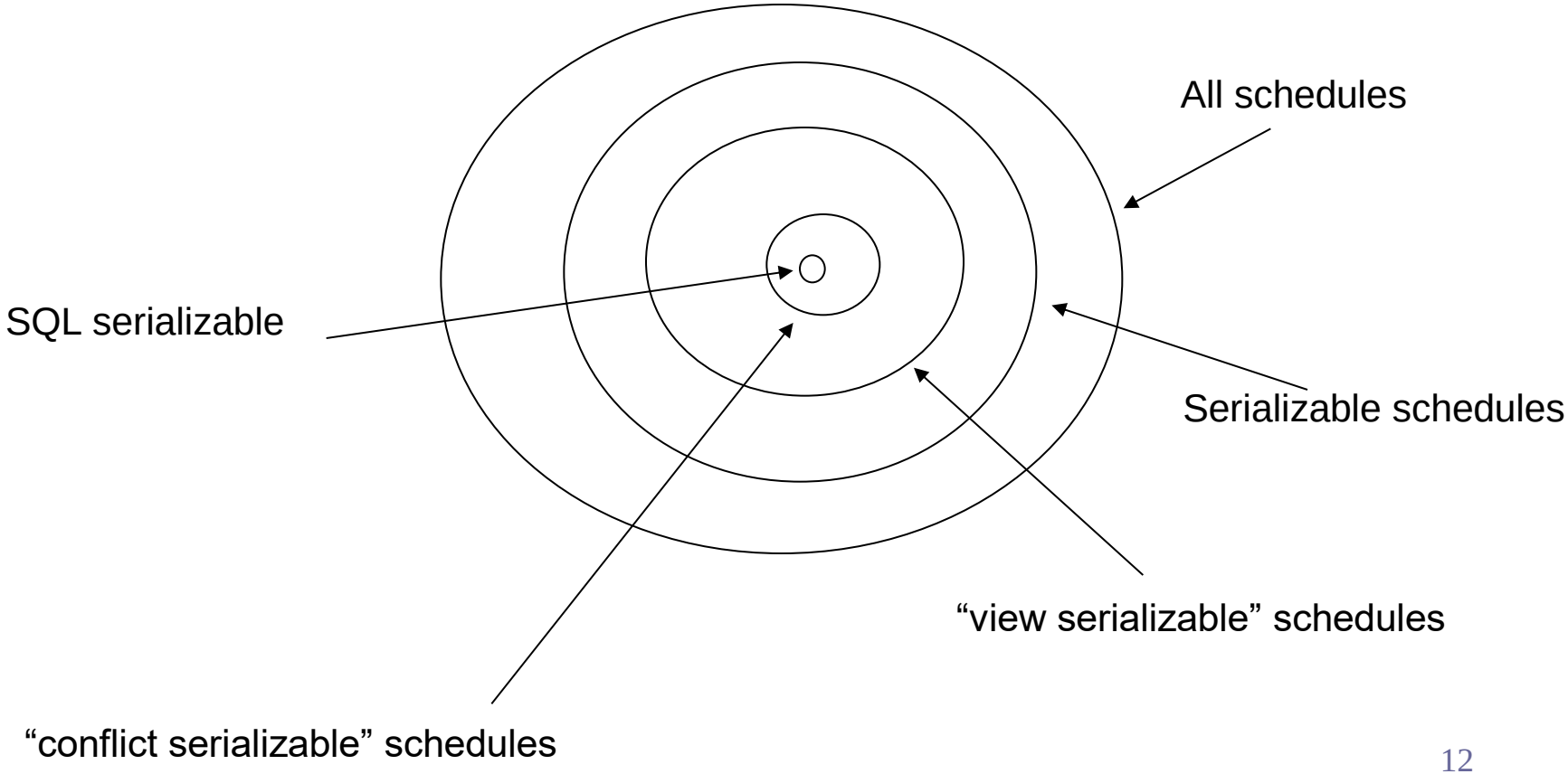
- Different forms of schedule equivalence give rise to the notions of:
  1. **view serializability**
  2. **conflict serializability**



Serializability

Serializable: A schedule is serializable if its effects on the database are the equivalent to some serial schedule.

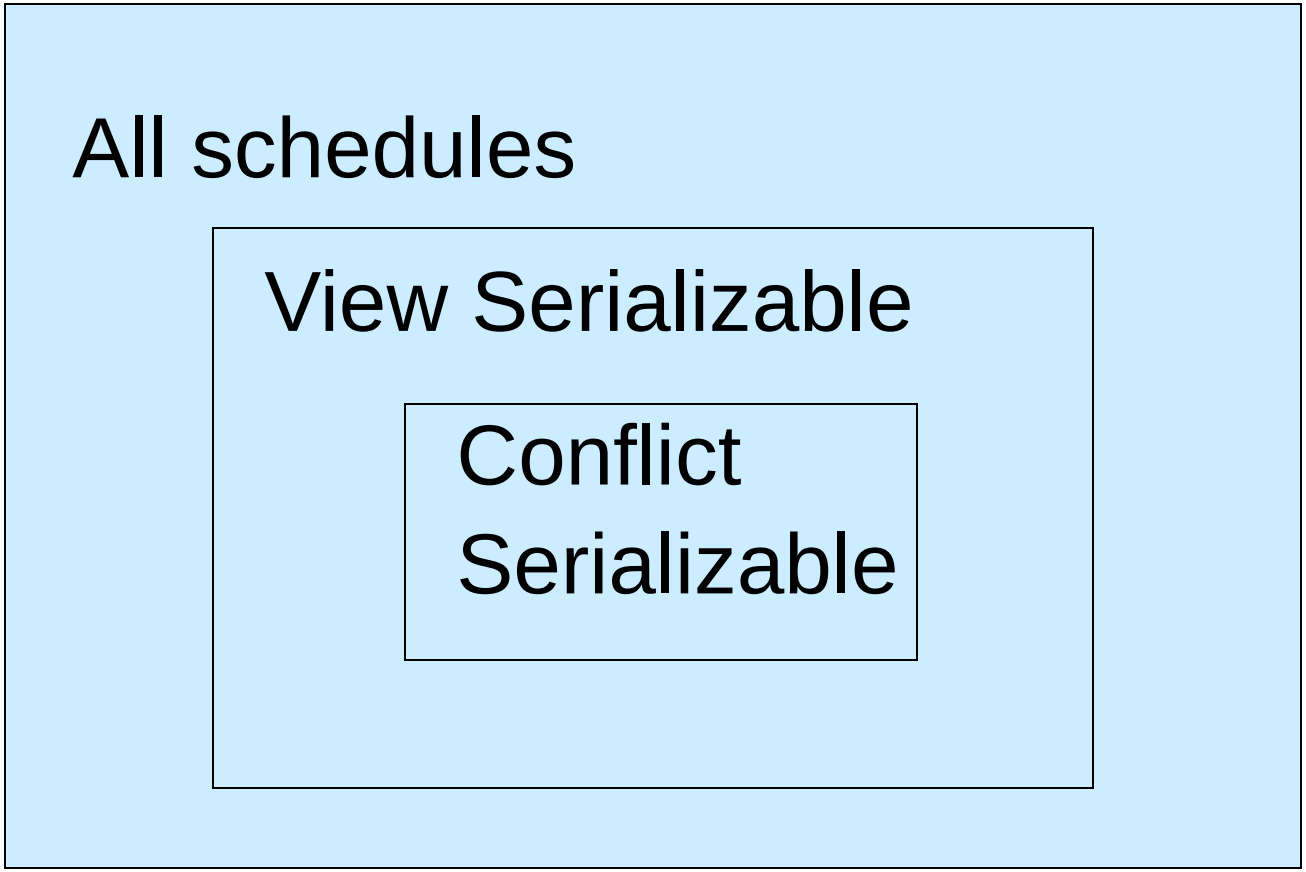
Hard to ensure; more conservative approaches are used in practice



Simplified view of transactions

- We **ignore operations** other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Diagram



Conflicts

- Definition
  - is a pair of consecutive actions in a schedule such that, if their order is interchanged, then the behavior of at least one of the transactions involved can change.

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.

Conflicting Instructions

- Non-conflicting actions: Let T_i and T_j be two different transactions (i ≠ j), then:
  - $r_i(X)$; $r_j(Y)$ is never a conflict, even if X = Y.
  - $r_i(X)$; $w_j(Y)$ is not a conflict provided X ≠ Y.
  - $w_i(X)$; $r_j(Y)$ is not a conflict provided X ≠ Y.
  - Similarly, $w_i(X)$; $w_j(Y)$ is also not a conflict, provided X ≠ Y.

continued…

- Three situations of conflicting actions (where we **may not swap** their order)
  - **Two actions of the same transaction.**
    - e.g., $r_i(X);w_i(X)$
  - **Two writes of the same database** element by different transactions.
    - e.g., $w_i(X);w_j(X)$
  - **A read and a write of the same database** element by different transactions.
    - e.g., $r_i(X);w_j(X)$

- To summarize, any two actions of different transactions may be swapped unless:
  - **They involve the same database element, and**
  - **At least one of them is a write operation.**

Converting conflict-serializable schedule to a serial schedule

- S: $r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B);$

- $r_1(A); w_1(A); r_2(A); \mathbf{w_2(A); r_1(B);} w_1(B); r_2(B); w_2(B);$
- $r_1(A); w_1(A); \mathbf{r_2(A); r_1(B);} w_2(A); w_1(B); r_2(B); w_2(B);$
- $r_1(A); w_1(A); r_1(B); r_2(A); \mathbf{w_2(A); w_1(B);} r_2(B); w_2(B);$
- $r_1(A); w_1(A); r_1(B); \mathbf{r_2(A); w_1(B);} w_2(A); r_2(B); w_2(B);$
- $r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B);$

conflict-serializable schedule

- **Schedule 3 can be transformed into Schedule 6**, a serial schedule, by series of swaps of non-conflicting instructions..

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)

conflict-serializable schedule

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 3

- **S3:** r1(A), w1(A), r2(A), **w2(A), r1(B),** w1(B), r2(B), w2(B)
- r1(A),w1(A),r2(A),r1(B),**w2(A), w1(B),** r2(B), w2(B)
- r1(A),w1(A),**r2(A),r1(B),** w1(B), w2(A), r2(B), w2(B)
- r1(A),w1(A),r1(B),**r2(A),w1(B),** w2(A), r2(B), w2(B)
- **S6:** r1(A),w1(A), r1(B), w1(B),r2(A), w2(A), r2(B), w2(B)

conflict-serializable schedule

- 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

conflict-serializable schedule

- Example of a schedule that is not conflict serializable:

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

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

Conflict Serializability & Graphs

- Theorem: **A schedule is conflict serializable iff its precedence graph is acyclic**.
- Theorem: 2PL ensures that the precedence graph will be acyclic!
- Strict 2PL improves on this by avoiding cascading aborts, problems with undoing WW conflicts; i.e., ensuring recoverable schedules.

Precedence Graphs

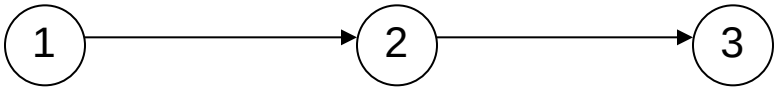
- For a schedule S, involving transactions T_1 and T_2 (among other transactions), we say that T_1 takes precedence over T_2 (written as $T_1 <_s T_2$)if there are actions A_1 of T_1 and A_2 of T_2, such that:
  - A_1 is ahead of A_2 in S,
  - **Both A_1 and A_2 involve the same database element, and**
  - **At least one of them is a write operation.**

Example of a precedence graph:

Consider a schedule S which involves three transactions T_1, T_2 and T_3, i.e.,

S: $r_2(A)$; $r_1(B)$; $w_2(A)$; $r_3(A)$; $w_1(B)$; $w_3(A)$; $r_2(B)$; $w_2(B)$;

The precedence graph for this as is shown below



Do not consider
Same database, same transaction, Read – Read, which will not conflict

Test for conflict-serializability

- Construct the precedence graph for S and observe if there are any cycles.
  - If yes, then S is not conflict-serializable
  - Else, it is a conflict-serializable schedule.
- Example of a cyclic precedence graph:
  - Consider the below schedule

    S_1: $r_2(A)$; $r_1(B)$; $w_2(A)$; $r_2(B)$; $r_3(A)$; $w_1(B)$; $w_3(A)$; $w_2(B)$;

    The precedence graph for this as shown below:

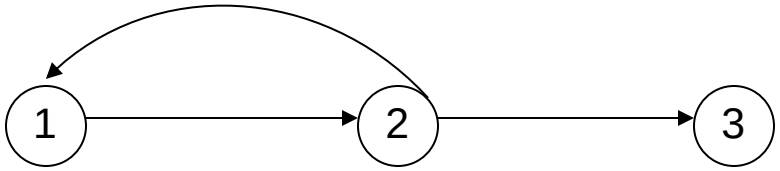


Figure 2

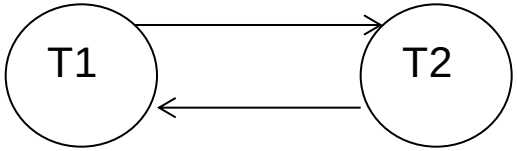
Precedence Graph- Algorithm

1. Create a node T in the graph for each participating transaction in the schedule.
2. Draw an edge from T_i to T_j in the graph for the conflicting operation.
   - read_item(X) and write_item(X)
   - write_item(X) and read_item(X)
   - write_item(X) and write_item(X)
3. **The Schedule S is serializable if there is no cycle in the precedence graph**.

Precedence Graph - Conflict serializable

- S : r1(x) r1(y) w2(x) w1(x) r2(y)
  - x: r1(x) w2(x) w1(x)
  - y: r1(y) r2(y)
- Step 1: Make two nodes corresponding to Transaction T_1 and T_2.

- Step 2: Draw edges for the conflicting pair

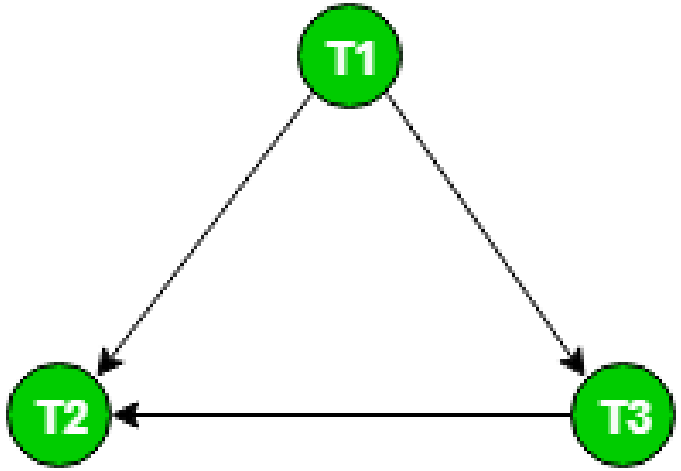


- Step 3: Check if there is any cycle formed in the graph. If there is no cycle found, then the schedule is conflict serializable otherwise not

Since the above graph is cyclic, we can conclude that it is **not conflict serializable**

Example

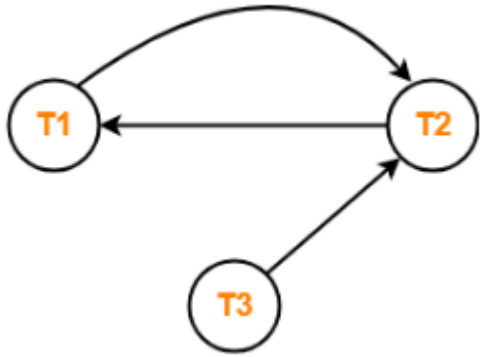
- S1: r1(x) r3(y) w1(x) w2(y) r3(x) w2(x)
  - x: r1(x) **w1(x)** r3(x) **w2(x)**
  - y: r3(y) w2(y)



Since the graph is acyclic, the schedule is conflict serializable.

Example

- Check whether the given schedule S is conflict serializable or not-
  - **S : $R_1(A)$, $R_2(A)$, $R_1(B)$, $R_2(B)$, $R_3(B)$, $W_1(A)$, $W_2(B)$**
  - **A: $R_1(A)$, $R_2(A)$, $W_1(A)$**
  - **B: $R_1(B)$, $R_2(B)$, $R_3(B)$, $W_2(B)$**

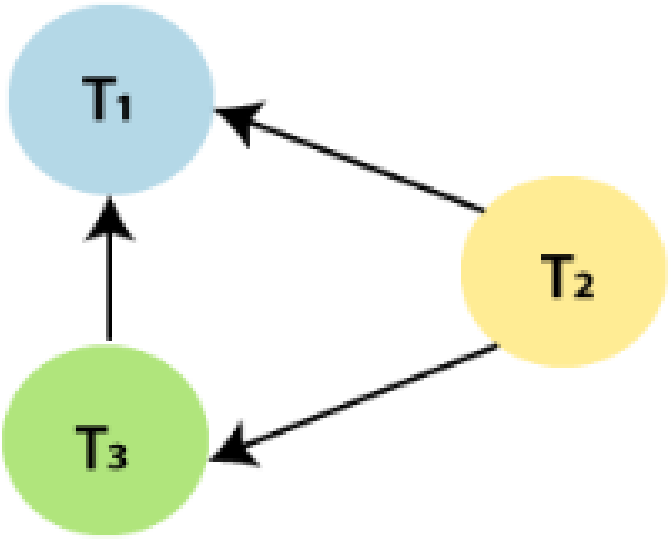


- Clearly, there exists a cycle in the precedence graph.
- Therefore, the given schedule S is not conflict serializable.

Exercise

Time	Transaction T1	Transaction T2	Transaction T3
t1	Read(X)		
t2			Read(Y)
t3			Read(X)
t4		Read(Y)	
t5		Read(Z)	
t6			Write(Y)
t7		Write(Z)	
t8	Read(Z)		
t9	Write(X)		
t10	Write(Z)		

Solution



- Since the graph is acyclic, the schedule is conflict serializable.

View Serializability

- Let *S* and *S´* be two schedules with the same set of transactions. *S* and *S´* are **view equivalent** if the following three conditions are met, for each data item *Q,*
  1. **Initial reads:** If in schedule S, transaction T_i reads the initial value of *Q*, then in schedule *S'* also transaction T_i must read the initial value of *Q.*
  2. **W-R Conflict:** If in schedule S transaction T_i executes **read**(*Q*), and that value was produced by transaction T_j **—write(Q)** (if any), then in schedule *S'* also transaction T_i must read the value of *Q* that was produced by the same **write**(Q) operation of transaction T_j .
  3. **Final write:** The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S'.*

As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

View Serializability

- **Initial reads**
  - S1: r1(A)
  - S2: r1(A)

- **W-R Conflict:**
  - These two schedule are not view-equivalent:
  - S1: w2(A), r3(A)
  - S2: w1(A), r3(A)

- **Final write**
  - S1: w1(A)
  - S2: w1(A)

View Serializability

- S1: r1(P), w2(P)
- S2: w2(P) r1(P)
- Initial reads
  - Step1: Satisfied
- W-R Conflict
  - Step2: Not Satisfied
- Final write

  Step3: Satisfied
- Conclusion
  - Not view serializable

Schedule S1:

Time	Transaction T1	Transaction T2
T1	Read (P)	
T2		Write (P)

Schedule S2:

Time	Transaction T1	Transaction T2
T1		Write (P)
T2	Read (P)	

View Serializability

- S1: w1(P), r2(P), w3(P)
- S2: r2(P), w1(P), w3(P)
- Initial reads
  - Step1: Satisfied
- W-R Conflict
  - Step2: Not Satisfied
- Final write
  - Step3: Satisfied
- Conclusion
  - Not view serializable

Schedule S1:

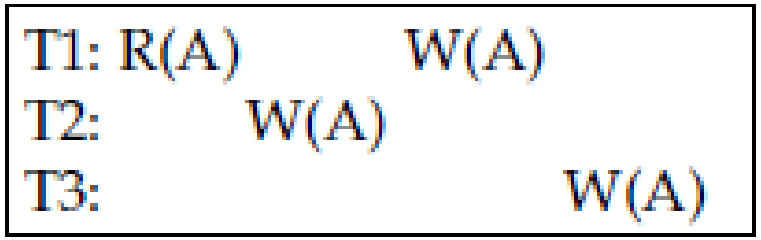
Time	Transaction T1	Transaction T2	Transaction T3
T1	Write (P)		
T2		Read (P)	
T3			Write (P)

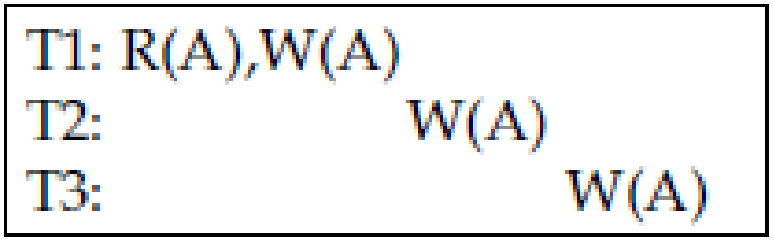
Schedule S2:

Time	Transaction T1	Transaction T2	Transaction T3
T1		Read (P)	
T2	Write (P)		
T3			Write (P)

Exercise

Schedules S1 and S2 are view equivalent





View Serializability Example

S: r2(B) w2(A) r1(A) r3(A) w1(B) w2(B) w3(B)

S': r2(B) w2(A) w1(B) r1(A) w2(B) r3(A) w3(B)

View Serializability Example

Step 1: Initial reads

r2(**B**) w2(A) r1(A) r3(A) w1(B) w2(B) w3(B)

---

r2(**B**) w2(A) w1(B) r1(A) w2(B) r3(A) w3(B)

View Serializability Example

Step 2: W-R Conflict

r2(B) **w2(A) r1(A) r3(A)** w1(B) w2(B) w3(B)

---

r2(B) **w2(A)** w1(B) **r1(A)** w2(B) **r3(A)** w3(B)

r2(B) w1(B) w2(B) w3(B)

w2(A) r1(A) r3(A)

View Serializability Example

Step 3: Final write

r2(B) w2(A) r1(A) r3(A) w1(B) w2(B) w3(B)

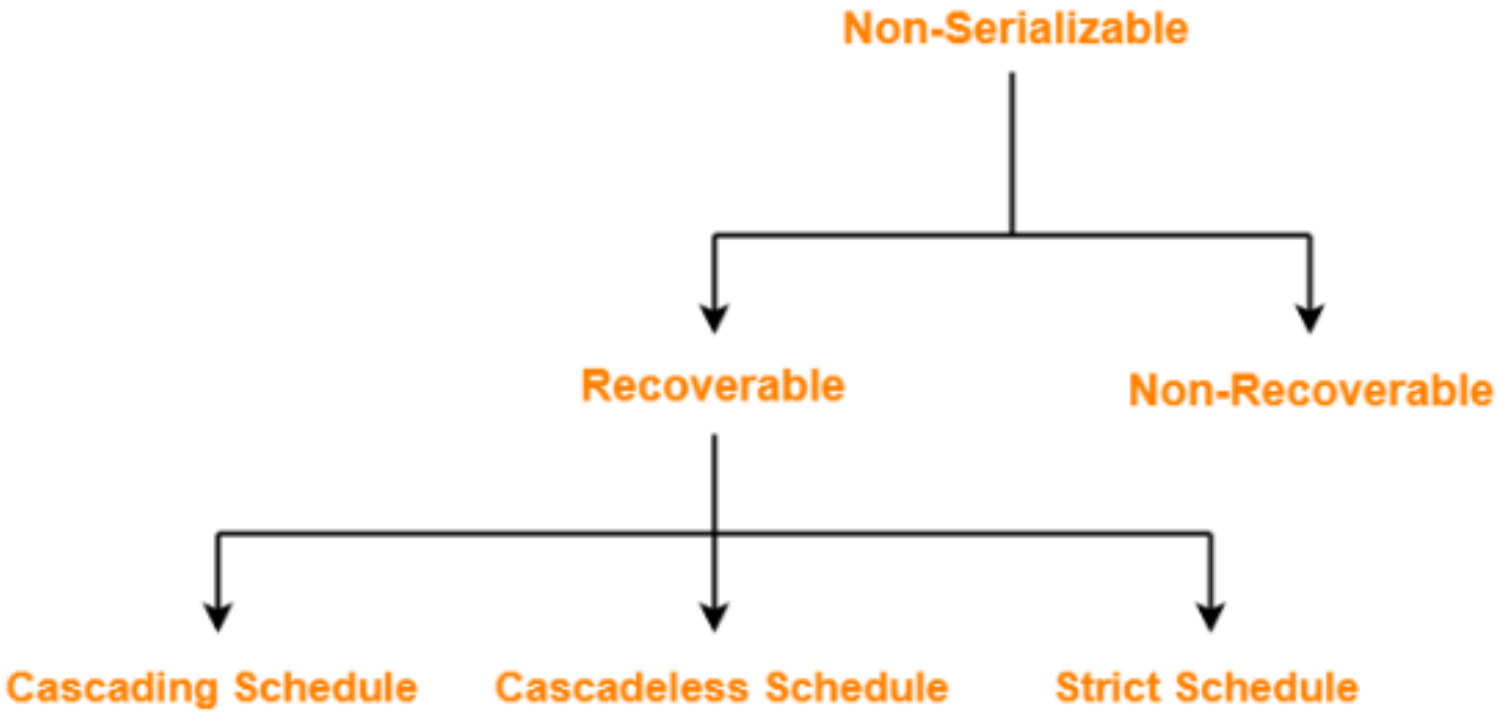
---

r2(B) w2(A) w1(B) r1(A) w2(B) r3(A) w3(B)

Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to **abort.**

Recoverable Schedules



Non-recoverable Schedule.

- If in a schedule,
  - A transaction performs a **dirty read** operation from an uncommitted transaction
  - And **commits before the transaction from which it has read the value**
    - then such a schedule is known as an **Non-recoverable Schedule**.

Recoverable Schedules

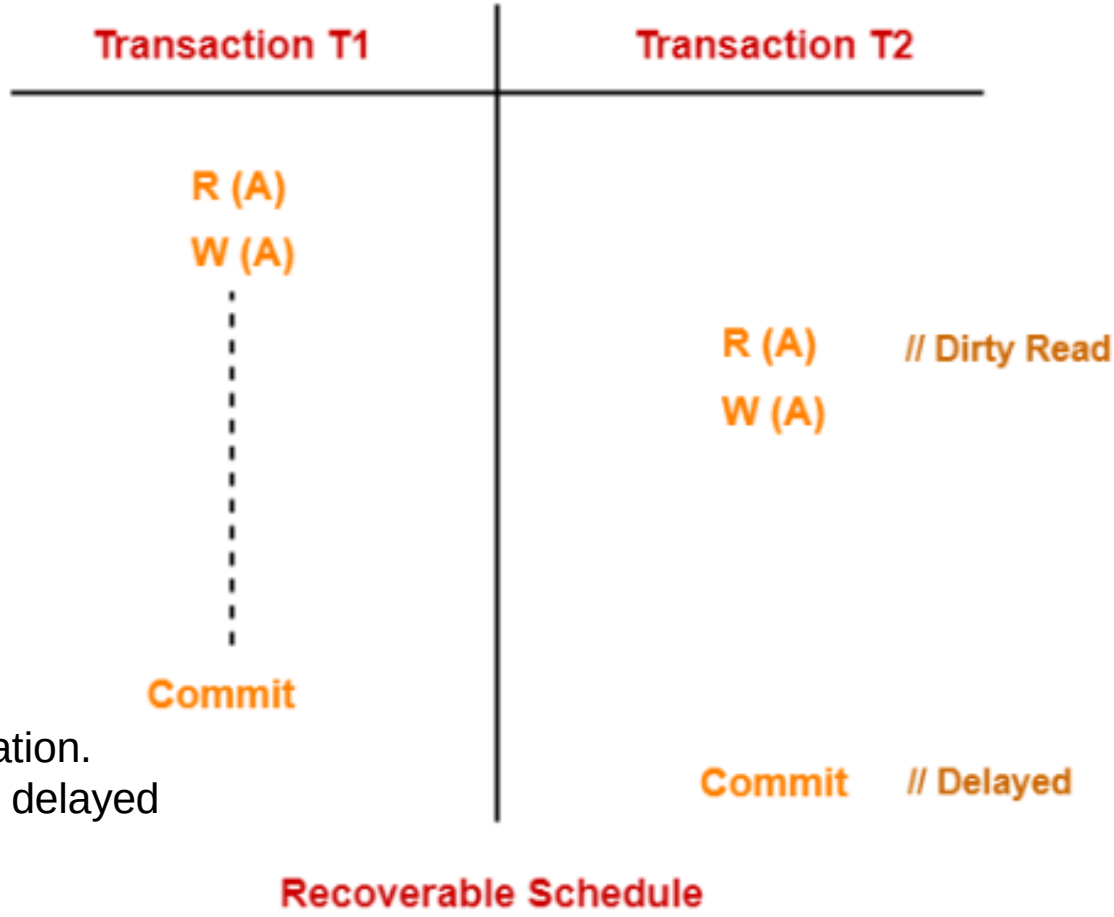
Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule** — if a transaction T_j reads a data item previously written by a transaction T_i , then the commit operation of T_i appears before the commit operation of T_j.
- The following schedule (Schedule 11) is not recoverable if T_9 commits immediately after the read

T_8	T_9
read (A)	
write (A)	
	read (A) // Dirty Read
	commit
read (B)	
Rollback	

- T2 performs a dirty read operation.
- T2 commits before T1.
- T1 fails later and roll backs.
- The value that T2 read now stands to be incorrect.
- T2 can not recover since it has already committed.

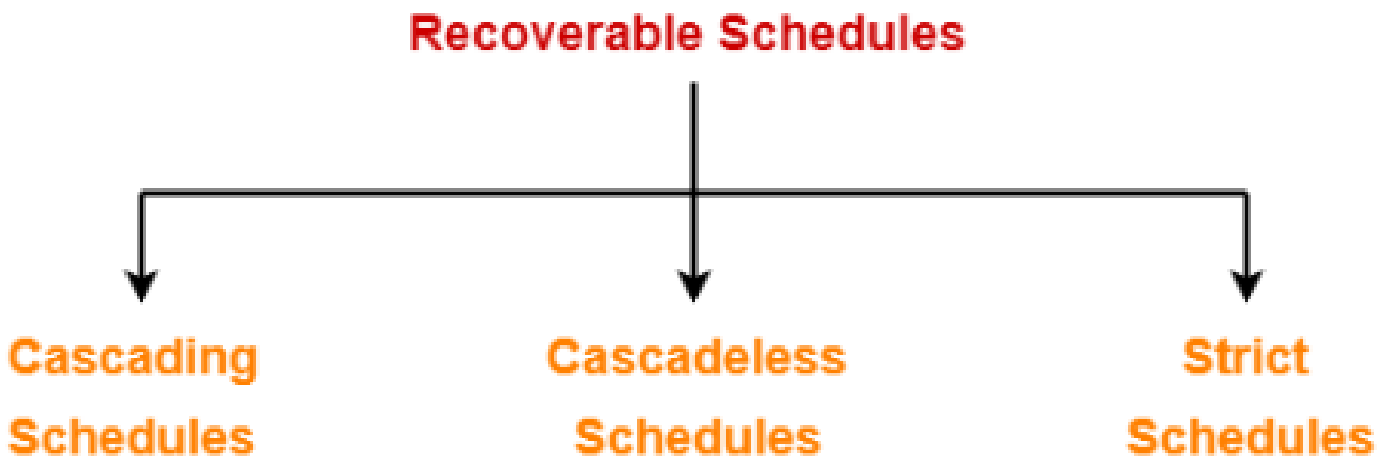
Recoverable Schedules

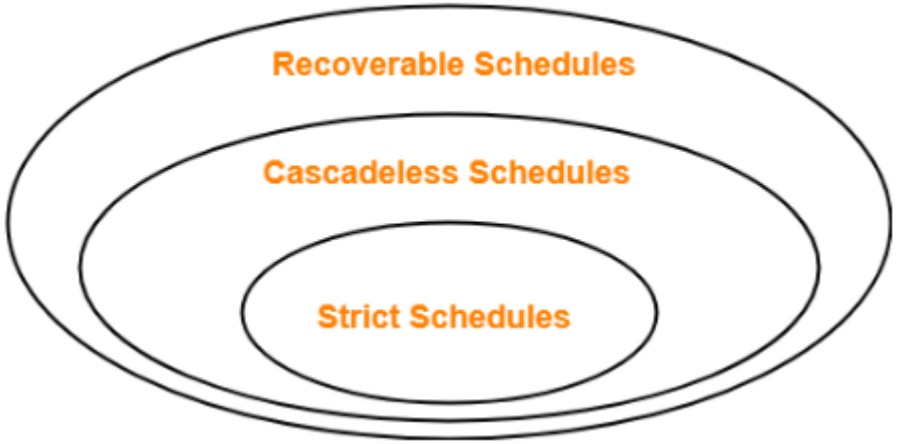


Recoverable Schedule

- T2 performs a dirty read operation.
- The commit operation of T2 is delayed till T1 commits or roll backs.
- T1 commits later.
- T2 is now allowed to commit.
- In case, T1 would have failed, T2 has a chance to recover by rolling back.

Types of Recoverable Schedules





Cascading Rollbacks

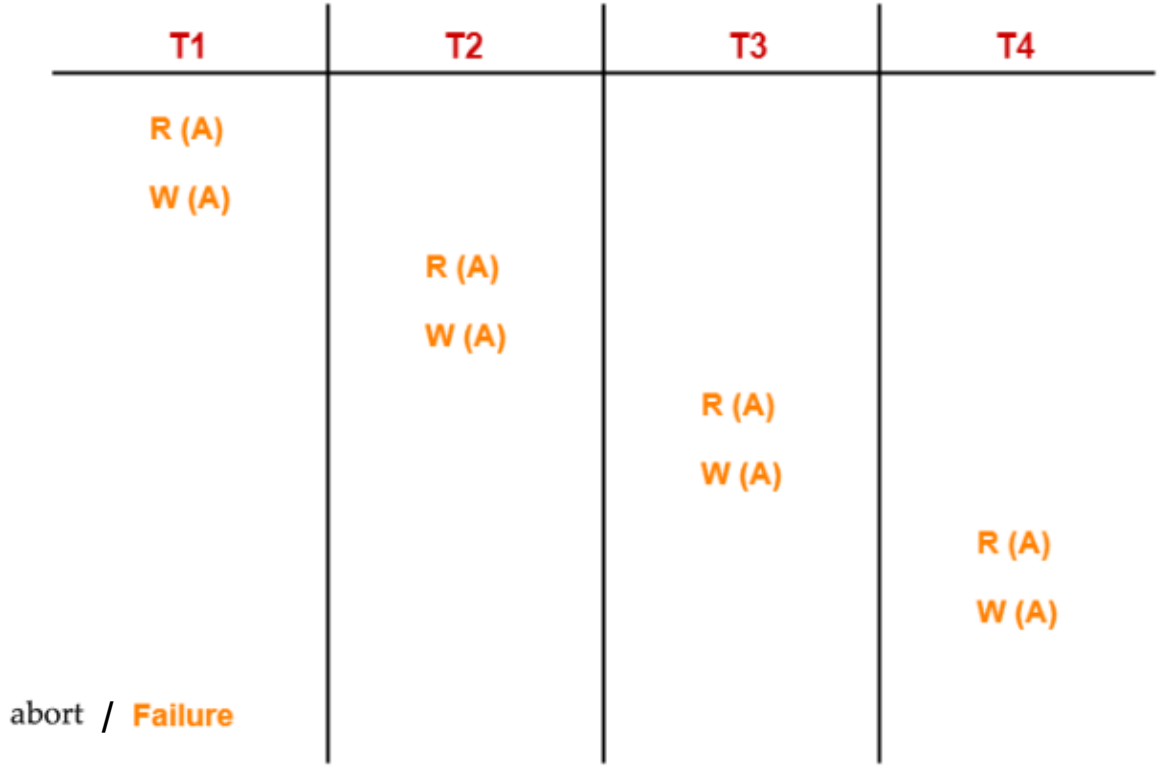
- **Cascading rollback** – **a single transaction failure leads to a series of transaction rollbacks.** Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

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

  If T_{10} fails, T_{11} and T_{12} must also be rolled back.

- Can lead to the undoing of a significant amount of work

Cascading Schedules and Rollbacks

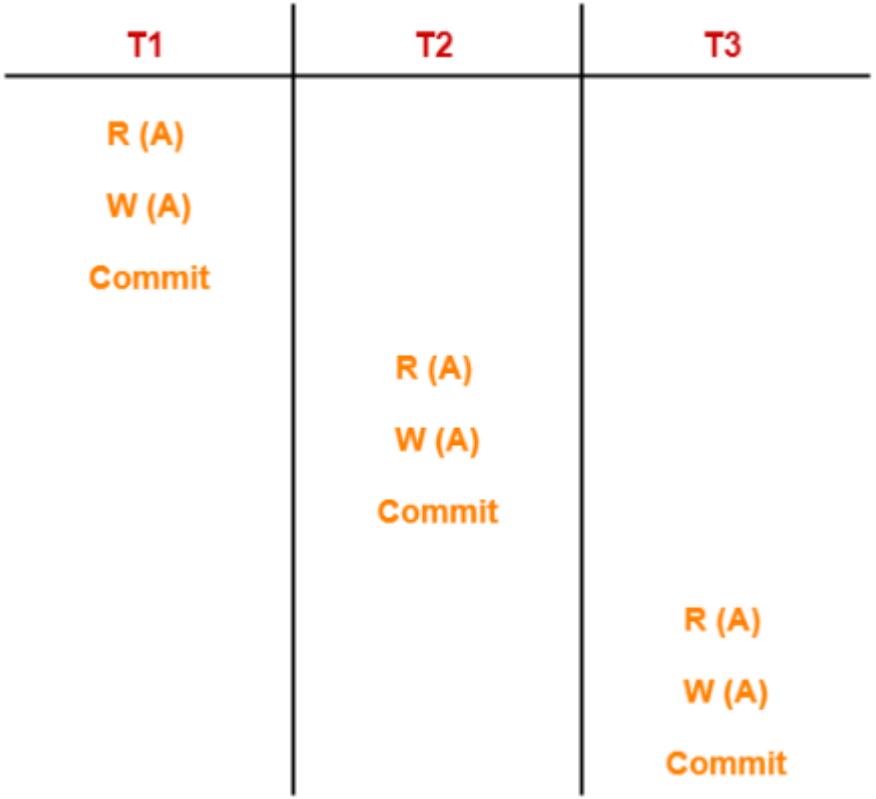


Cascading Recoverable Schedule

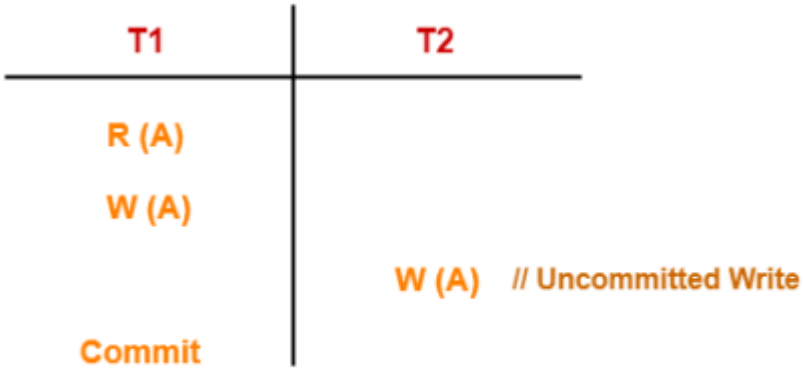
Transaction T2 depends on transaction T1.
Transaction T3 depends on transaction T2.
Transaction T4 depends on transaction T3
The failure of transaction T1 causes T2, T3 and T4 to rollback.
Such a rollback is called as a **Cascading Rollback**.

Cascadeless Schedule

- If in a schedule, a transaction is **not allowed to read a data item** until the last transaction that has written it is committed or aborted, then such a schedule is called as a **Cascadeless Schedule**.



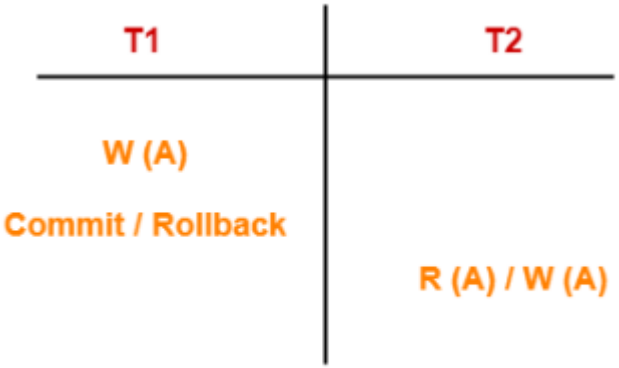
Cascadeless Schedule



Cascadeless Schedule

Strict Schedule

- If in a schedule, a transaction is **neither allowed to read nor write a data item** until the last transaction that has written it is committed or aborted, then such a schedule is called as a **Strict Schedule**.



Strict Schedule

References

- Abraham Silberschatz, Henry F. Korth and S. Sudarshan, *Database System Concepts*, 6th Edition, McGraw-Hill
- https://www.gatevidyalay.com/serializability-in-dbms-conflict-serializability/