Transactions

TRANSACTION

➢A transaction can be defined as a group of tasks. A single task is the minimum processing unit which cannot be divided further.

➢ Example:

A's Account

Open_Account(A)
Old_Balance = A.balance
New_Balance = Old_Balance - 500
A.balance = New_Balance
Close_Account(A)

B's Account

Open_Account(B)
Old_Balance = B.balance
New_Balance = Old_Balance + 500
B.balance = New_Balance
Close_Account(B)

ACID PROPERTIES

A transaction in a database system must maintain **A**tomicity, **C**onsistency, **I**solation, and **D**urability ,commonly known as ACID properties - in order to ensure accuracy, completeness, and data integrity

- **Atomicity** – A transaction must be treated as an atomic unit, (either all of its operations are executed or none) There must be no state in a database where a transaction is left partially completed.

- Example: We have two accounts A and B, each containing Rs 1000/-. We now start a transaction to deposit Rs 100/- from account A to Account B.

ACID PROPERTIES

Read A;
A = A – 100;
Write A;
Read B;
B = B + 100;
Write B;

- After successful execution of transaction it will show Rs 900/- in A and Rs 1100/- in B.

- Suppose there is a power failure just after instruction 3 (Write A) has been completed. What happens now? After the system recovers the database will show Rs 900/- in A, but the same Rs 1000/- in B
- Not acceptable situation

ACID PROPERTIES

- **Consistency – The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database**. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.

- Example: In the internal fund transfer i.e. from account A to account B, the total amount of account A and account B must be same as before the transaction successfully executed

ACID PROPERTIES

- **Durability – 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.

- Example: Let us consider account A has balance of $300. Now $100 have been credited in account A then if power failure occurred after COMMIT then when the system becomes operable again then account A must contain $400.

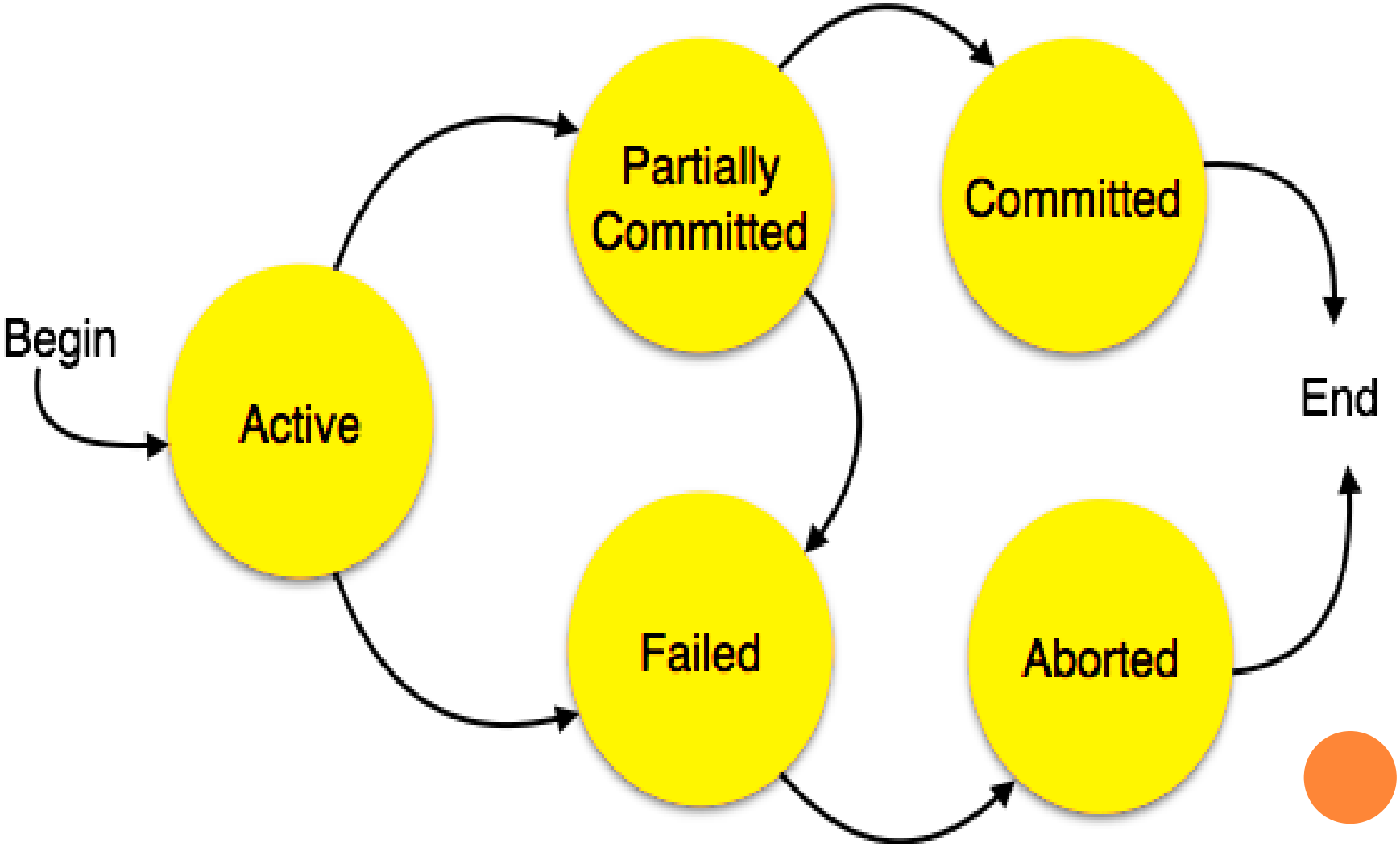
ACID PROPERTIES

- **Isolation** – In a database system where more than one transaction are being executed simultaneously and in parallel(Concurrent Transactions), **the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system**. No transaction will affect the existence of any other transaction.

- Example: For every pair of transactions Ti and Tj , it appears to Ti that either Tj , finished execution before Ti started, or Tj started execution after Ti finished. (Ti followed by Tj or Tj followed by Ti)

States of Transactions

A transaction in a database can be in one of the following states:



States of Transactions

- **Active** − In this state, the transaction is being executed. This is the initial state of every transaction.

- **Partially Committed** − When a transaction executes its final operation, it is said to be in a partially committed state. The values generated during the execution are all stored in volatile storage.

- **Failed** − If the transaction fails for some reason. The temporary values are no longer required, and the transaction is set to **ROLLBACK**. It means that any change made to the database by this transaction up to the point of the failure must be undone. If the failed transaction has withdrawn Rs. 100/- from account A, then the ROLLBACK operation should add Rs 100/- to account A.

States of Transactions

➢**Aborted** − If any of the checks fails and the transaction has reached a failed state, then the recovery manager rolls back all its write operations on the database to bring the database back to its original state where it was prior to the execution of the transaction. Transactions in this state are called aborted.

The database recovery module can select one of the two operations after a transaction aborts −

- ✓ Re-start the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.
- ✓ Kill the transaction It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.

States of Transactions

- **Committed** − If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.

SERIALIZABILITY

When multiple transactions are being executed by the operating system in a multiprogramming environment, there are possibilities that instructions of one transaction are interleaved with some other transaction

➢**Schedule** − **A chronological execution sequence of a transaction is called a schedule**. A schedule can have many transactions in it, each comprising of a number of instructions/tasks.

➢**Serial Schedule** − It is **a schedule in which transactions are aligned in such a way that one transaction is executed first**. When the first transaction completes its cycle, then the next transaction is executed. Transactions are ordered one after the other. This type of schedule is called a serial schedule, as transactions are executed in a serial manner.

Concurrent Transactions

- In a multi-transaction environment, serial schedules are considered as a benchmark. The execution sequence of an instruction in a transaction cannot be changed, but two transactions can have their instructions executed in a random fashion.

- This execution does no harm if two transactions are mutually independent and working on different segments of data.

- But in case these two transactions are working on the same data, then the results may vary.

Checking Serializabity

➢Let $T1$ and $T2$ be two transactions that transfer funds from one account to another. Transaction $T1$ transfers \$50 from account A to account B. It is defined as:

$T1$: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B)

Transaction $T2$ transfers 10 percent of the balance from account A to account B. It is defined as:

$T2$: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B)

CHECKING SERIALIZABITY

➢Suppose the current values of accounts *A* and *B* are $1000 and $2000, respectively. Suppose also that the two transactions are executed one at a time in the order *T*1 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

Figure 1 Schedule 1 – A serial schedule in which T1 is followed by T2

CHECKING SERIALIZABITY

- If the transactions are executed one at a time in the order T2 followed by T1, then the corresponding execution sequence is that of Figure 2.

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	

Figure 2 Schedule 2 – A serial schedule in which a T2 is followed by T1

CHECKING SERIALIZABITY

- One possible schedule appears in Figure 3.

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

Figure 3 Schedule 3 – A concurrent schedule equivalent to schedule 1

Checking Serializabity

- Consider the schedule of Figure 4

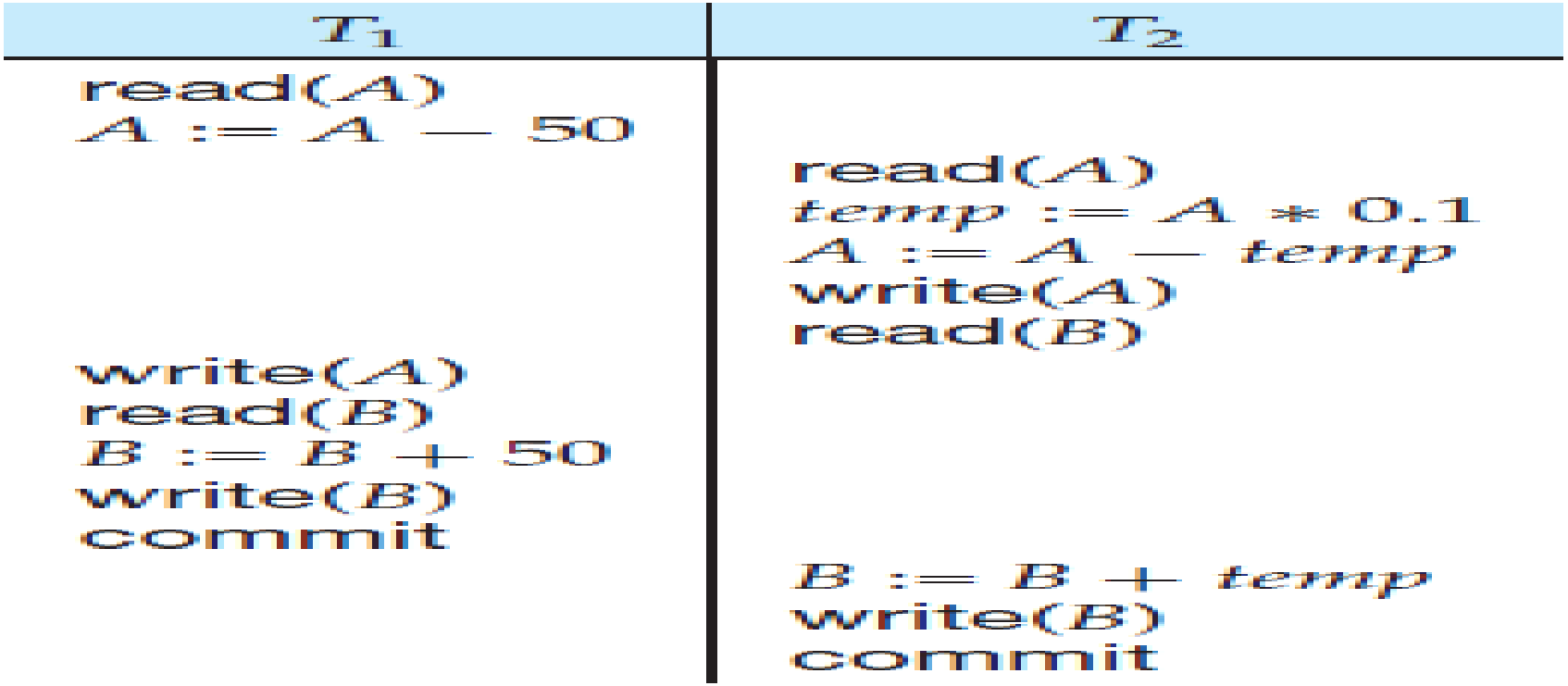


Figure 4 Schedule 4 – A concurrent schedule resulting in an inconsistent state

It is the job of the database system to ensure that any schedule that is executed will leave the database in a consistent state. The concurrency-control component of the database system carries out this task.

Conflict Serializabity

- Serial schedules are serializable, but if steps of multiple transactions are interleaved, it is harder to determine whether a schedule is serializable
- It is difficult to determine exactly what operations a transaction performs and how operations of various transactions interact
- We will consider only two operations: read and write
- Between a **read(Q)** instruction and a **write(Q)** instruction on a data item **Q**, a transaction may perform an arbitrary sequence of operations on the copy of Q that is residing in the local buffer of the transaction

Conflict Serializabity

- Let us consider a schedule S in which there are two consecutive instructions, I and J , of transactions Ti and Tj , 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

- Since we are dealing with only read and write instructions, there are four cases that we need to consider:

Conflict Serializabity

1. I = read(Q), J = read(Q). The order of I and J does not matter, (same value of Q is read by Ti and Tj)

2. I = read(Q), J = write(Q). If I comes before J , then Ti does not read the value of Q that is written by Tj in instruction J . If J comes before I, then Ti reads the value of Q that is written by Tj. (order of I and J matters)

3. I = write(Q), J = read(Q). (order of I and J matters)

4. I = write(Q), J = write(Q). (order doesn't affect Ti & Tj) 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.

Conflict Serializabity

- I and J **conflict** if :

✓**They are operations by different transactions**

✓**Access the same data item and**

✓ **At least one of these instructions is a write operation**

- Consider Schedule 3 in the form of only read and write operations:

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

Schedule 3

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

Schedule 4: Only read and write operations

Conflict Serializabity

- 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
- The write(A) instruction **of T2** in schedule 3 does not conflict with the read(B) instruction **of T1**, we can swap these instructions to generate an equivalent schedule, schedule 5

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

Schedule 4: Only read and write operations

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

Schedule 5: After swapping

Conflict Serializabity

- Schedules 3 and 5 both produce the same final system state.
- We continue to swap non conflicting instructions:
  - ✓ **Swap the read(B) instruction of T1 with the read(A) instruction of T2.**
  - ✓ **Swap the write(B) instruction of T1 with the write(A) instruction of T2**
  - ✓ **Swap the write(B) instruction of T1 with the read(A) instruction of T2**

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: Final Schedule