**Transactions**

Set of operations used to perform a logical unit of work

It represents change in database

**ACID**

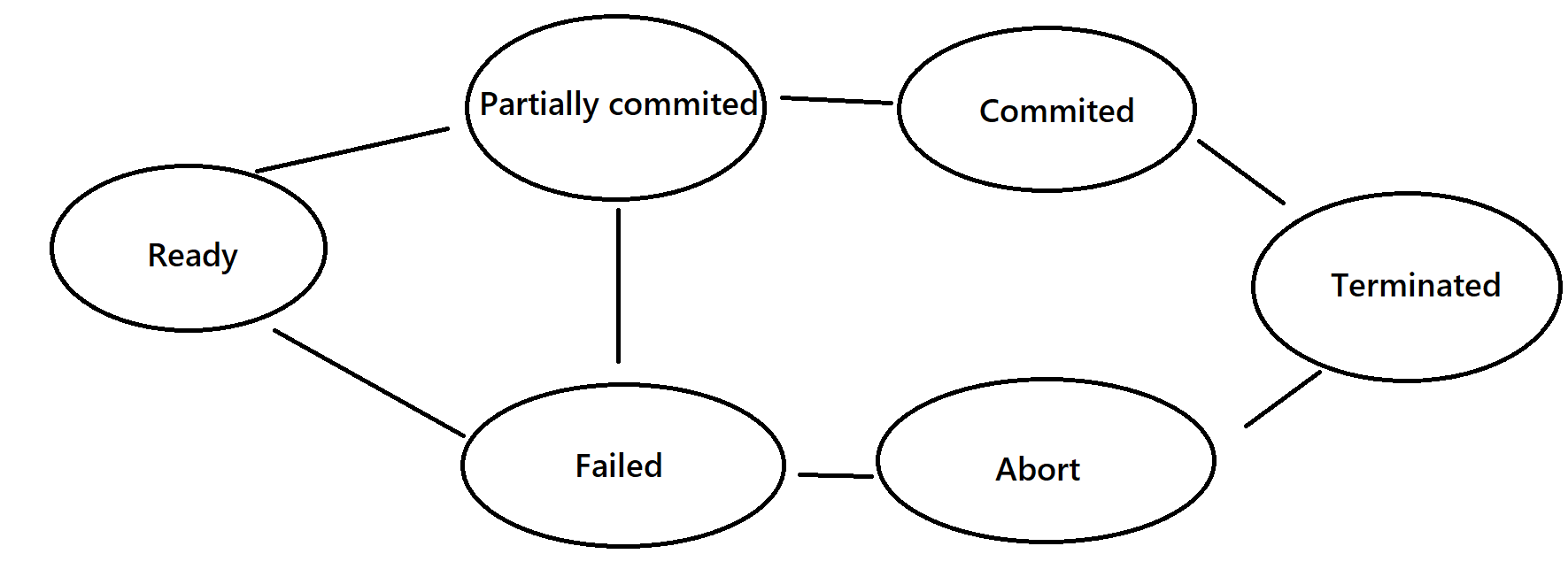
ATOMICITY -> Either all operation or None

CONSISTENCY -> Data is in a consistent state when a transaction starts and when it ends. For example, in an application that transfers funds from one account to another, the consistency property ensures that the total value of funds in both the accounts is the same at the start and end of each transaction.

ISOLATION -> One transaction’s opeartions are not visible to other other transactions

DURABILITY -> Durability is **the property that makes sure that transactions are permanently stored and do not disappear or are erased by accident, even during a database crash**.

**Transaction State**



**Schedule**

Chronological execution sequence of multiple transactions

**Serial schedule**

a serial schedule is a schedule in which all transactions are executed one after the other, in a predetermined order. This means that no two transactions are allowed to run concurrently, and each transaction must wait for the previous one to complete before it can begin

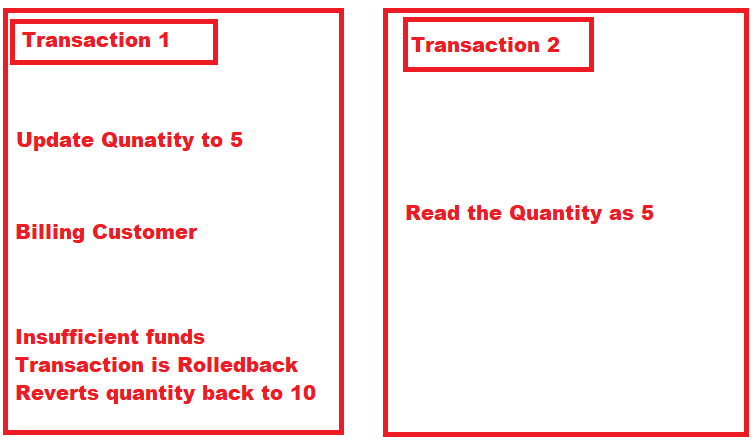
**Parallel Schedule**

a parallel schedule is a schedule in which multiple transactions are allowed to run concurrently, or in parallel. This means that more than one transaction can be in the process of being executed at the same time. A parallel schedule can improve the performance of the database system, because multiple transactions can be processed simultaneously. However, it can also introduce the possibility of conflicts between transactions, if two or more transactions attempt to modify the same data at the same time. To avoid such conflicts, the database system may use locking or other concurrency control mechanisms to ensure that transactions are executed in a consistent and reliable manner.

**Concurrency control problem / Anomalies**

**Dirty Read**

a dirty read is a type of concurrency control anomaly that can occur when multiple transactions are executed concurrently. It refers to the **situation where a transaction reads data that has been written by another transaction, but that has not yet been committed to the database**. This can lead to inconsistencies in the database, because the transaction that reads the data may be based on incorrect or incomplete information.

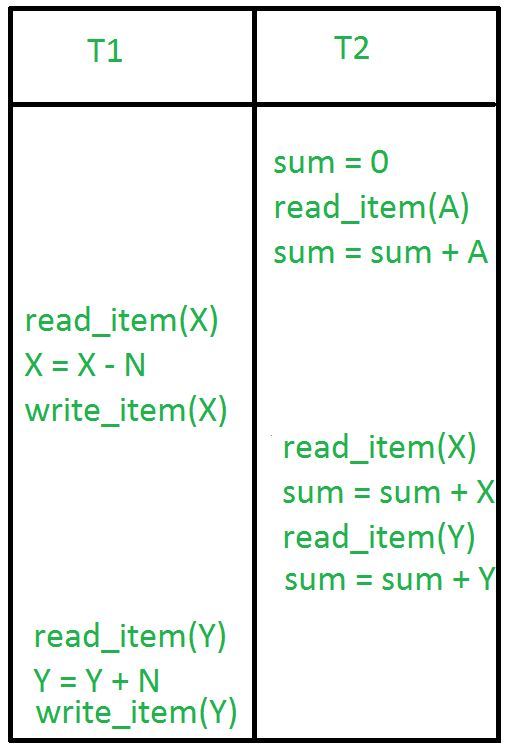


In the above example, if transaction 1 fails for some reason then quantity will revert back to its previous value. But transaction 2 has already read the incorrect value of quantity.

**Incorrect Summary Problem:**

Consider a situation, where one transaction is applying the aggregate function on some records while another transaction is updating these records. The aggregate function may calculate some values before the values have been updated and others after they are updated.

**Example:**

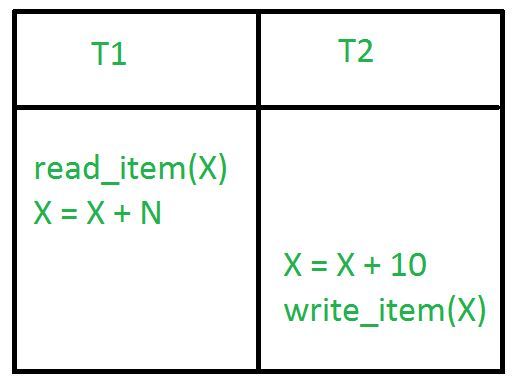


In the above example, transaction 2 is calculating the sum of some records while transaction 1 is updating them. Therefore the aggregate function may calculate some values before they have been updated and others after they have been updated.

**Lost Update Problem:**

In the lost update problem, an update done to a data item by a transaction is lost as it is overwritten by the update done by another transaction.

**Example:**

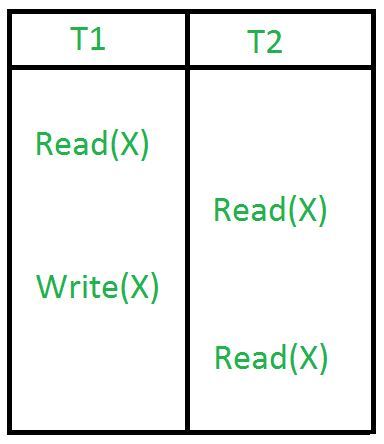


In the above example, transaction 2 changes the value of X but it will get overwritten by the write commit by transaction 1 on X *(not shown in the image above)*. Therefore, the update done by transaction 2 will be lost. Basically, the write commit done by the **last transaction** will overwrite all previous write commits.

**Unrepeatable Read Problem:**

The unrepeatable problem occurs when two or more read operations of the same transaction read different values of the same variable.

**Example:**

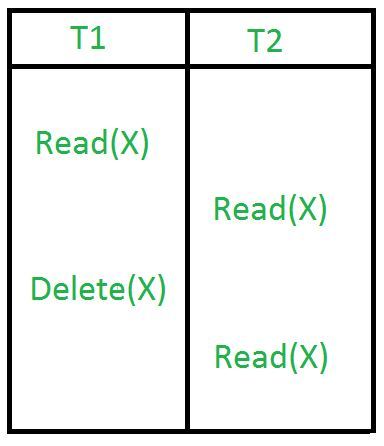


In the above example, once transaction 2 reads the variable X, a write operation in transaction 1 changes the value of the variable X. Thus, when another read operation is performed by transaction 2, it reads the new value of X which was updated by transaction 1.

**Phantom Read Problem:**

The phantom read problem occurs when a transaction reads a variable once but when it tries to read that same variable again, an error occurs saying that the variable does not exist.

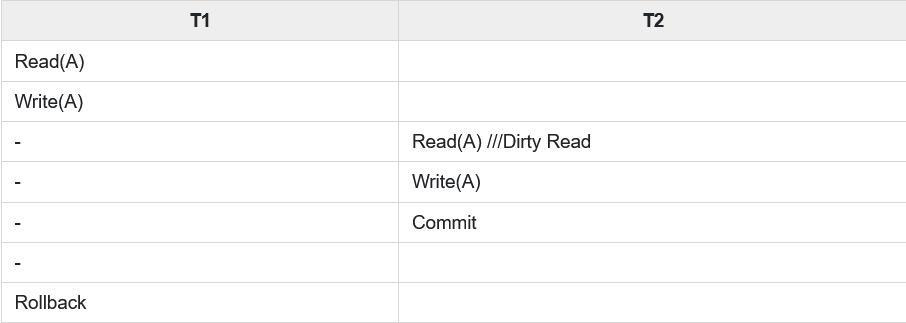
**Example:**



Schedules cont…

## Irrecoverable schedules

If a transaction does a dirty read operation from an uncommitted transaction and commits before the transaction from where it has read the value, then such a schedule is called an irrecoverable schedule.



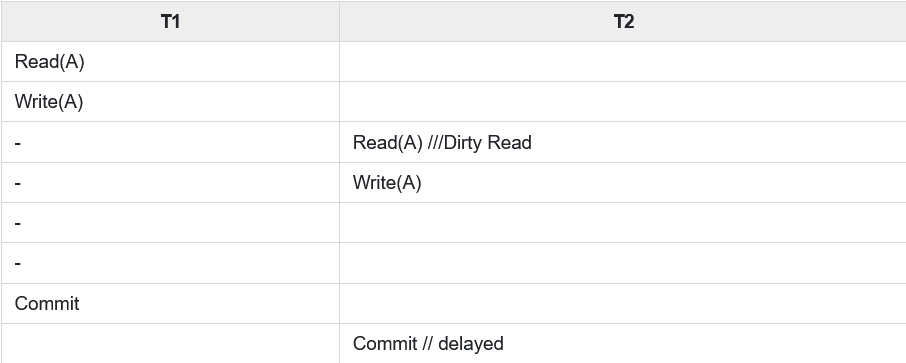
The above schedule is a irrecoverable because of the reasons mentioned below −

* The transaction T2 which is performing a dirty read operation on A.
* The transaction T2 is also committed before the completion of transaction T2.
* The transaction T1 fails later and there are rollbacks.
* The transaction T2 reads an incorrect value.
* Finally, the transaction T2 cannot recover because it is already committed.

## Recoverable Schedules

If any transaction that performs a dirty read operation from an uncommitted transaction and also its committed operation becomes delayed till the uncommitted transaction is either committed or rollback such type of schedules is called as Recoverable Schedules.

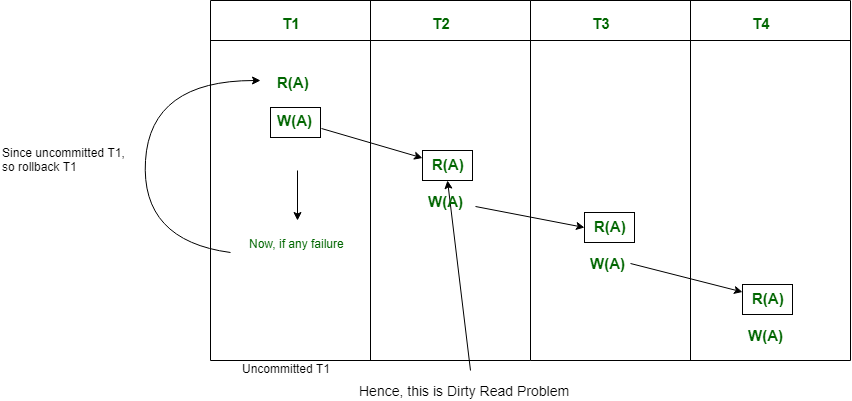
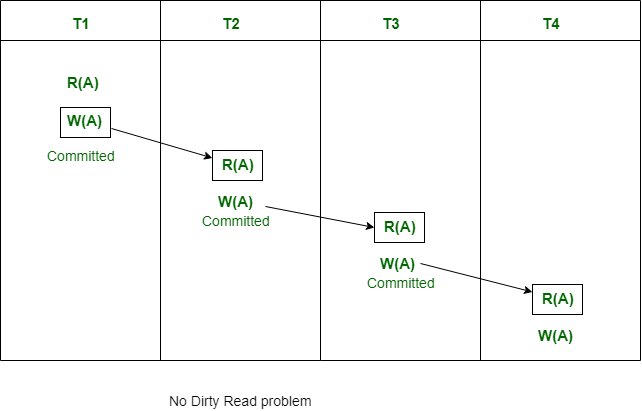
## Example



The above schedule is a recoverable schedule because of the reasons mentioned below −

* The transaction T2 performs dirty read operation on A.
* The commit operation of transaction T2 is delayed until transaction T1 commits or rollback.
* Transaction commits later.
* In the above schedule transaction T2 is now allowed to commit whereas T1 is not yet committed.
* In this case transaction T1 is failed, and transaction T2 still has a chance to recover by rollback.

**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 Rollback or Cascading Abort or Cascading Schedule. It simply leads to the wastage of CPU time.  
These Cascading Rollbacks occur because of **Dirty Read problems**.

* For example, transaction T1 writes uncommitted x that is read by Transaction T2. Transaction T2 writes uncommitted x that is read by Transaction T3.  
  Suppose at this point T1 fails.  
  T1 must be rolled back, since T2 is dependent on T1, T2 must be rolled back, and since T3 is dependent on T2, T3 must be rolled back.
*   
    
    
  Because of T1 rollback, all T2, T3, and T4 should also be rollback (Cascading dirty read problem).
* This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks is called **Cascading rollback**.
* **Cascadeless Schedule:**  
  This schedule avoids all possible *Dirty Read Problem*.
* In Cascadeless Schedule, if a transaction is going to perform read operation on a value, it has to wait until the transaction who is performing write on that value commits. That means there must not be **Dirty Read**. Because Dirty Read Problem can cause *Cascading Rollback*, which is inefficient.
* Cascadeless Schedule avoids cascading aborts/rollbacks (ACA). Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called cascadeless schedules. Avoids that a single transaction abort leads to a series of transaction rollbacks. A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.
* In other words, if some transaction Tj wants to read value updated or written by some other transaction Ti, then the commit of Tj must read it after the commit of Ti.
* 
* Note that Cascadeless schedule allows only committed read operations. However, it allows uncommitted write operations.
* Also note that Cascadeless Schedules are always recoverable, but all recoverable transactions may not be Cascadeless Schedule.

**Conflict Equivalent**

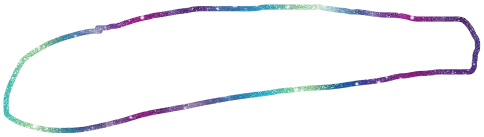
Two schedules are said to be conflict equivalent if the order of any two conflicting operations are the same in both the schedules.

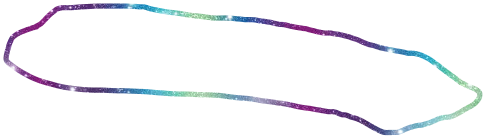
The pre-requisite conditions for conflicting operations are −

* The two conflicting operations should belong to two different transactions.
* They should be acting over the same database or variable say x.
* At least one of the operations should be "Write". For example, operations should be like Read-write; Write-Write; Write-Read

**S1 S2**

**R(X)**

** R(X)**

**W(X)**

**W(X)**

\*\*Conflicting

Now swap the positions

S1 S2

R(X)

W(X)

R(X)

W(X)

**Conflict serilizability**

[**https://www.youtube.com/watch?v=zv0ba0Iok1Y&list=PLxCzCOWd7aiFAN6I8CuViBuCdJgiOkT2Y&index=84**](https://www.youtube.com/watch?v=zv0ba0Iok1Y&list=PLxCzCOWd7aiFAN6I8CuViBuCdJgiOkT2Y&index=84)