Transaction Processing- II

Scheduling of Transactions

- **Serializable Schedule:** A non-serial schedule that is equivalent to some serial execution of transactions is called a serializable schedule.
- The objective of serializability is to find non-serial schedules that allow transactions to execute concurrently without interfering with one another, and thereby produce a database state that could be produced by a serial execution.

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**

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

- We need to check the conflicts between two consecutive operations of two different transactions in a schedule. Operations upon data can be *read or write*. There are two possibilities:
  - If two consecutive operations are on different data items, then they do not conflict i.e., their order of execution does not matter and we can swap them without affecting their result.
  - If two consecutive operations are on same data items, then they can conflict i.e. their order of execution matters and we cannot swap them.

Conflict Serializability

- Consider a schedule S in which there are two consecutive operations O_i and O_j of two transactions T_i and T_j and both operations refer to the same data item A. Then, there are following four cases:
  1. O_i= read(A) and O_j= read(A). Then, their order does not matter because same value of A is read by T_i and T_j.
  2. O_i= read(A) and O_j= write(A). Then, their order matters. If O_i comes before O_j then, O_j does not read the value of A written by O_i. If O_j comes before O_i then, O_i reads the value of A that is written by O_j.
  3. O_i= write(A) and O_j= read(A). Then, their order matters. If O_i comes before O_j then, O_j reads the value of A written by O_i. If O_j comes before O_i then, O_j does not read the value of A that is written by O_i.
  4. O_i= write(A) and O_j= write(A). Then, their order matters. If O_i comes after O_j then, the value of A written by O_i is stored in database. If O_j comes after O_i then, the value of A written by O_j is stored in database.

Conflict Serializability

- Schedule 3 can be transformed into Schedule 6 – a serial schedule where T_2 follows T_1, by a series of swaps of non-conflicting instructions.
  - Swap T1.read(B) and T2.write(A)
  - Swap T1.read(B) and T2.read(A)
  - Swap T1.write(B) and T2.write(A)
  - Swap T1.write(B) and T2.read(A)

These swaps do not conflict as they work with different items (A or B) in different transactions.

- 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(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Schedule 5

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

Schedule 6

Conflict Serializability (Cont.)

- 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 >.

Question

Consider the following schedule for a set of three transactions.

w1(A),w2(A),w2(B),w1(B),w3(B)

Find whether the schedule is conflict serializable or not?

T_1	T_2	T_3
W(A)		
c	W(A)	
	W(B)	
W(B)	c	conflict
		W(B)

So, it is neither conflict equivalent nor conflict serializable

Q ⇒ $W_1(A)\ W_2(A)\ W_2(B)\ W_1(B)\ W_3(B)$

Question

ScheduleU:

r2(A),w2(A),r1(A),w1(A),r2(B),w2(B)

Check whether the given schedule is conflict serializable?

$r_2(A), w_2(A), r_1(A), w_1(A), r_2(B), w_2(B)$

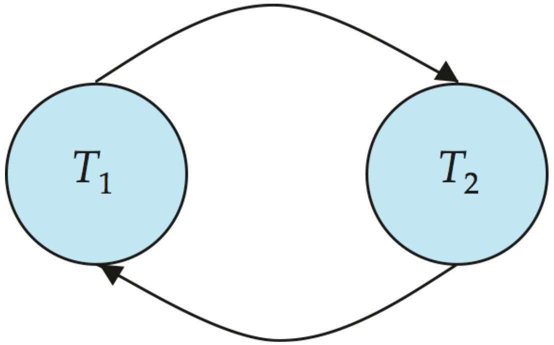
T_1	T_2
	$r(A)$
	$w(A)$
	conflict
$r(A)$	
$w(A)$	
	no conflict
	$r(B)$
	$w(B)$

→

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

Precedence Graph

- Consider some schedule of a set of transactions $T_1, T_2, ..., T_n$
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- **Example**

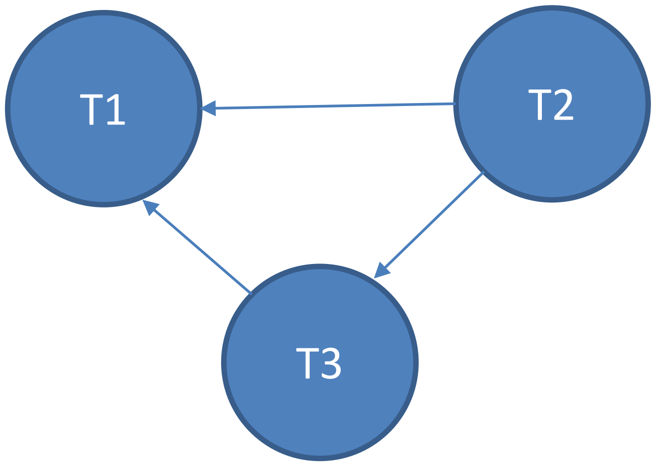


T1	T2	T3
R(X)		
		R(Y)
		R(X)
	R(Y)	
	R(Z)	
		W(Y)
	W(Z)	
R(Z)		
W(X)		
W(Z)		

- Check the conflict pairs in other transaction and draw edges

T1	T2	T3
R(X)		
		R(Y)
		R(X)
	R(Y)	
	R(Z)	
		W(Y)
	W(Z)	
R(Z)		
W(X)		
W(Z)		

- Check the conflict pairs in other transaction and draw edges



Conflict Serializability

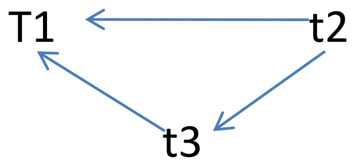
- If no loop or cycle found then schedule is conflict serializable.

serializable

- Consistent (no conflicts)
- Now, for the S schedule there are 6 possibilities
- T1 -> T2 -> T3
- T1 -> T3 - > T2
- T2 -> T1 -> T3
- T2 -> T3 -> T1
- T3 ->T1 -> T2
- T3 -> T2 -> T1

to find the conflict equivalent schedule from these possibilities we need to check indegree of precedence graph

Conflict Serializability



Vertex T2 is having 0 indegree, So REMOVE THIS

- T1
-
- t3 NOW, vertex t3 is having 0 indegree , So remove this

- So THE ORDER OF transactions will be
- **T2 -> t3 -> t1 this will be the conflict serializable schedule of the given schedule S.**

- Consider the following schedule:
  - $w_1(A)$, $r_2(A)$, $w_1(B)$, $w_3(C)$, $r_2(C)$, $r_4(B)$, $w_2(D)$, $w_4(E)$, $r_5(D)$, $w_5(E)$
- We start with an empty graph with five vertices labeled T_1, T_2, T_3, T_4, T_5.
- We go through each operation in the schedule:

  $w_1(A)$: A is subsequently read by T_2, so add edge $T_1 \rightarrow T_2$

  $r_2(A)$: no subsequent writes to A, so no new edges

  $w_1(B)$: B is subsequently read by T_4, so add edge $T_1 \rightarrow T_4$

  $w_3(C)$: C is subsequently read by T_2, so add edge $T_3 \rightarrow T_2$

  $r_2(C)$: no subsequent writes to C, so no new edges

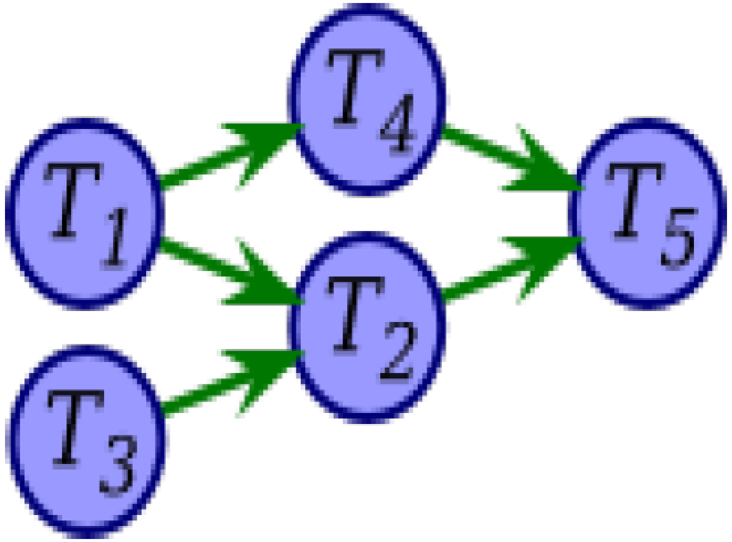
  $r_4(B)$: no subsequent writes to B, so no new edges

  $w_2(D)$: C is subsequently read by T_2, so add edge $T_3 \rightarrow T_2$

  $w_4(E)$: E is subsequently written by T_5, so add edge $T_4 \rightarrow T_5$

  $r_5(D)$: no subsequent writes to D, so no new edges

  $w_5(E)$: no subsequent operations on E, so no new edges



- We end up with precedence graph

Question

Schedule S:

r1(A),w2(A),w1(A),w3(A)

Check whether the given schedule is conflict serializable?

A) Yes, it is conflict serializable

B) No, it is not conflict serializable

View Serializability

- A schedule will view serializable if it is view equivalent to a serial schedule.
- If a schedule is conflict serializable, then it will be view serializable.
- The view serializable which does not conflict serializable contains blind writes.

View Equivalent

- Two schedules S1 and S2 are said to be view equivalent if they satisfy the following conditions:
- 1. Initial Read
- An initial read of both schedules must be the same. Suppose two schedule S1 and S2. In schedule S1, if a transaction T1 is reading the data item A, then in S2, transaction T1 should also read A.

T1	T2
Read(A)	
	Write(A)

Schedule S1

T1	T2
	Write(A)
Read(A)	

Schedule S2

Above two schedules are view equivalent because Initial read operation in S1 is done by T1 and in S2 it is also done by T1.

Updated Read

- In schedule S1, if Ti is reading A which is updated by Tj then in S2 also, Ti should read A which is updated by Tj.

T1	T2	T3
Write(A)		
	Write(A)	
		Read(A)

Schedule S1

T1	T2	T3
	Write(A)	
Write(A)		
		Read(A)

Schedule S2

Above two schedules are not view equal because, in S1, T3 is reading A updated by T2 and in S2, T3 is reading A updated by T1.

Final Write

- A final write must be the same between both the schedules. In schedule S1, if a transaction T1 updates A at last then in S2, final writes operations should also be done by T1.
-

T1	T2	T3
Write(A)		
	Read(A)	
		Write(A)

Schedule S1

T1	T2	T3
	Read(A)	
Write(A)		
		Write(A)

Schedule S2

Above two schedules is view equal because Final write operation in S1 is done by T3 and in S2, the final write operation is also done by T3.

Schedule S

T1	T2	T3
Read(A)		
	Write(A)	
Write(A)		
		Write(A)

Schedule S1

T1	T2	T3
Read(A)		
Write(A)		
	Write(A)	
		Write(A)

- **Taking first schedule S1:**

T1	T2	T3
Read(A) Write(A)		
	Write(A)	
		Write(A)

Schedule S1
Step 1: final updation on data items
In both schedules S and S1, there is no read except the initial read that's why we don't need to check that condition.
Step 2: Initial Read
The initial read operation in S is done by T1 and in S1, it is also done by T1.
Step 3: Final Write
The final write operation in S is done by T3 and in S1, it is also done by T3. So, S and S1 are view Equivalent.

What is recovery?

- Serializability helps to ensure Isolation and Consistency of a schedule
- Yet, the Atomicity and Consistency may be compromised in the face of system failures
- Consider a schedule comprising a single transaction (obviously serial):
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
  7. **commit** *// Make the changes permanent; show the results to the user*
- What if system fails after Step 3 and before Step 6?
  - Leads to inconsistent state
  - Need to rollback update of A
- This is known as **Recovery**

Recoverable Schedules

- **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 **must** appear before the commit operation of T_j.
- The following schedule is not recoverable if T_9 commits immediately after the read(A) operation.

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

- If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Recoverable Schedules

- A schedule is said to be *recoverable*, if for each pair of transaction 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.

Schedule1

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Commit	
	Commit

Schedule2

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Abort	
	Abort

- Both the above schedules are recoverable .

Recoverable Schedules

- Schedule1 is recoverable because T2 reads the value of A updated by T1 and also T1 commits before T2 which makes the value read by T2 correct. Then T2 can commit itself.
- In schedule2, if T1 is aborted, T2 has to abort because the value A it read is incorrect.
- In both cases, the database is in consistent state.

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

Cascadeless Schedules

- **Cascadeless schedules** — for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless
- Example of a schedule that is NOT cascadeless

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

Failure Classification

- To find that where the problem has occurred, we generalize a failure into the following categories:

- Transaction failure
- System crash
- Disk failure

1. Transaction failure

- The transaction failure occurs when it fails to execute or when it reaches a point from where it can't go any further. If a few transaction or process is hurt, then this is called as transaction failure.
- Reasons for a transaction failure could be -
  - **Logical errors:** If a transaction cannot complete due to some code error or an internal error condition, then the logical error occurs.
  - **Syntax error:** It occurs where the DBMS itself terminates an active transaction because the database system is not able to execute it. **For example,** The system aborts an active transaction, in case of deadlock or resource unavailability

2. System Crash

- System failure can occur due to power failure or other hardware or software failure. **Example:** Operating system error.
- **Fail-stop assumption:** In the system crash, non-volatile storage is assumed not to be corrupted.

3. Disk Failure

- It occurs where hard-disk drives or storage drives used to fail frequently. It was a common problem in the early days of technology evolution.
- Disk failure occurs due to the formation of bad sectors, disk head crash, and unreachability to the disk or any other failure, which destroy all or part of disk storage.

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are both:
  - Conflict serializable.
  - Recoverable and preferably cascadeless

- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency

- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur

- Testing a schedule for serializability *after* it has executed is a little too late!
  - Tests for serializability help us understand why a concurrency control protocol is correct

- **Goal – to develop concurrency control protocols that will assure serializability**.