



Transaction Management

Instructor : Nitesh Kumar Jha

niteshjha@soa.ac.in

ITER,S'O'A(DEEMED TO BE
UNIVERSITY)

Transaction Concept

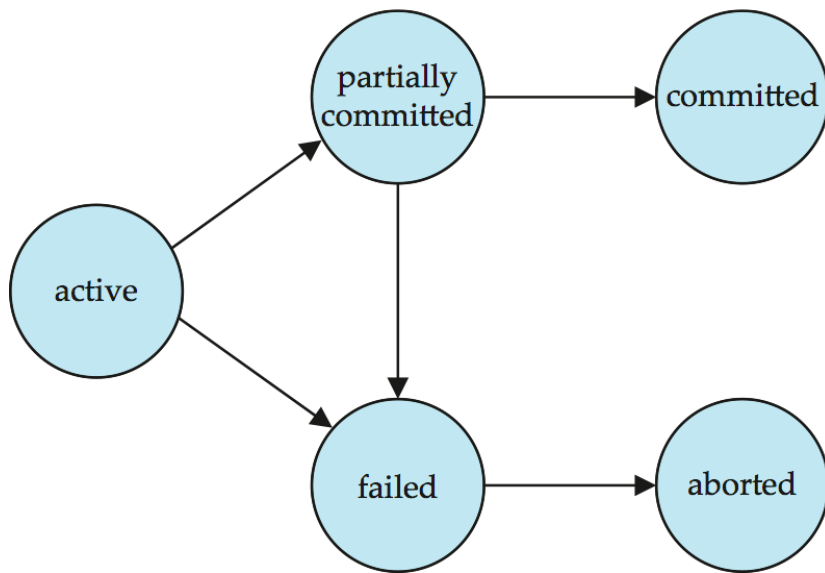
- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g., transaction to transfer Rs. 50 from account A to account B:

1. **read**(A)
2. $A := A - 50$
3. **write**(A)
4. **read**(B)
5. $B := B + 50$
6. **write**(B)

One Transaction consists of a Set of instruction

- Two main issues to deal with:
 - Failures of various kinds, such as hardware failures and system crashes
 - Concurrent execution of multiple transactions

Transaction State - I



- **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 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.

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.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **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.
 - That is, 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.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

ACID Properties w.r.t. a Transaction

■ Atomicity requirement

- If the transaction fails after step 3, money will be “lost” leading to an inconsistent database state
- The system should ensure that updates of a partially executed transaction are not reflected in the database

1. **read(A)**
2. $A := A - 50$
3. **write(A)**
4. **read(B)**
5. $B := B + 50$
6. **write(B)**

- **Durability requirement** —The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.

■ Consistency requirement in above example:

- The sum of A and B is unchanged by the execution of the transaction

Required Properties of a Transaction (Cont.)

- **Isolation requirement** — if between steps 3 and 6, another transaction **T2** is allowed to access the partially updated database

T1

1. **read**(A)
2. $A := A - 50$
3. **write**(A)
4. **read**(B)
5. $B := B + 50$
6. **write**(B)

T2

read(A), read(B), print(A+B)

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

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

Serial Schedule 1

- Let T_1 transfer Rs.50 from A to B, and T_2 transfer 10% of the balance from A to B.
- An example of 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	

Serial execution of transactions
always ensures isolation and
consistence in database

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

Consistency is preserved
i.e. $A+B$ remains same

Serial Schedule 2

- A **serial** schedule in which T_2 is followed by T_1 :

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

Consistency is preserved
irrespective of execution sequence
of both T_1 and T_2

Concurrent Schedule 3

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

- concurrent schedule does not preserve the sum of A+B

Concurrent Schedule 4

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

- the sum “A + B” is preserved
- It is not a serial schedule, but it is **equivalent** to a serial Schedule.
- These schedules are called **serializable** schedules.
- i.e. out of multiple possible concurrent schedules, some may ensure isolation and other may not.
- Hence only the concurrent schedules that ensures isolation and consistency shall be acceptable.

Serializability

- If the final outcome of a concurrent schedule S_1 , is same as that of a serial schedule S_2 , then S_1 is said to be a serializable schedule.
- i.e. A 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 transaction
 - We ignore operations other than **read** and **write** instructions
 - We assume that transactions may perform arbitrary computations in between reads and writes

Conflicting Instructions

- Let I_i and I_j be two Instructions of transactions T_i and T_j respectively. Instructions I_i and I_j **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. $I_i = \text{read}(Q)$, $I_j = \text{read}(Q)$. I_i and I_j don't conflict, order does not matter
 - 2. $I_i = \text{read}(Q)$, $I_j = \text{write}(Q)$. They conflict, as the order matters
 - 3. $I_i = \text{write}(Q)$, $I_j = \text{read}(Q)$. They conflict, as the order matters
 - 4. $I_i = \text{write}(Q)$, $I_j = \text{write}(Q)$. They conflict, order does not affect.

However, the value obtained by the next $\text{read}(Q)$ is affected, since the result of only the latter of the two write instructions is preserved in the database

- 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.
 - i.e. if both I_i and I_j represent read operation, then they can be swapped, but if any one of them is a write operation then they can not be swapped.

Serializability

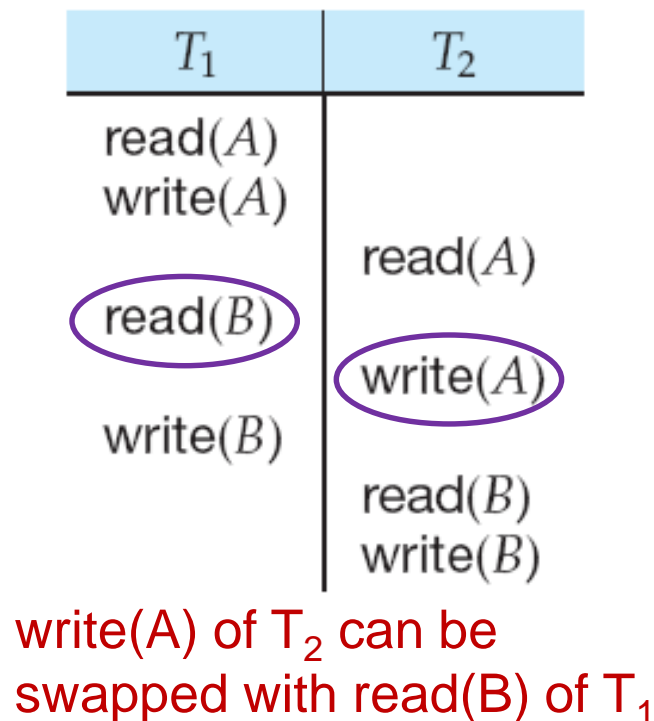
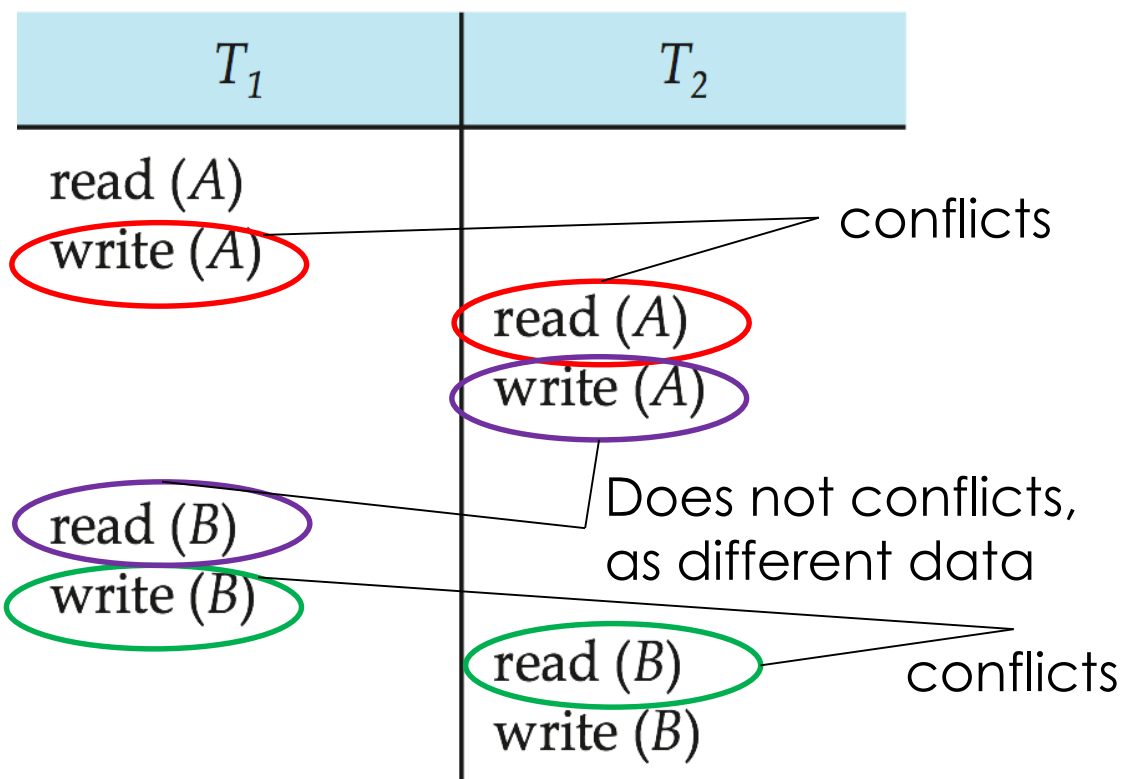
If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent**.

1. **Conflict Serializability:** Two schedules are said to be conflict equivalent if all the conflicting operations in both the schedule get executed in the same order. If a schedule is a conflict equivalent to its serial schedule then it is called Conflict Serializable Schedule.
2. **View Serializability:** Two schedules are said to be view equivalent if the order of initial read, final write and update operations is the same in both the schedules. If a schedule is view equivalent to its serial schedule then it is called View Serializable Schedule.

If a schedule is conflict serializable then it is also view serializable schedule. If a schedule is view serializable then it may or may not be conflict serializable.

Conflict Serializability

- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule



Conflict Serializability (Cont.)

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

- Swap the read(B) instruction of T_1 with the read(A) instruction of T_2 .
- Swap the write(B) instruction of T_1 with the write(A) instruction of T_2 .
- The final result of these swaps is a serial schedule
- i.e. S and S' are **conflict equivalent**
- and hence S is **conflict serializable**

Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

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

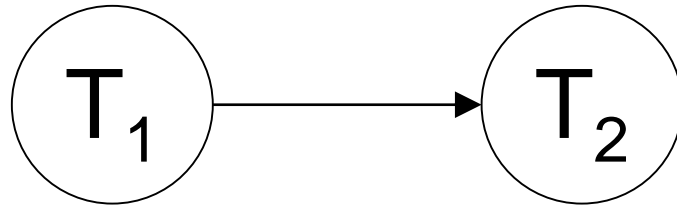
- It is not possible to swap instructions in the above schedule to obtain a serial schedule

Precedence Graph

- It is a simple and efficient method for determining conflict serializability
- Consider schedule S of a set of transactions T_1, T_2, \dots, T_n
- **Precedence graph** — a direct graph $G=(V,E)$,
 - where the vertices are participating transactions.
 - We draw a directed from T_i to T_j if the two transaction conflict, i.e.
 - 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 the precedence graph for S has a **cycle**, then S is **not conflict serializable**

Precedence graph for Schedule 1

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



- Since all instructions of T_1 are executed before the first instruction of T_2 is executed.
- An edge is formed from T_1 to T_2
- As there is no cycle, therefore S_1 is conflict serializable

Concurrent Schedule 4

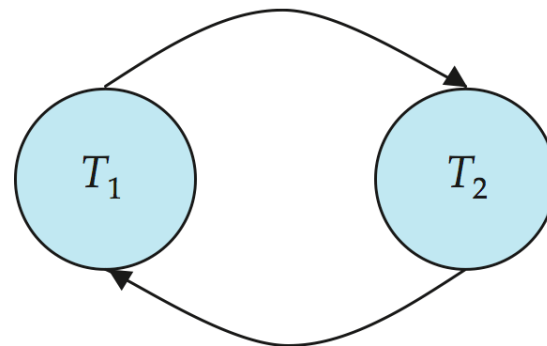
T_1	T_2
read (A) $A := A - 50$ write (A)	read (A) $temp := A * 0.1$ $A := A - temp$ write (A)
read (B) $B := B + 50$ write (B) commit	read (B) $B := B + temp$ write (B) commit



- T_1 executes write(A) before T_2 executes read(A)
- T_1 executes read(B) before T_2 executes write(B)
- T_1 executes write(B) before T_2 executes write(B)
- As there is no cycle, therefore S_4 is conflict serializable

Precedence Graph Schedule 3

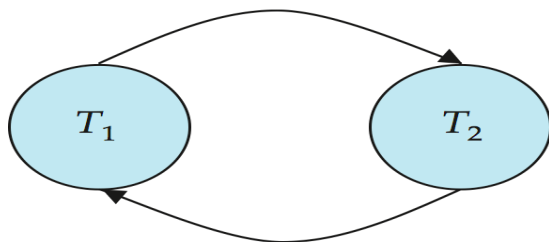
T_1	T_2
<code>read (A)</code> $A := A - 50$	<code>read (A)</code> $temp := A * 0.1$ $A := A - temp$ <code>write (A)</code> <code>read (B)</code>
<code>write (A)</code> <code>read (B)</code> $B := B + 50$ <code>write (B)</code> <code>commit</code>	$B := B + temp$ <code>write (B)</code> <code>commit</code>



- One edge from $T_1 \rightarrow T_2$, as T_1 executes `read(A)`, before T_2 executes `write(A)`
- Another edge $T_2 \rightarrow T_1$, as T_2 executes `read(B)`, before T_1 executes `write(B)`
- As the precedence graph contains a **cycle**, therefore S_3 is **not conflict serializable**

Concurrent Schedule 5

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)



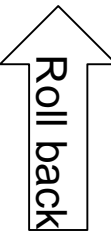
- Test for Conflict serializability
- Test for schedule equivalence
- Precedence Graph
 - T_1 executes write(A) before T_2 executes read(A) (edge from $T_1 \rightarrow T_2$)
 - T_2 executes write(B) before T_1 executes read(B) (edge from $T_2 \rightarrow T_1$)
 - So S5 is not conflict serializable
- Schedule equivalence (A+B)
 - Before transaction, A+B = 1500
 - After transaction, A+B = 1500
 - So S_5 and S_5' (Any schedule equivalent to S_5 by swapping non conflicting instructions) are equivalent schedules
- It is possible to have two schedules that produce same outcome but are not conflict serializable
- Schedule equivalence have **less-stringent** definitions

Recoverability

- If a transaction fails during its execution

Then, the partial executed failed transaction must be rolled back, thereby undoing all its effects as to preserve the Atomicity property.

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



Recoverable schedules

In a concurrent transaction execution failure of transaction requires rolling back of that transaction along with those transaction, which are dependent on failed transaction in order to preserve atomicity.

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 .

- To make this schedule recoverable, T_9 have to delay committing until after T_8 commits

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

- This is a **non recoverable** schedule because
 - If T_8 fails before it commits, then T_9 reads new value of A. i.e. T_9 is dependent on T_8 .
 - Therefore T_9 should also be aborted along with T_8
 - But T_9 already committed with a inconsistent database state.

Cascading Rollbacks

- A single transaction failure may lead to a series of **transaction rollbacks**.

- T_{10} writes A, read by T_{11}
- T_{11} writes A, read by T_{12}
- T_{12} depends on T_{11} and T_{11} depends on T_{10}

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

- Now if T_{10} fails, then T_{11} , and T_{12} has also to be rolled back along with T_{10} due their interdependency.
- If a transaction failure leads to a series of rollbacks, is called **cascading rollback**
- It is undesirable as involves significant amount of work.

Cascadeless Schedules

- It is desirable to restrict the schedules so that cascading rollback can't occur.
- These schedules are called **cascadeless** schedule.
- i.e. 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

End of Chapter
Thank you