Transactions

by

Dr. Pratik Roy

Definition

- A transaction is a unit of program execution that accesses and possibly updates various data items.
- A transaction is an executing program that forms a logical unit of database processing.
- **Example:** Let T is a transaction that transfers Rs. 50/- from account A to account B:

T: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Concurrency Control

- Transactions submitted by various users may execute concurrently.
  - Access and update the same database items
  - Some form of concurrency control is needed
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

Why Concurrency Control Is Needed

❑Lost Update Problem

- Occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect.

Time

T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;
write_item(X); read_item(Y);	
	write_item(X);
$Y := Y + N$; write_item(Y);	

Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

Why Concurrency Control Is Needed

❑Temporary Update (or Dirty Read) Problem

- Occurs when one transaction updates a database item and then the transaction fails for some reason.
- Meanwhile, the updated item is accessed (read) by another transaction before it is changed back to its original value.

Time ↓

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y);	

← Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

*T*1 updates item *X* and then fails before completion

Why Concurrency Control Is Needed

❑Incorrect Summary Problem

- If one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated.

T_1	T_3

	sum := 0;
	read_item(A);
	sum := *sum* + A;
	⋮

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	sum := *sum* + X;
	read_item(Y);
	sum := *sum* + Y;

read_item(Y);
$Y := Y + N$;
write_item(Y);	

T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

Why Concurrency Control Is Needed

❑Unrepeatable Read Problem

- Transaction T reads the same item twice and the item is changed by another transaction T’ between the two reads.
- T receives different values for its two reads of the same item.

Required Properties of a Transaction

- Consider a transaction to transfer Rs. 50/- from account A to account B:

  read(A)
  A := A – 50
  write(A)
  read(B)
  B := B + 50
  write(B)

- **Atomicity requirement**
  - If the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database

Required Properties of a Transaction

- **Durability requirement**
  - Once the user has been notified that the transaction has completed (i.e., the transfer of the Rs. 50/- has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.
- **Consistency requirement** in above example:
  - Sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
- A transaction, when starting to execute, must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent
  - Erroneous transaction logic can lead to inconsistency

Required Properties of a Transaction

- **Isolation requirement:** If between steps 3 and 6 (of the fund transfer transaction), another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum A + B will be less than it should be).

T1	**T2**
read(A)	
A := A – 50	
write(A)	
	read(A), read(B), print(A+B)
read(B)	
B := B + 50	
write(B	

- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other.
- Executing multiple transactions concurrently has significant benefits.

ACID Properties

- To preserve the integrity of data the database system must ensure:
- **Atomicity -** Either all operations of the transaction are properly reflected in the database or none are.
  - Transaction performed in its entirety or not at all
- **Consistency -** Execution of a transaction in isolation preserves the consistency of the database.
  - Takes database from one consistent state to another
- **Isolation -** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - For every pair of transactions T_i and T_j, it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i finished.

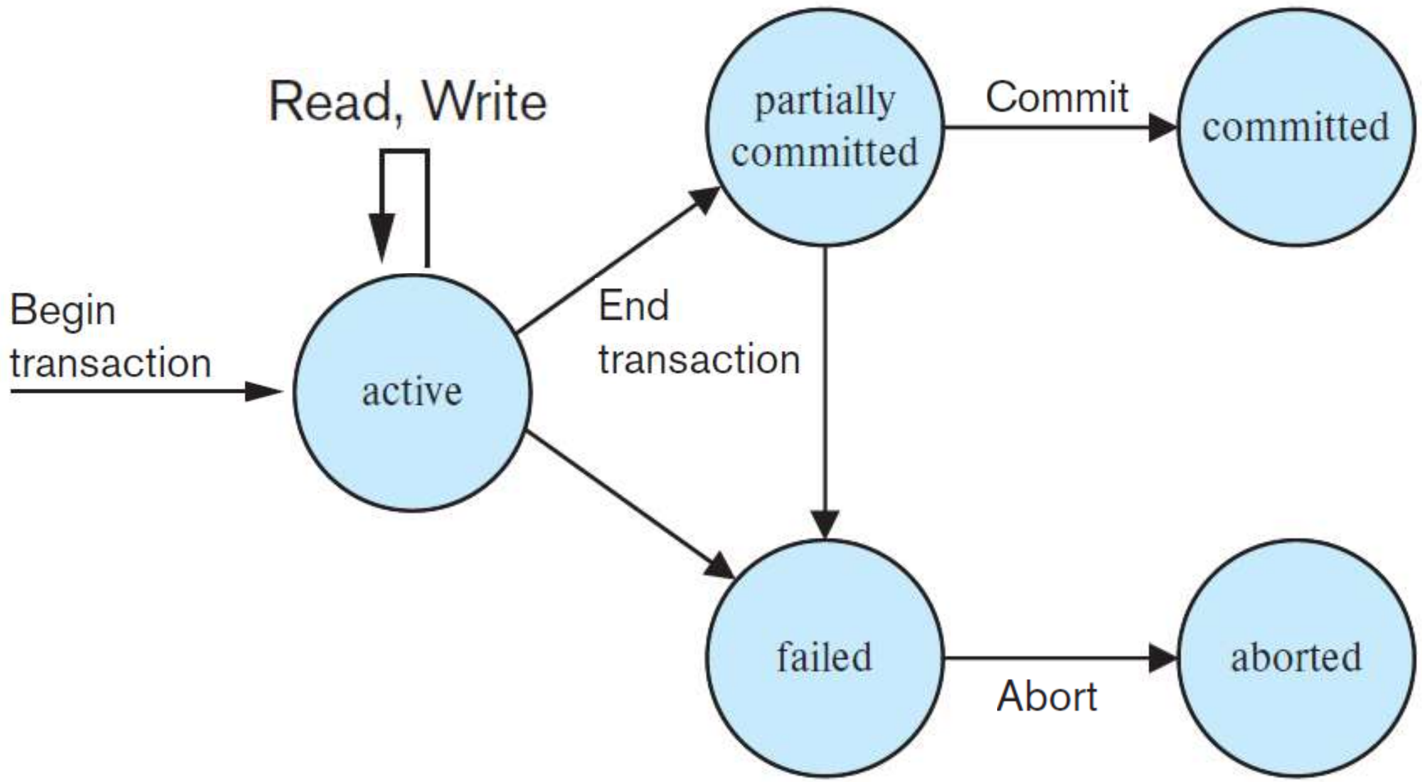
ACID Properties

- **Durability -** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State

- **Active:** The initial state; the transaction stays in this state while it is executing.
- **Partially committed:** After the final statement has been executed.
- **Failed:** After the discovery that normal execution can no longer proceed.
- **Aborted:** After the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - Restart the transaction
    - can be done only if no internal logical error
  - Kill the transaction
- **Committed:** After successful completion.

State diagram of Transaction



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
  - To control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.

Schedules (Histories) of Transactions

- A sequences of instructions that specify the chronological order in which instructions are executed in the system.
- Order of execution of operations from all transactions.
- A schedule for a set of transactions must consist of all instructions of those transactions.
- Operations from different transactions can be interleaved in the schedule.
- Must preserve the order in which the instructions appear in each individual transaction.
- Total ordering of operations in a schedule
  - For any two operations in the schedule, one must occur before the other.

Example of Schedule

Time

T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;
write_item(X); read_item(Y);	
	write_item(X);
$Y := Y + N$; write_item(Y);	

$$S_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$$

Serial Schedule

- Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule.

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

Time

T_1	T_2
read_item(X);	
$X := X - N$;	
write_item(X);	
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	
	read_item(X);
	$X := X + M$;
	write_item(X);

T_1 is followed by T_2

Serial Schedule

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	

Time

T_1	T_2
	read_item(X);
	$X := X + M$;
	write_item(X);
read_item(X);	
$X := X - N$;	
write_item(X);	
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

T_2 is followed by T_1

Nonserial schedule

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

Time

T_1	T_2
read_item(X); $X := X - N$; write_item(X);	
	read_item(X); $X := X + M$; write_item(X);
read_item(Y); $Y := Y + N$; write_item(Y);	

Nonserial schedule

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

Time

T_1	T_2
read_item(X);	
$X := X - N$;	
	read_item(X);
	$X := X + M$;
write_item(X);	
read_item(Y);	
	write_item(X);
$Y := Y + N$;	
write_item(Y);	

Give erroneous result (inconsistent state)

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Serial execution of a set of transactions preserves database consistency.
- **Problem with serial schedules**
  - Limit concurrency by prohibiting interleaving of operations
  - Unacceptable in practice
  - **Solution:** determine which schedules are equivalent to a serial schedule and allow those to occur
- A (possibly concurrent) schedule is **serializable** if it is equivalent to a serial schedule.
- Serializable schedule of *n* transactions
  - Equivalent to some serial schedule of same *n* transactions
- Different forms of schedule equivalence give rise to the notions of:
  - **Conflict serializability**
  - **View serializability**

Conflicting Instructions

- Let us consider a schedule S in which there are two consecutive instructions, I and J, of transactions T_i and T_j, respectively ($i \neq j$).
- If I and J refer to different data items, then we can swap I and J without affecting the results of any instruction in the schedule.
- If I and J refer to the same data item Q, then the order of the two steps may matter.

1. **I = read(Q), J = read(Q).** The order of I and J does not matter, since the same value of Q is read by T_i and T_j, regardless of the order.
2. **I = read(Q), J = write(Q)**. If I comes before J, then T_i does not read the value of Q that is written by T_j in instruction J. If J comes before I, then T_i reads the value of Q that is written by T_j. Thus, the order of I and J matters.
3. **I = write(Q), J = read(Q)**. The order of I and J matters.

Conflicting Instructions

4. **I = write(Q), J = write(Q).** Since both instructions are write operations, the order of these instructions does not affect either T_i or T_j. However, the value obtained by the next read(Q) instruction of S is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write(Q) instruction after I and J in S, then the order of I and J directly affects the final value of Q in the database state that results from schedule S.

- We say that I and J **conflict** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

Conflicting Instructions

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

- write(A) instruction of T_1 conflicts with the read(A) instruction of T_2.
- write(A) instruction of T_2 does not conflict with the read(B) instruction of T_1 because the two instructions access different data items.

Conflicting Instructions

- Let I and J be consecutive instructions of a schedule S.
- If I and J are instructions of different transactions and I and J do not conflict, then we can swap the order of I and J to produce a new schedule S'.
- S is equivalent to S', since all instructions appear in the same order in both schedules except for I and J, whose order does not matter.

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

Conflicting Instructions

- Continue to swap nonconflicting instructions:
  - Swap read(B) instruction of T_1 with read(A) instruction of T_2.
  - Swap write(B) instruction of T_1 with write(A) instruction of T_2.
  - Swap write(B) instruction of T_1 with read(A) instruction of T_2.

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

Conflict Serializability

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

Conflict Serializability

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

Schedule 1

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

Schedule 2

- Schedule 1 can be transformed into Schedule 2 - a serial schedule where T_2 follows T_1, by a series of swaps of non-conflicting instructions.
- Schedule 1 is conflict serializable.

Conflict Serializability

- Example of a schedule that is not conflict serializable:

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

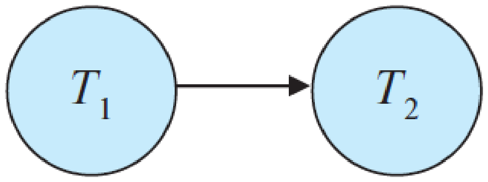
- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

Precedence Graph

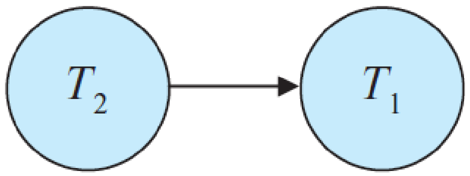
- Consider a schedule S.
- We construct a directed graph, called precedence graph, from S.
- This graph consists of a pair G = (V, E), where V is a set of vertices and E is a set of edges.
- Set of vertices consists of all the transactions participating in the schedule.
- Set of edges consists of all edges $T_i \rightarrow T_j$ for which one of three conditions holds:
  - T_i executes write(Q) before T_j executes read(Q)
  - T_i executes read(Q) before T_j executes write(Q)
  - T_i executes write(Q) before T_j executes write(Q)
- If an edge $T_i \rightarrow T_j$ exists in precedence graph, then, in any serial schedule S′ equivalent to S, T_i must appear before T_j.

Precedence Graph

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

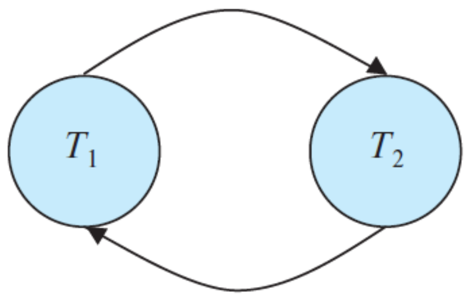


T_1	T_2
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	



Precedence Graph

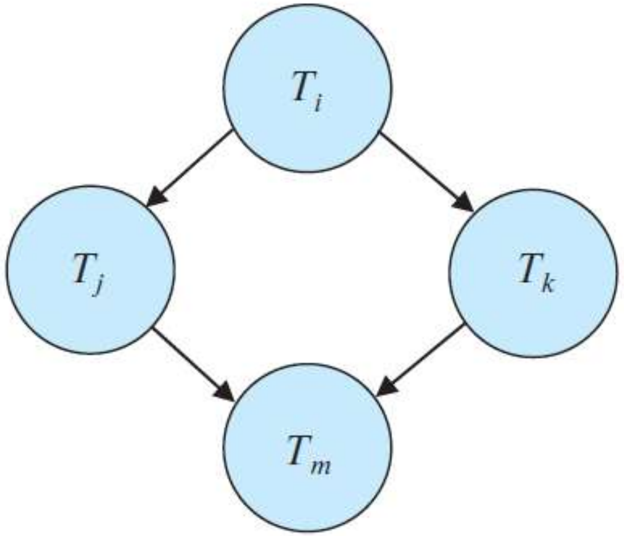
T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A$ - $temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

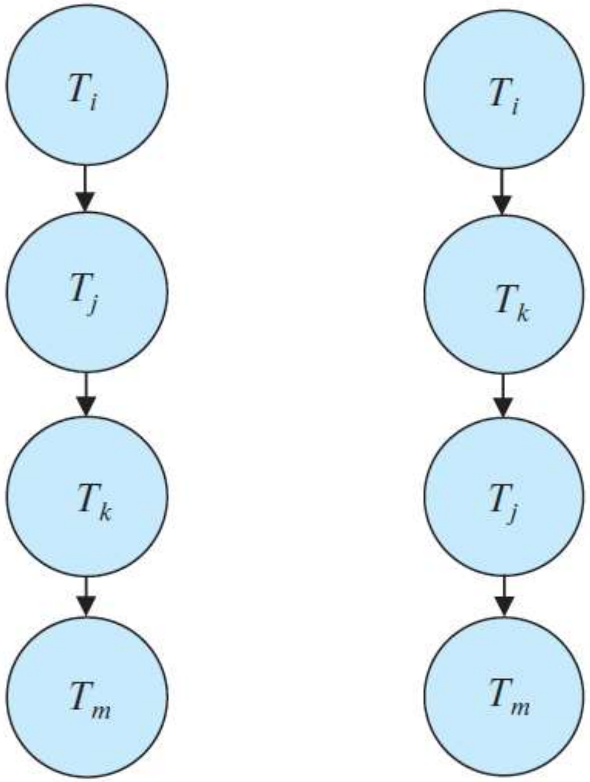


Testing for Conflict Serializability

- If the precedence graph for schedule S has a cycle, then schedule S is not conflict serializable.
- If the graph contains no cycles, then the schedule S is conflict serializable.
- A schedule is conflict serializable if and only if its precedence graph is acyclic.

Serializability Order



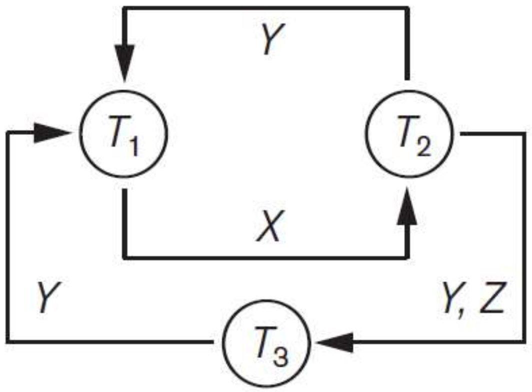


- To find an equivalent serial schedule, start with a node that does not have any incoming edges, and then make sure that the node order for every edge is not violated.

Testing for Conflict Serializability

Time

Transaction T_1	Transaction T_2	Transaction T_3
	read_item(Z);	
	read_item(Y);	
	write_item(Y);	
		read_item(Y);
		read_item(Z);
read_item(X);		
write_item(X);		
		write_item(Y);
		write_item(Z);
	read_item(X);	
read_item(Y);		
write_item(Y);		
	write_item(X);	



Equivalent serial schedules

None

Reason

Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$

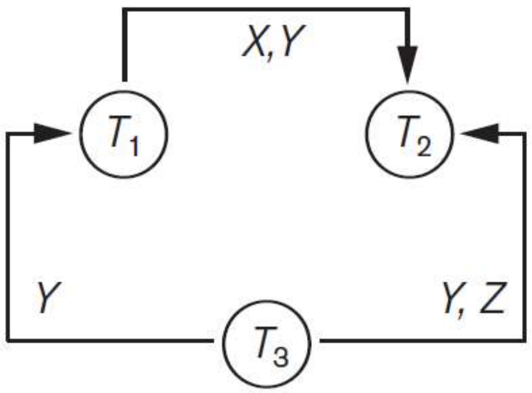
Cycle $X(T_1 \rightarrow T_2), YZ\ (T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

Testing for Conflict Serializability

Time

Transaction T_1	Transaction T_2	Transaction T_3
		read_item(Y); read_item(Z);
read_item(X); write_item(X);		
		write_item(Y); write_item(Z);
	read_item(Z);	
read_item(Y); write_item(Y);		

	read_item(Y);
	write_item(Y);
	read_item(X);
	write_item(X);	



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

Conflict Serializability

T_1	T_2
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)

Conflict Serializable??

Conflict Serializable?

$$r1(X);\ r3(X);\ w1(X);\ r2(X);\ w3(X);$$

$$S1\text{: } r1\ (X);\ r2\ (Z);\ r1\ (Z);\ r3\ (X);\ r3\ (Y);\ w1\ (X);\ w3\ (Y);\ r2\ (Y);\ w2\ (Z);w2\ (Y);$$

Recoverable Schedules

- Recovery is possible.
- Nonrecoverable schedules should not be permitted by the DBMS.
- No committed transaction ever needs to be rolled back.
- **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.

Recoverable Schedules

- ❑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.

Cascading Rollback

- Cascading rollback may occur in some recoverable schedules.
  - Uncommitted transaction may need to be rolled back.
- 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 Schedule

- 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.
- Avoids cascading rollback.
- 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		

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.

Concurrency Control

- Serializable schedule gives benefit of concurrent execution.
  - Without giving up any correctness
- DBMS enforces protocols
  - Set of rules to ensure serializability

View Equivalent

- Let S and S' be two schedules with the same set of transactions and include the same operations of those transactions.
- S and S' are **view equivalent** if the following three conditions are met for each data item Q:
  - 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.*
  - If in schedule S transaction T_i executes read(Q), and that value was produced by transaction T_j (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.
  - 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'.

View Serializability

- A schedule S is **view serializable** if it is view equivalent to a serial schedule.
- Any conflict serializable schedule is also view serializable but not vice versa.
- Below is a schedule which is view-serializable but not conflict serializable.

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

- Every view serializable schedule that is not conflict serializable has **blind writes**.
- Problem of checking if a schedule is view serializable falls in the class of NP-hard.
  - Finding an efficient polynomial time algorithm for this problem is highly unlikely.

Neither Conflict nor View Serializable

- Schedule below produces the same outcome as the serial schedule < T_1, T_5 >, yet is not conflict equivalent or view equivalent to a serial schedule.

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)