Schedules and Serializability

Database System Concepts, 6th Ed.

Chapter 14: Transactions

- Concurrent Executions
- Serializability
- Implementation of Isolation

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
    - Will study in Chapter 16, after studying notion of correctness of concurrent executions.

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

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

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	

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

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

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

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. **conflict serializability**
  2. **view serializability**

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.

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.

Conflict Serializability

- If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent**.
- We say that a schedule S is **conflict serializable** if it is conflict **equivalent to a serial schedule**

Conflict Serializability (Cont.)

- 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.

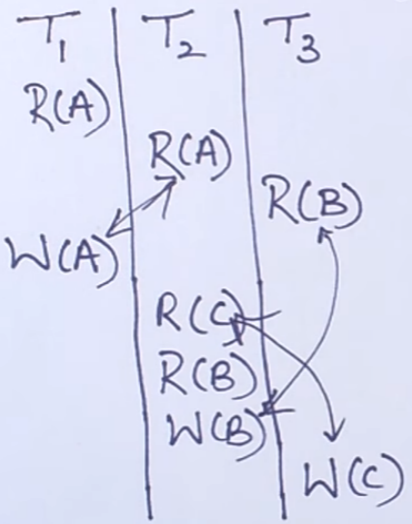
T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

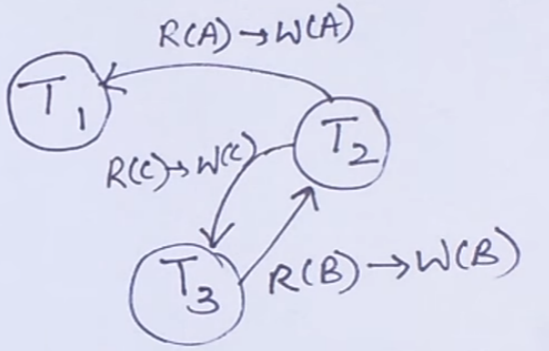
T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

Schedule S_1

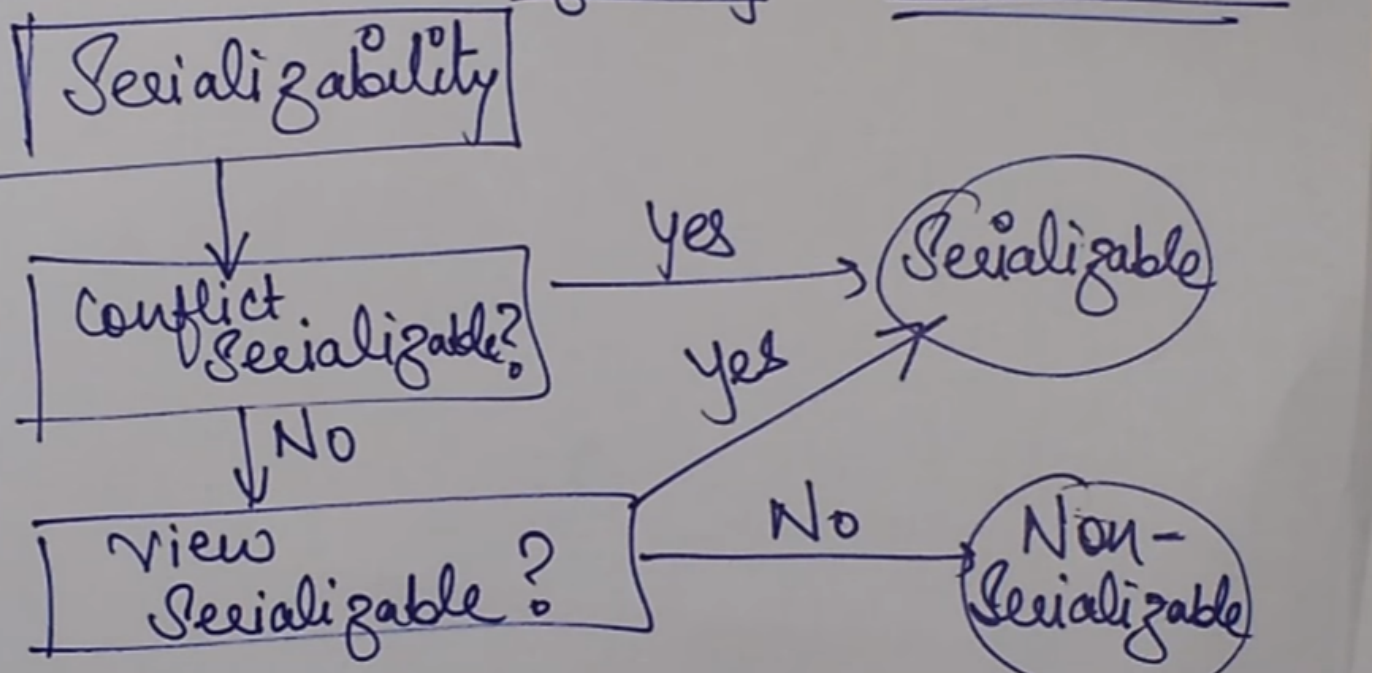


CONFLICT SERIALIZABILITY



CYCLE → NON-SERIALIZABLE

Serializability : A Broader View



Conflict Serializability

R-W
W-R
W-W

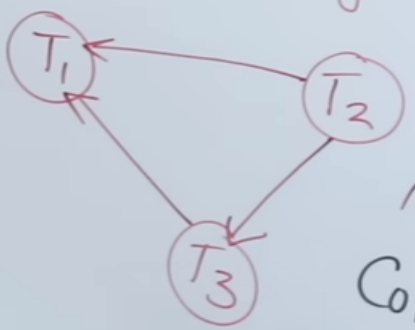
S

T_1	T_2	T_3
~~R(x)~~		
		~~R(y)~~
		~~R(x)~~
	~~R(y)~~	
	~~R(z)~~	
		W(y)
	W(z)	
R(z)		
W(x)		
W(z)		

W(y) < R(y), W(y)

- Check Conflict pairs in other transactions and draw edges

Precedence graph



Loop / Cycle

No loop / Cycle

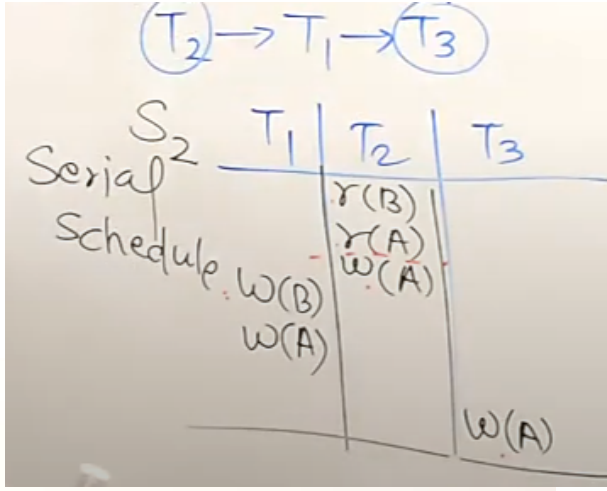
Conflict Serializable

↓

Serializable

↓

Consistent



View Serializability & View Serializable Schedule

IR: by T_2 on data item (A), (B)

FW: by T_1 on data item (B)
by T_3 on data item (A)

WR: Not present so need to check

1. **Initial Read**
2. **Final Write**
3. **Write-Read**

Example :

Consider the following Non-Serial Schedule S_1 :

T_1	T_2	T_3
	r(B)	
	r(A)	
w(B)		
w(A)		
	w(A)	
		w(A)

→ Not Conflict Serializable

is Schedule S_1 View-Serializable Schedule ?

Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, "**dirty reads**"):

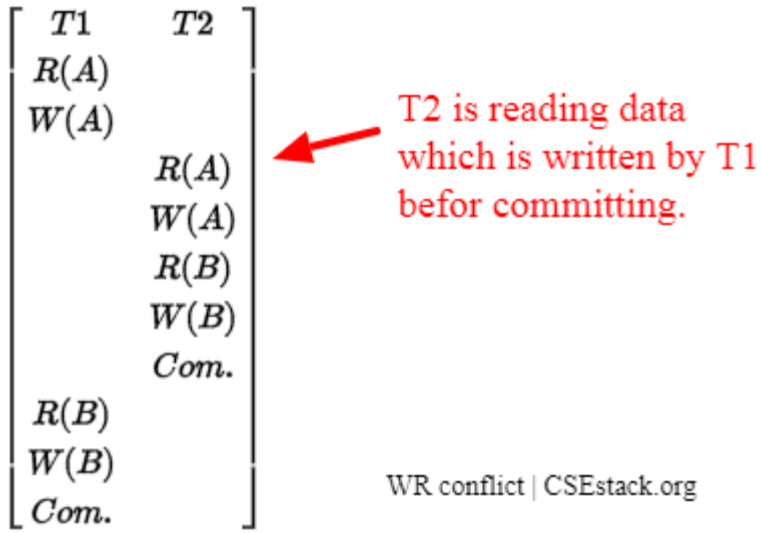
T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
	C
read (B)	
write (B)	
Abbort	

Occurs when a transaction reads data that has been modified by another transaction, but not yet committed

Anomalies with Interleaved Execution

- Unrepeatable Reads (RW Conflicts):

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



A user reads two different values for the same record

because another user updated the record in between the two reads.

Anomalies (Continued)

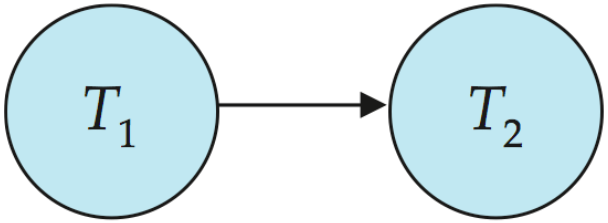
- Overwriting Uncommitted Data (WW Conflicts):

T_1	T_2
write (A)	
	write (A)
	write (B)
	C
write (B)	
C	

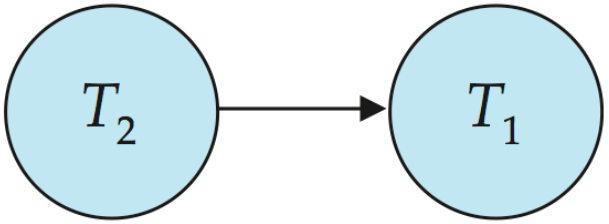
also known as overwriting uncommitted data

Precedence graph for schedule 1 and schedule 2.

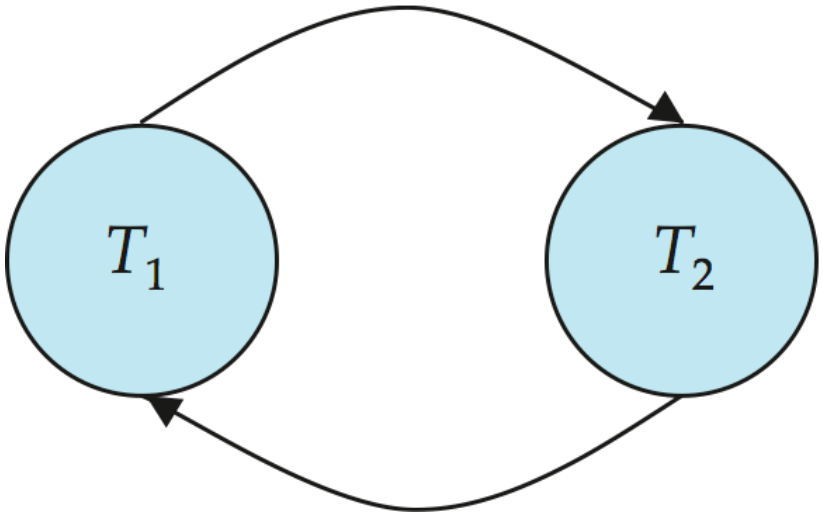
- Simple and efficient method for determining conflict serializability of a schedule.
- Consider a schedule S. We construct a directed graph, called a **precedence graph**, from S.
- This graph consists of a pair G = (V, E), where V is a set of vertices and E is a set of edges.
- The set of edges consists of all edges Ti → T j for which one of three conditions holds:
  1. Ti executes write(Q) before T j executes read(Q).
  2. Ti executes read(Q) before T j executes write(Q).
  3. Ti executes write(Q) before T j executes write(Q).

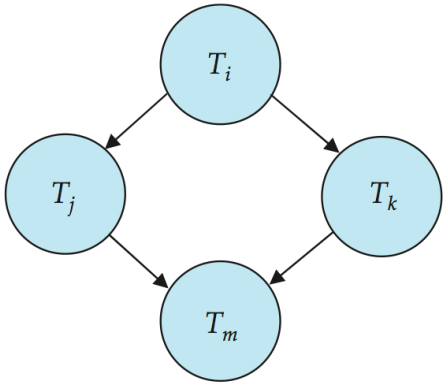


(a)

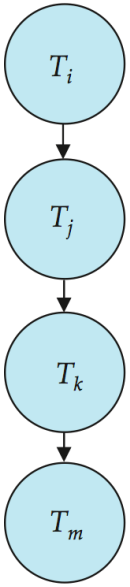


(b)

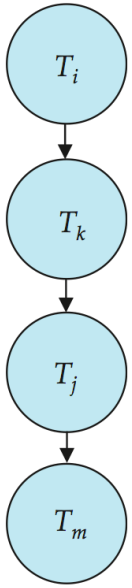

T_1
T_2



(a)



(b)



(c)

Figure 14.13

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)

View Serializability

- Consider two schedules S and S′, where the same set of transactions participates in both schedules. The schedules S and S′ are said to be **view equivalent** if three conditions are met:
  - For each data item Q, if transaction Ti reads the initial value of Q in schedule S, then transaction Ti must, in schedule S′, also read the initial value of Q.
  - For each data item Q, if transaction Ti executes read(Q) in schedule S, and if that value was produced by a write(Q) operation executed by transaction T j , then the read(Q) operation of transaction Ti must, in schedule S′, also read the value of Q that was produced by the same write(Q) operation of transaction T j .
  - For each data item Q, the transaction (if any) that performs the final write(Q) operation in schedule S must perform the final write(Q) operation in schedule S′.

View Serializability (Cont.)

- Schedule 5 is view equivalent to the serial schedule <T27, T28, T29>, since the one read(Q) instruction reads the initial value of Q in both schedules and T29 performs the final write of Q in both schedules.
- Every conflict-serializable schedule is also view serializable, but there are view-serializable schedules that are not conflict serializable.
- In schedule 5, transactions T28 and T29 perform write(Q) operations without having performed a read(Q) operation.
- Writes of this sort are called **blind writes**.
- Blind writes appear in any view-serializable schedule that is not conflict serializable.

Schedule 4

T_{27}	T_{28}
read(Q)	
	write(Q)
write(Q)	

Schedule 5

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

Transaction Isolation and Atomicity

- Lets see the effect of transaction failures during concurrent execution.
- If a transaction Ti fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction.
- In a system that allows concurrent execution, the atomicity property requires that any transaction Tj that is dependent on Ti is also aborted.
- To achieve this, we need to place restrictions on the type of schedules permitted in the system.
  - Non-recoverable Schedules
  - Recoverable Schedules
  - Cascading Rollback
  - Cascadeless Schedules

Nonrecoverable Schedules

- Consider the partial schedule in which T7 is a transaction that performs only one instruction: read(A).
- We call this a partial schedule because we have not included a commit or abort operation for T6.
- Notice that T7 commits immediately after executing the read(A) instruction. Thus, T7 commits while T6 is still in the active state.
- Now suppose that T6 fails before it commits. T7 has read the value of data item A written by T6 . Therefore, we say that T7 is dependent on T6.
- Because of this, we must abort T7 to ensure atomicity. However, T7 has already committed and cannot be aborted. Thus, we have a situation where it is impossible to recover correctly from the failure of T6.
- This Schedule is an example of a **nonrecoverable schedule**.

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

Recoverable Schedules

- A recoverable schedule is one where, for each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti , the commit operation of Ti appears before the commit operation of Tj .
- For the example of the previous schedule is said to be recoverable, T7 would have to delay committing until after T6 commits.

Cascading Rollback

- A situation in which the abort of one transaction forces the abort of another transaction to prevent the second transaction from reading invalid data. Also called "cascading rollback".
- If in a schedule, failure of one transaction causes several other dependent transactions to rollback or abort, then such a schedule is called as a **Cascading Schedule or Cascading Rollback or Cascading Abort**.

T_8	T_9	T_{10}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
abort		

Cascadeless Schedules

- It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called **cascadeless schedules**.
- Formally, a cascadeless schedule is one where, for each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti , the commit operation of Ti appears before the read operation of Tj .
- It is easy to verify that every cascadeless schedule is also recoverable.

THANK YOU