TRANSACTION

- Why we study transaction?

In normal computer systems (like operating systems), programs run **instruction by instruction**. So, if your computer crashes in the middle of a program, **only part of the program may have run** — leading to problems like data loss or inconsistency. That's why in databases, we study **transactions** — they make sure that even if something goes wrong (power failure, crash), **either everything happens or nothing does**.

- But in DBMS view, user perform a logical work(operation) which is always atomic in nature i.e. either operation is execute or not executed, there is no concept like partial execution. For example, Transaction T_1 which transfer 100 units from account A to B.

- In this transaction if a failure occurs after Read(B) then the final statue of the system will be inconsistent as 100 units are debited from account A but not credited in account B, this will generate inconsistency. Here for 'consistency' before (A + B) == after (A + B)".

T_1
Read(A)
A = A-100
Write(A)
Read(B)
B = B+100
Write(B)

What is transaction

- To remove this partial execution problem, we increase the level of atomicity and bundle all the instruction of a logical operation into a unit called transaction.
- So formally ‘A transaction is a Set of logically related instructions to perform a logical unit of work’.

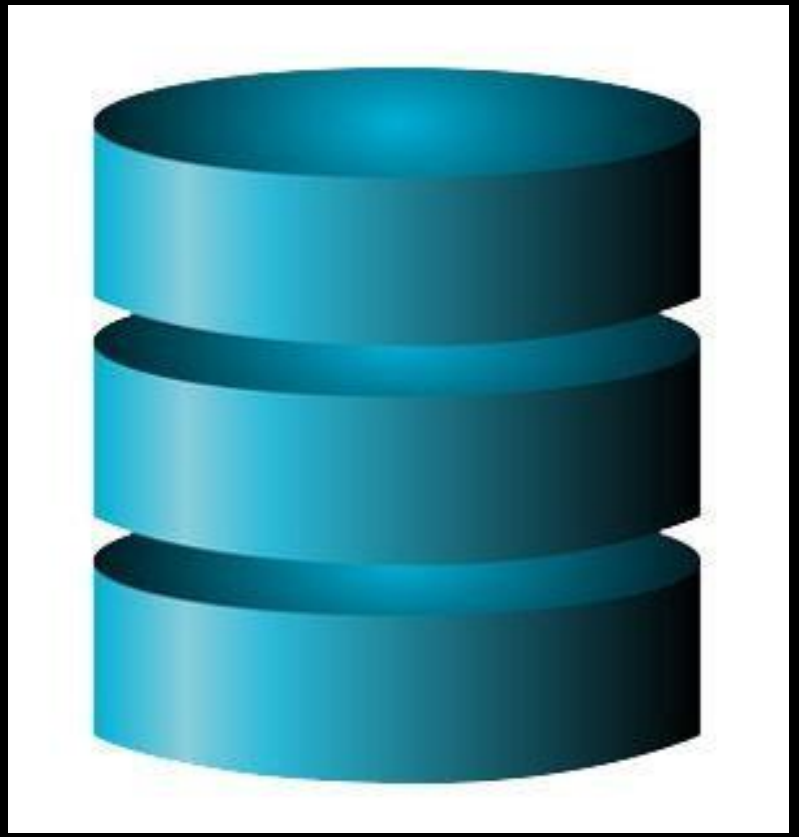
T_1
Read(A)
A = A-100
Write(A)
Read(B)
B = B+100
Write(B)

- As here we are only concerned with DBMS so we well only two basic operation on database.
- **READ (X)** - Accessing the database item x from disk (where database stored data) to memory variable also name as X.
- **WRITE (X)** - Writing the data item from memory variable X to disk.

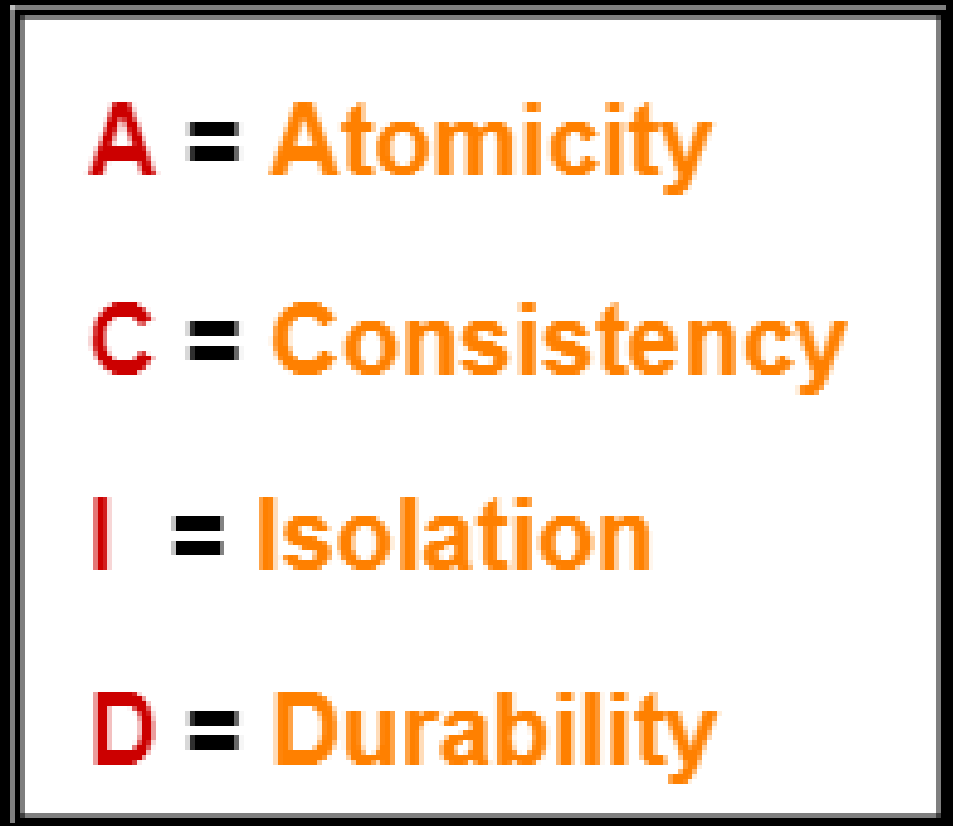
Desirable properties of transaction

- Now as the smallest unit which have atomicity in DBMS view is transaction, so if want that our data should be consistent then instead of concentrating on data base, we must concentrate on the transaction for our data to be consistent.

T_1
Read(A)
A = A-100
Write(A)
Read(B)
B = B+100
Write(B)



- Transactions should possess several properties, often called the **ACID** properties; to provide integrity and consistency of the data in the database. The following are the ACID properties:



- **Atomicity -** A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.

T_1
Read(A)
A = A-100
Write(A)
Read(B)
B = B+100
Write(B)

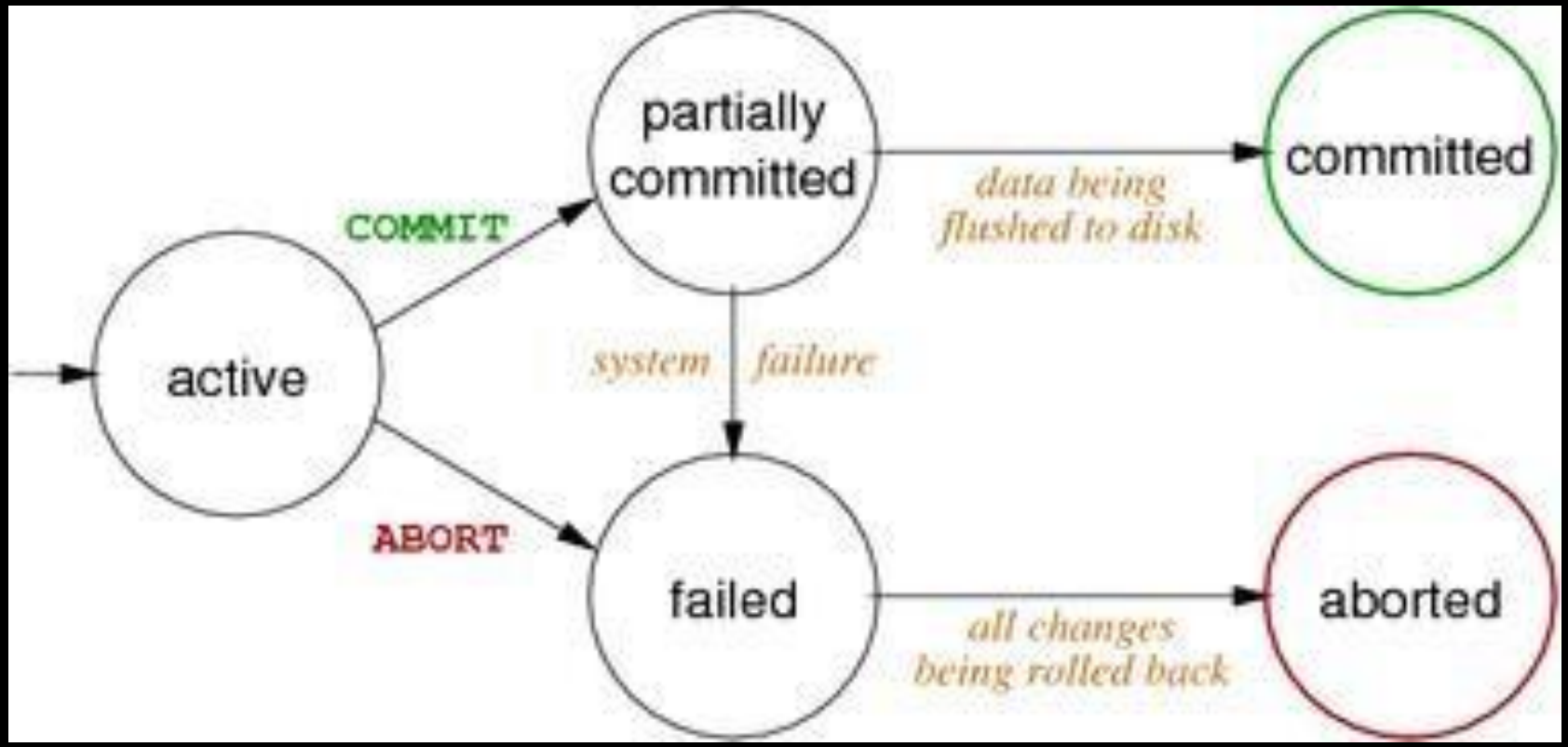
- **<u>Consistency</u>** - A transaction should be preserving consistency, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.

- **Isolation -** A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently.
- That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.

- **Durability -** The changes applied to the database by a committed transaction must persist in the database.

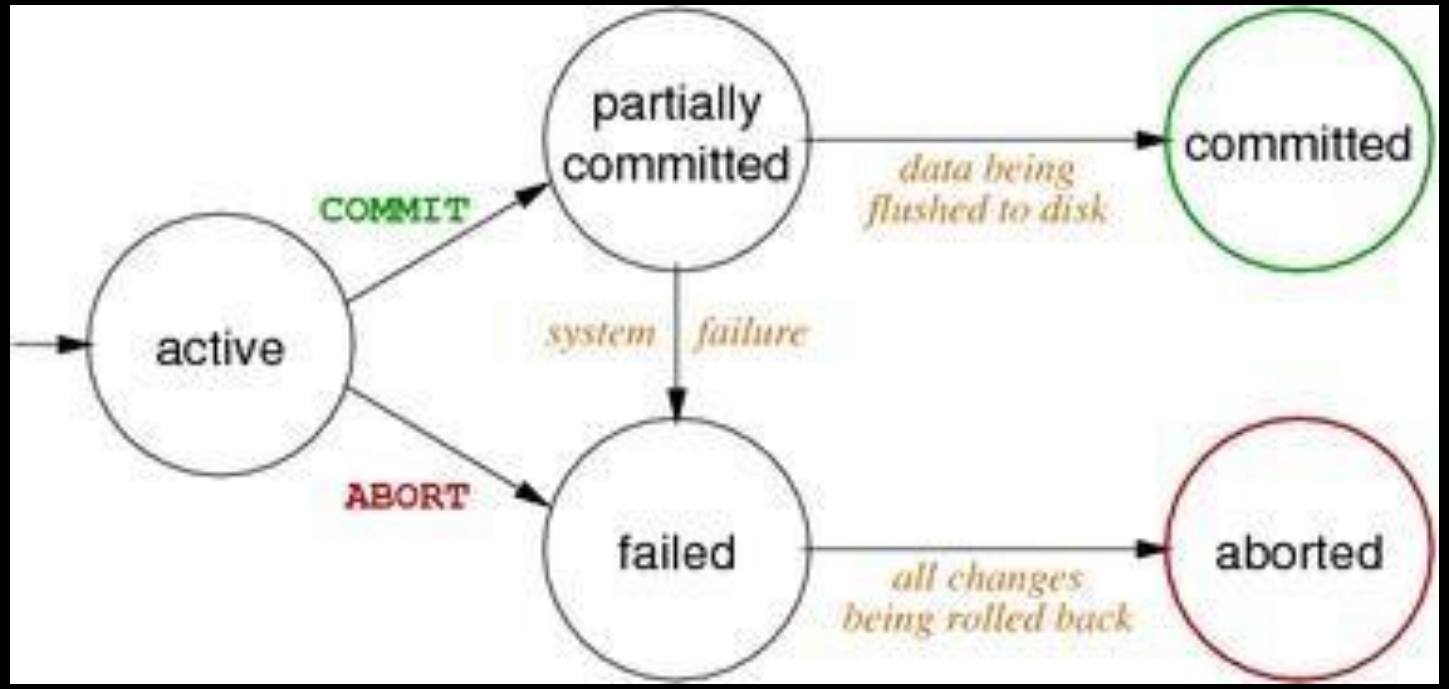
Transaction states

- **ACTIVE** - It is the initial state. Transaction remains in this state while it is executing operations.



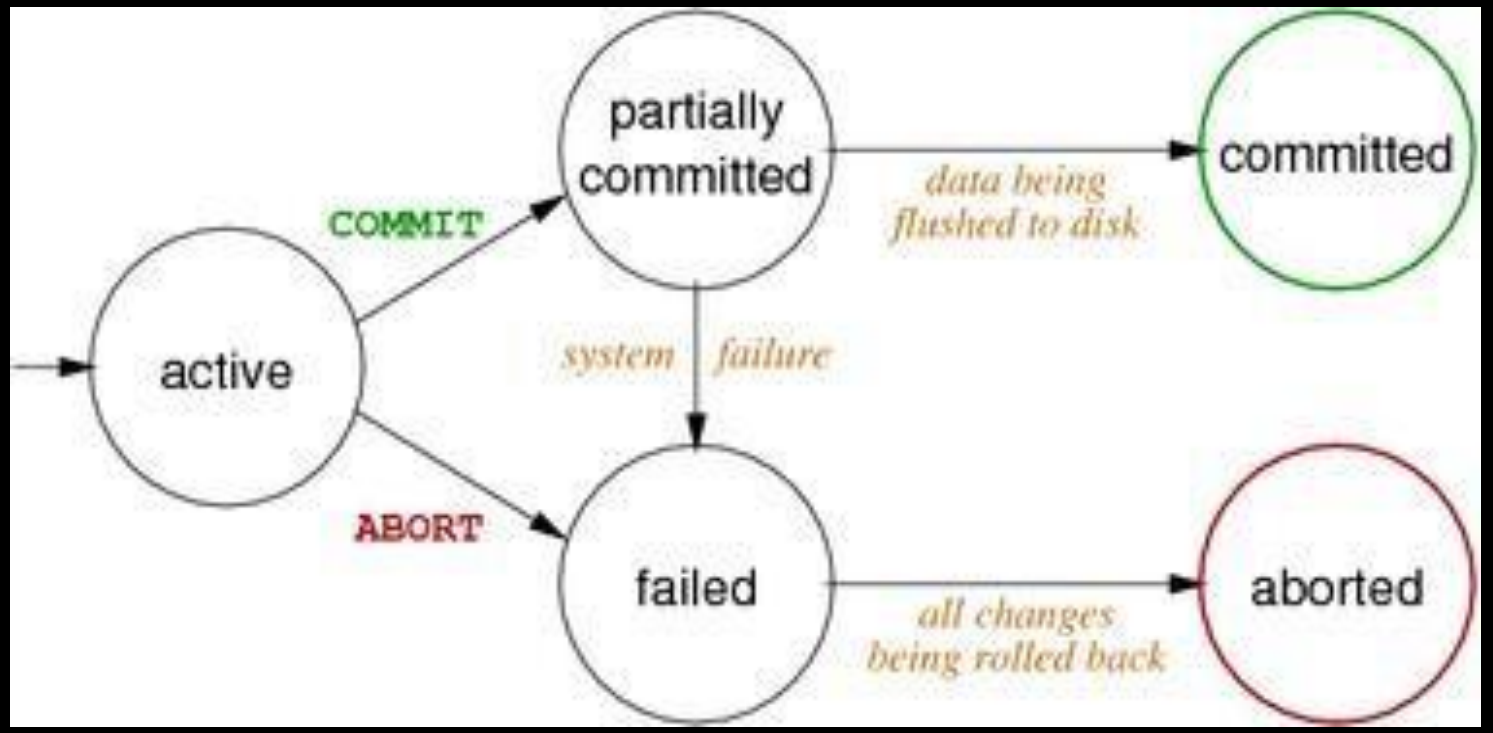
Transaction states

- **PARTIALLY COMMITTED** - After the final statement of a transaction has been executed, the state of transaction is partially committed as it is still possible that it may have to be aborted (due to any failure) since the actual output may still be temporarily residing in main memory and not to disk.



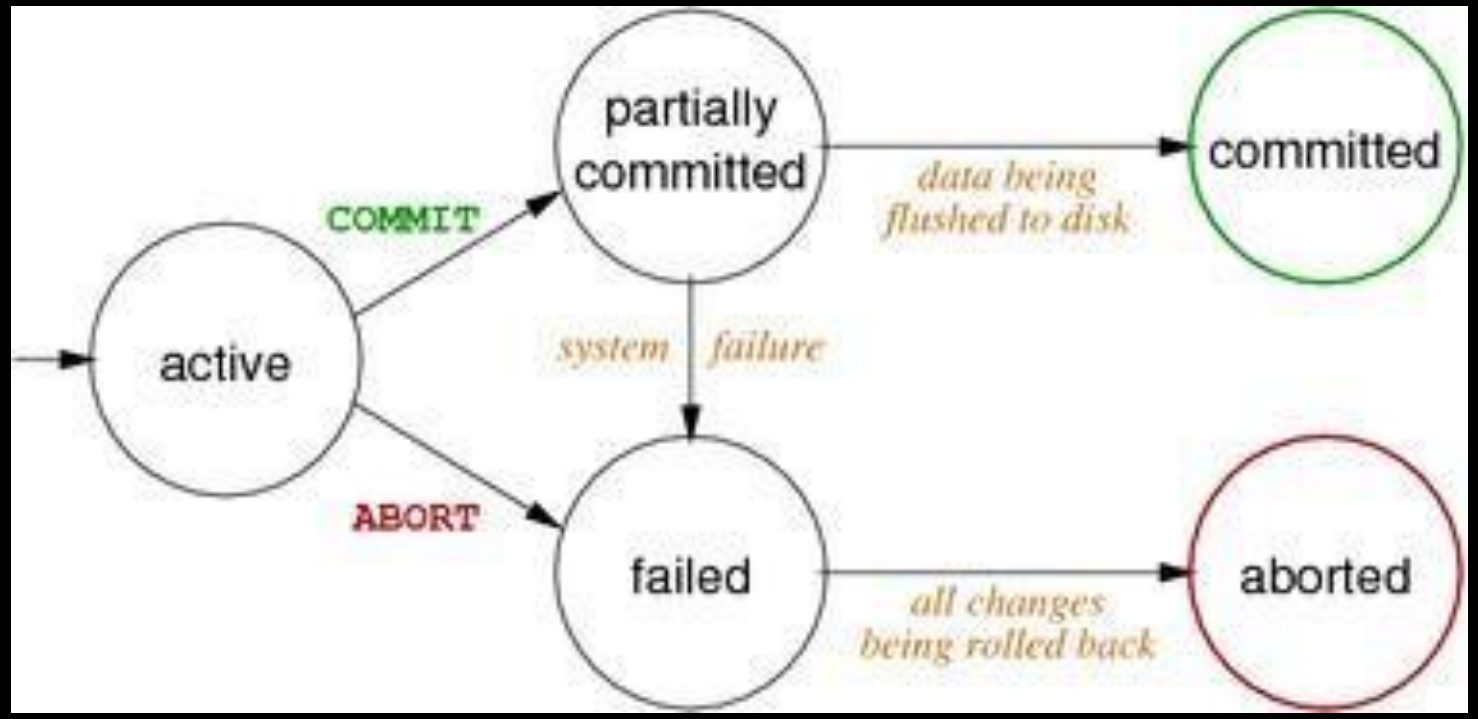
Transaction states

- **FAILED** - After the discovery that the transaction can no longer proceed (because of hardware /logical errors). Such a transaction must be rolled back.



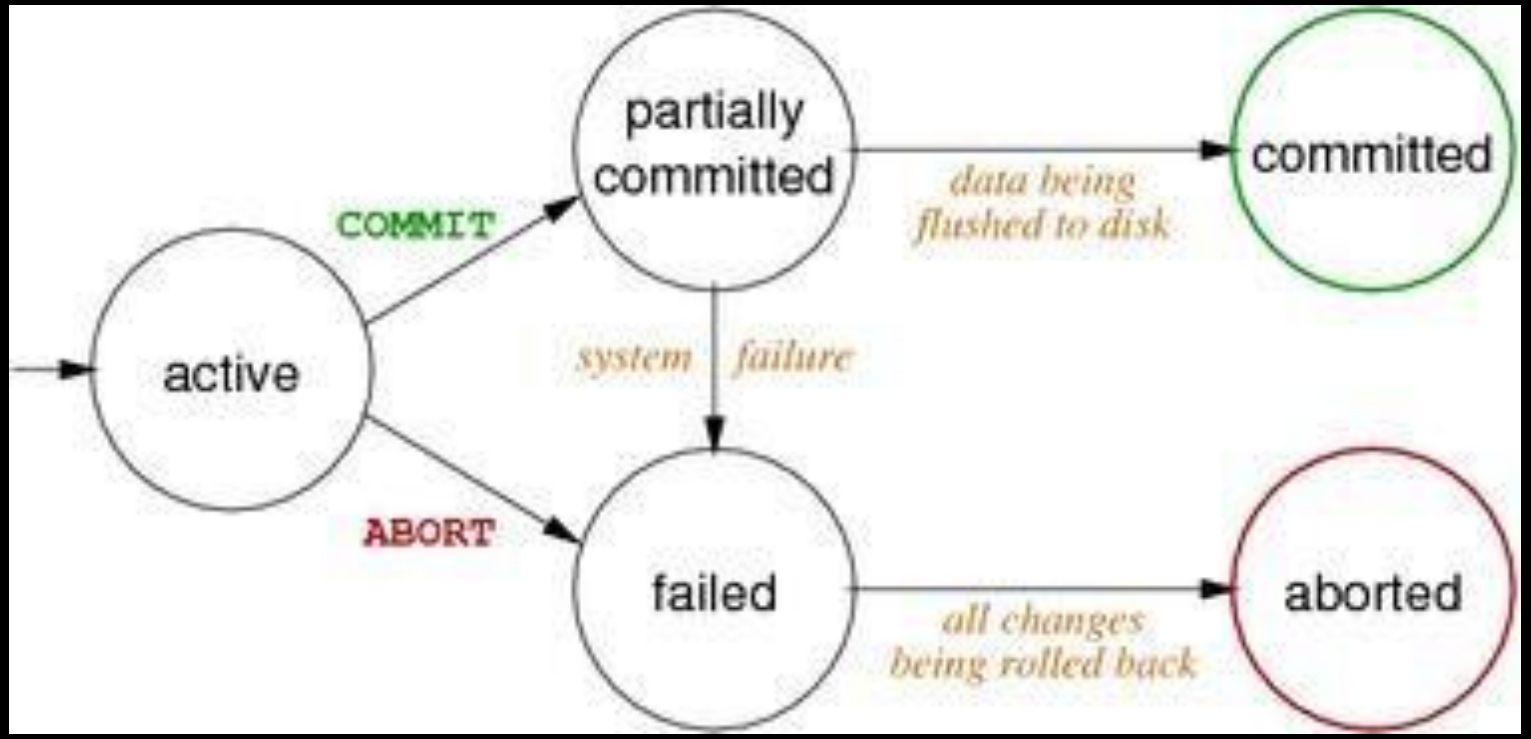
Transaction states

- **ABORTED** - A transaction is said to be in aborted state when the when the transaction has been rolled back and the database has been restored to its state prior to the start of execution.



Transaction states

- **COMMITTED** - A transaction enters committed state after successful completion of a transaction and final updation in the database.



Why we need concurrent execution

- Concurrent execution is necessary because-
  - It leads to good database performance , less weighting time.
  - Overlapping I/O activity with CPU increases throughput and response time.

PROBLEMS DUE TO CONCURRENT EXECUTION OF TRANSACTION

- But interleaving of instructions between transactions may also lead to many problems that can lead to inconsistent database.
- Sometimes it is possible that even though individual transaction are satisfying the acid properties even though the final statues of the system will be inconsistent.

🏦 Real-Life Example: Bank Account

Assume:

- **Account A balance = ₹1000**
- Two transactions happen at same time:
  - **T1**: Withdraw ₹500
  - **T2**: Deposit ₹200

Expected Outcomes (if done one after the other):

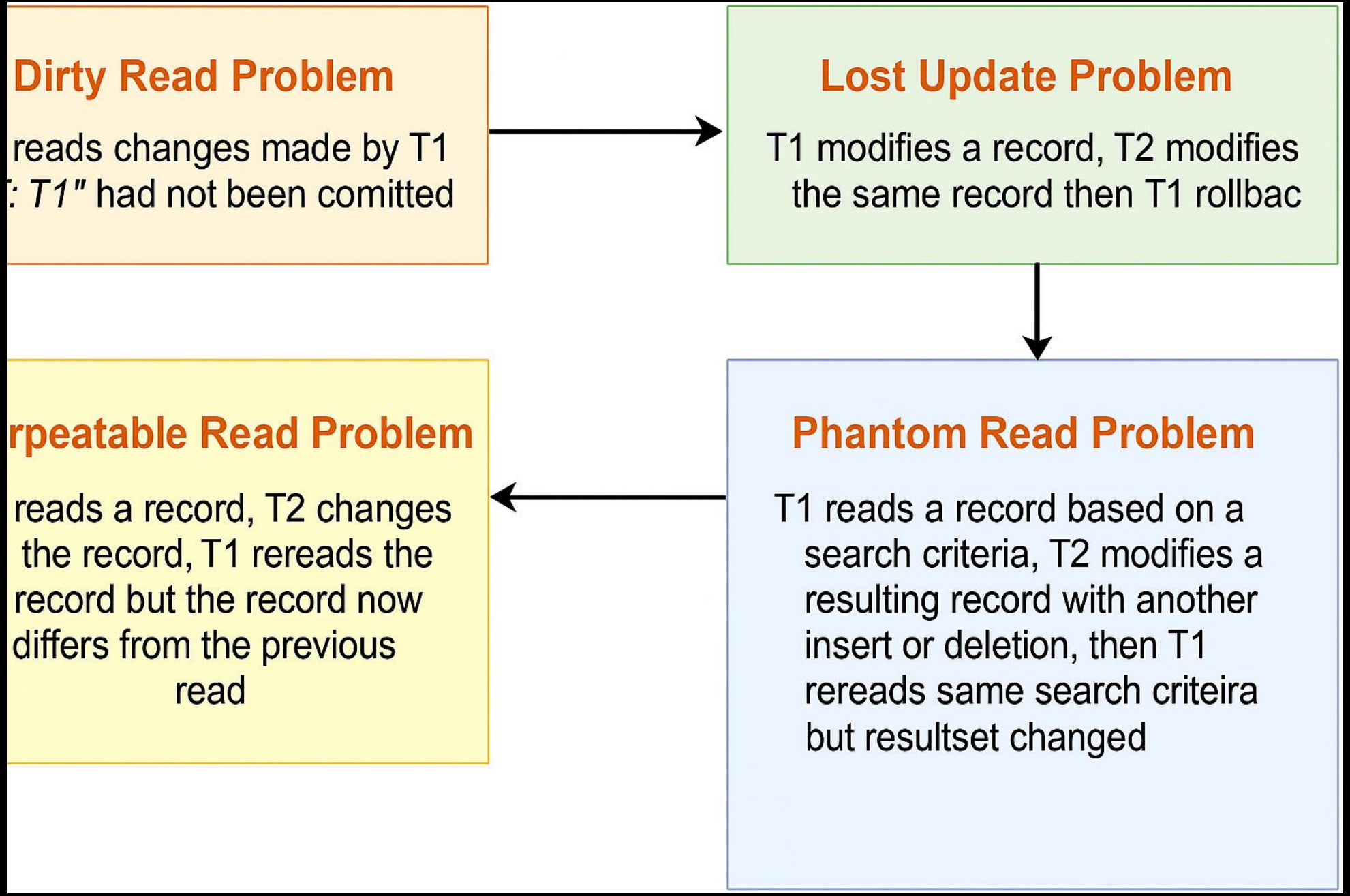
- If T1 first: ₹1000 - ₹500 = ₹500 → then T2: ₹500 + ₹200 = **₹700**
- If T2 first: ₹1000 + ₹200 = ₹1200 → then T1: ₹1200 - ₹500 = **₹700**

In both cases, final balance = ₹700 ✅

😳 What if they interleave wrongly?

- T1 reads balance: ₹1000
- T2 reads balance: ₹1000
- T1 withdraws ₹500 → wants to update balance to ₹500
- T2 deposits ₹200 → wants to update balance to ₹1200
- **T2 saves ₹1200**, overwriting T1's ₹500 update!

❌ **Final balance: ₹1200 (should be ₹700)**


Dirty Read Problem
reads changes made by T1
: T1" had not been comitted
Lost Update Problem
T1 modifies a record, T2 modifies
the same record then T1 rollbac
rpeatable Read Problem
reads a record, T2 changes
the record, T1 rereads the
record but the record now
differs from the previous
read
Phantom Read Problem
T1 reads a record based on a
search criteria, T2 modifies a
resulting record with another
insert or deletion, then T1
rereads same search criteira
but resultset changed

Solution is Schedule

- When two or more transaction executed together or one after another then they can be bundled up into a higher unit of execution called schedule.

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

- **Serial schedule** - A serial schedule consists of sequence of instruction belonging to different transactions, where instructions belonging to one single transaction appear together. Before complete execution of one transaction another transaction cannot be started. Every serial schedule lead database into consistent state.

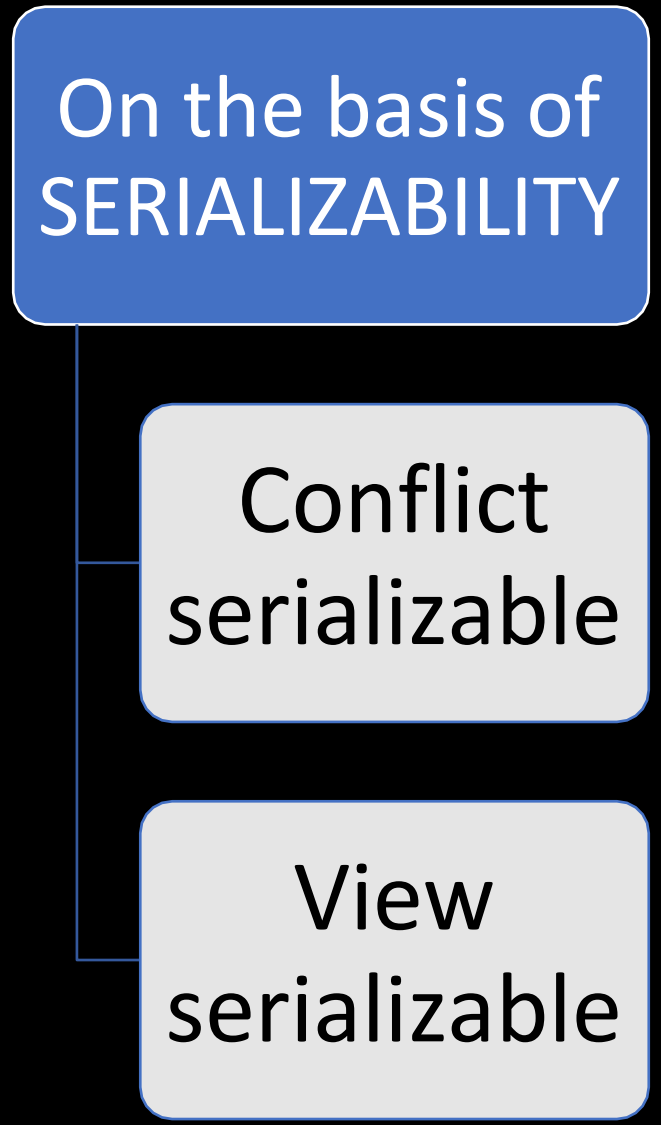
T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

- **Non-serial schedule** - A schedule in which sequence of instructions of a transaction appear in the same order as they appear in individual transaction but the instructions may be interleaved with the instructions of different transactions i.e. concurrent execution of transactions takes place.

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

- **Conclusion of schedules**
  - We do not have any method to proof that a schedule is consistent, but from the above discussion we understand that a serial schedule will always be consistent.

  - So if somehow we proof that a non-serial schedule will also have same effects as of a serial schedule than we can proof that, this particular non-serial schedule will also be consistent.


On the basis of SERIALIZABILITY
Conflict serializable
View serializable

T_1	T_2
R(A)	
	R(B)

T_1	T_2
	R(B)
R(A)	

T_1	T_2
R(A)	
	W(A)

T_1	T_2
	W(A)
R(A)	

T_1	T_2
R(A)	
	W(B)

T_1	T_2
	W(B)
R(A)	

T_1	T_2
	R(A)
W(A)	

T_1	T_2
W(A)	
	R(A)

T_1	T_2
R(A)	
	R(A)

T_1	T_2
	R(A)
R(A)	

T_1	T_2
	W(A)
W(A)	

T_1	T_2
W(A)	
	W(A)

Conflict equivalent – if one schedule can be converted to another schedule by swapping of non- conflicting instruction then they are called conflict equivalent schedule.

T_1	T_2
R(A)	
A=A-50	
	R(B)
	B=B+50
R(B)	
B=B+50	
	R(A)
	A=A+10

T_1	T_2
	R(B)
	B=B+50
R(A)	
A=A-50	
R(B)	
B=B+50	
	R(A)
	A=A+10

SERIALIZABILITY

- ***Conflicting instructions*** **-** Let I and J be two consecutive instructions belonging to two different transactions T_i and T_j in a schedule S, the possible I and J instruction can be as-
  - I= READ(Q), J=READ(Q) ->*Non-conflicting*
  - I= READ(Q), J=WRITE(Q) ->*Conflicting*
  - I= WRITE(Q), J=READ(Q) ->*Conflicting*
  - I= WRITE(Q), J=WRITE(Q) ->*Conflicting*

- So, the instructions I and J are said to be conflicting, if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

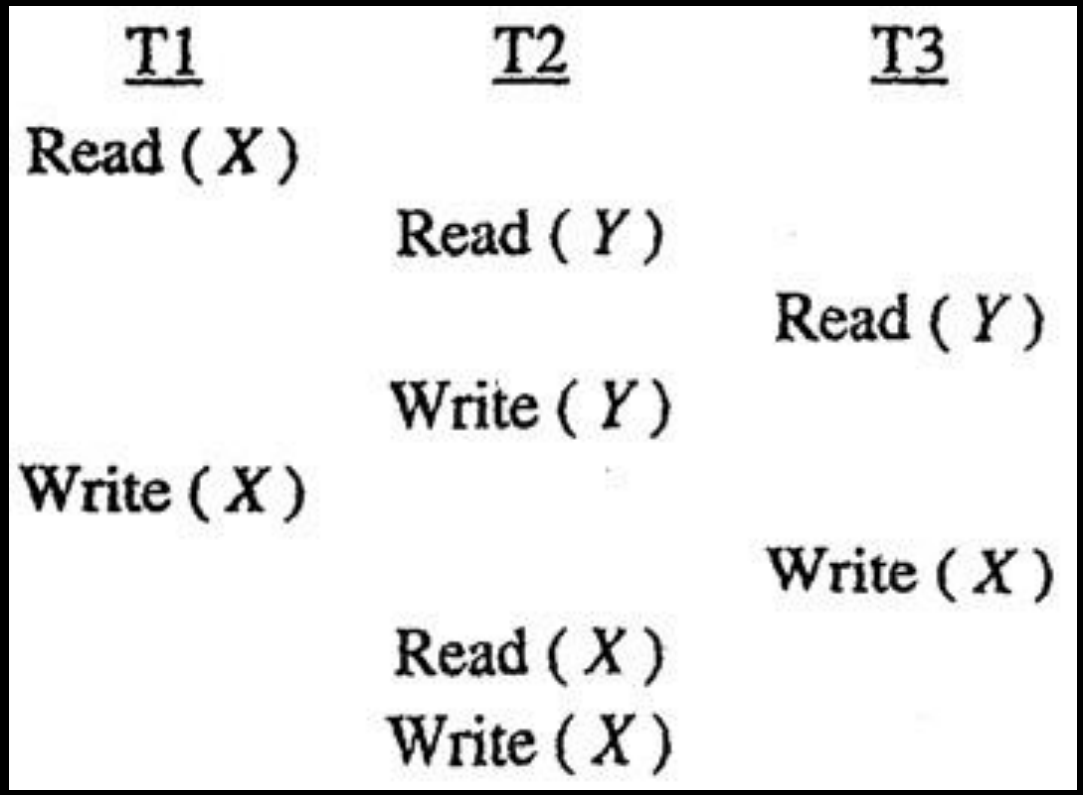
CONFLICT SERIALIZABLE

- The schedules which are conflict equivalent to a serial schedule are called conflict serializable schedule.
- 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.*** A schedule S is ***conflict serializable***, if it is conflict equivalent to a serial schedule.

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

Q Consider the following schedule for transactions T_1, T_2 and T_3: what is the correct serialization of the above?



Procedure for determining conflict serializability of a schedule

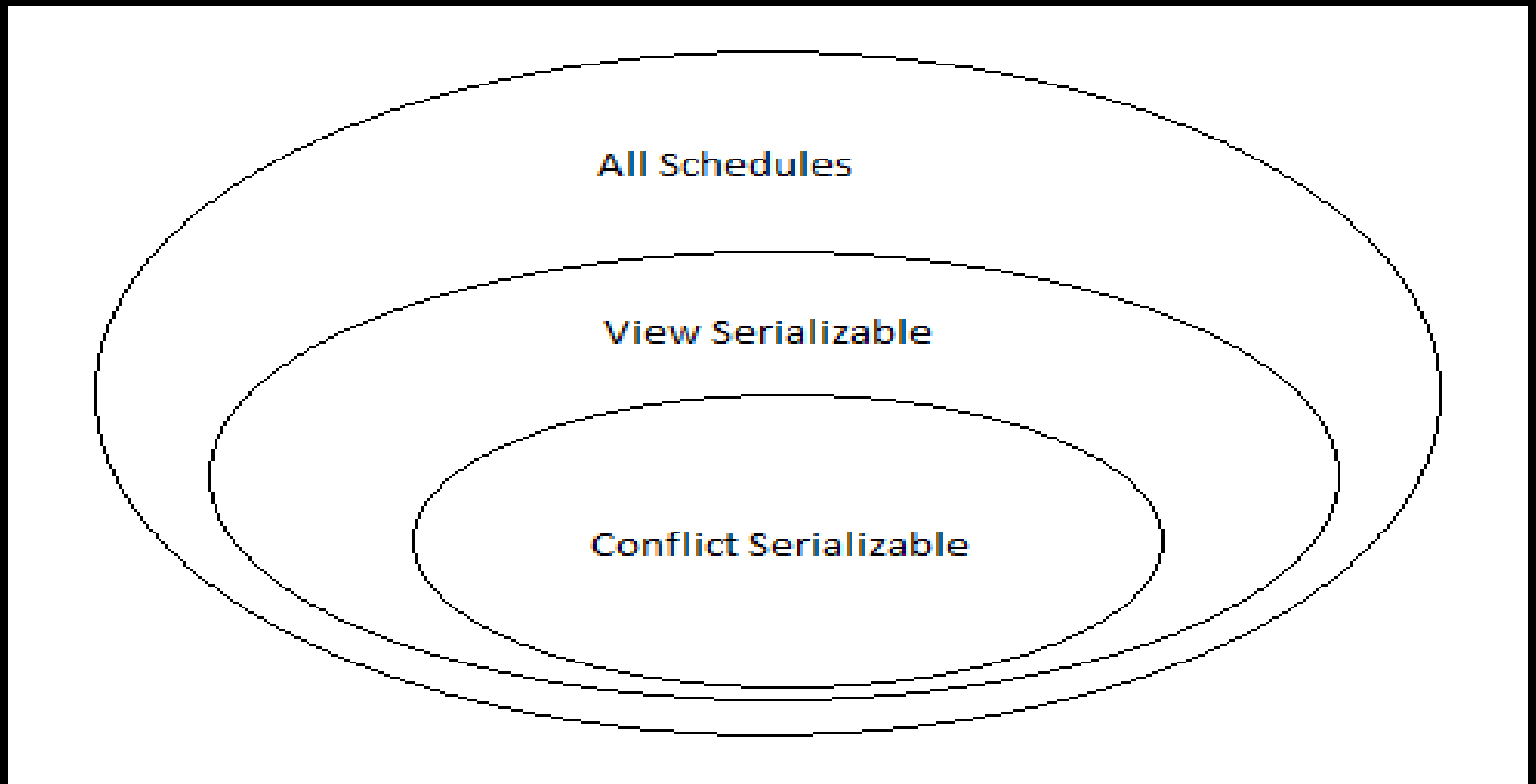
- It can be determined using PRECEDENCE GRAPH method:
- A precedence graph consists of a pair G (V, E)
- V= set of vertices consisting of all the transactions participating in the schedule.
- E= set of edges consists of all edges $T_i \rightarrow T_j$, for which one of the following conditions holds:
  - 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 the precedence graph, then in any serial schedule S' equivalent to S, Ti must appear before T_j.
- **If the precedence graph for S has no cycle, then schedule S is conflict serializable, else it is not.**

VIEW SERIALIZABLE

- If a schedule is not conflict serializable, still it can be consistent, so let us study a weaker form of serializability called View serializability, and even if a schedule is view serializable still it can be consistent.

- If a schedule is conflict serializable then it will also be view serializable, so we must check view serializability only if a schedule is not conflict serializable.



- Two schedules S and S’ are view equivalent, if they satisfy following conditions –
  - For each data item Q, if the transaction T_i reads the initial value of Q in schedule S , then then the transaction T_i must, in schedule S’ ,also read the initial value of Q.
  - If a transaction T_i in schedule S reads any data item Q, which is updated by transaction T_j, then a transaction T_i must in schedule S’ also read data item Q updated by transaction T_j in schedule S’.
  - For each data item Q, the transaction (if any) that performs the final write(Q) operation in schedule S, then the same transaction must also perform final write(Q) in schedule S’.

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

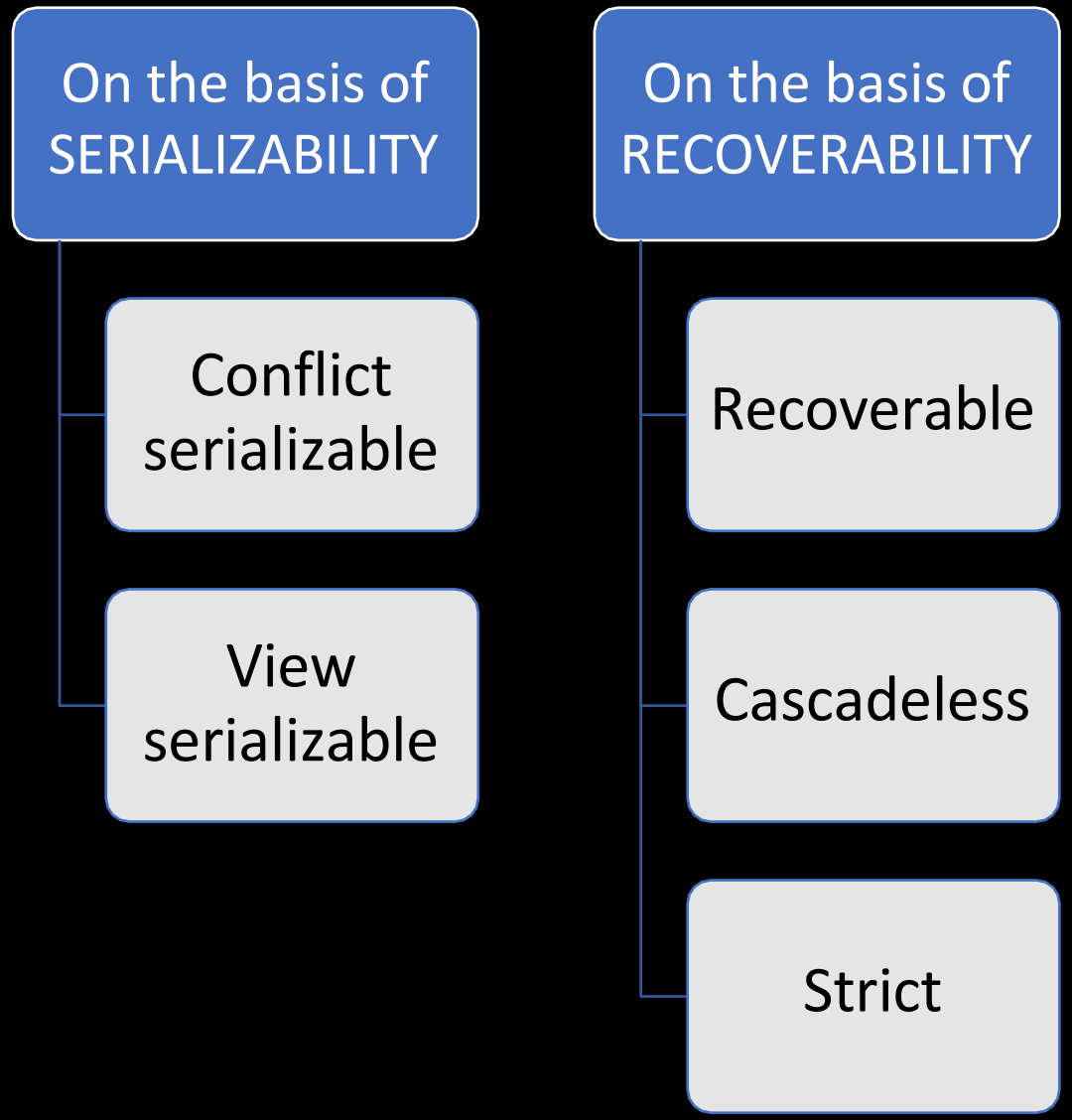
T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

View Serializable

A schedule S is view serializable, if it is view equivalent to a serial schedule.

Schedule A		
T3	T4	T6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

Serial schedule <T3,T4,T6>		
T3	T4	T6
read(Q)		
write(Q)		
	write(Q)	
		write(Q)


On the basis of SERIALIZABILITY
Conflict serializable
View serializable
On the basis of RECOVERABILITY
Recoverable
Cascadeless
Strict

NON- RECOVERABLE Vs RECOVERABLE SCHEDULE

- A schedule in which for each pair of transaction T_i and T_j, such that if T_j reads a data item previously written by T_i, then the commit or abort operation of T_i appears before T_j. Such a schedule is called Non- Recoverable schedule.
- A schedule in which for each pair of transaction T_i and T_j, such that if T_j reads a data item previously written by T_i, then the commit or abort of T_i must appear before T_j. Such a schedule is called Recoverable schedule.

S	
T_1	T_2
R(X)	
W(X)	
	R(X)
	C
C	

S	
T_1	T_2
R(X)	
W(X)	
	R(X)
C	
	C

CASCADING ROLLBACK Vs CASCADELESS SCHEDULE

- It is a phenomenon, in which even if the schedule is recoverable, the a single transaction failure leads to a series of transaction rollbacks, is called cascading rollback. Cascading rollback is undesirable, since it leads to undoing of a significant amount of work. Uncommitted reads are not allowed in cascade less schedule.

- A schedule in which for each pair of transactions T_i and T_j, such that if T_j reads a data item previously written by T_i then the commit or abort of T_i must appear before read operation of T_j. Such a schedule is called cascade less schedule.

S		
T_1	T_2	T_3
R(X)		
W(X)		
	R(X)	
	W(X)	
		R(X)
C		
	C	
		C

S		
T_1	T_2	T_3
R(X)		
W(X)		
C		
	R(X)	
	W(X)	
	C	
		R(X)
		C

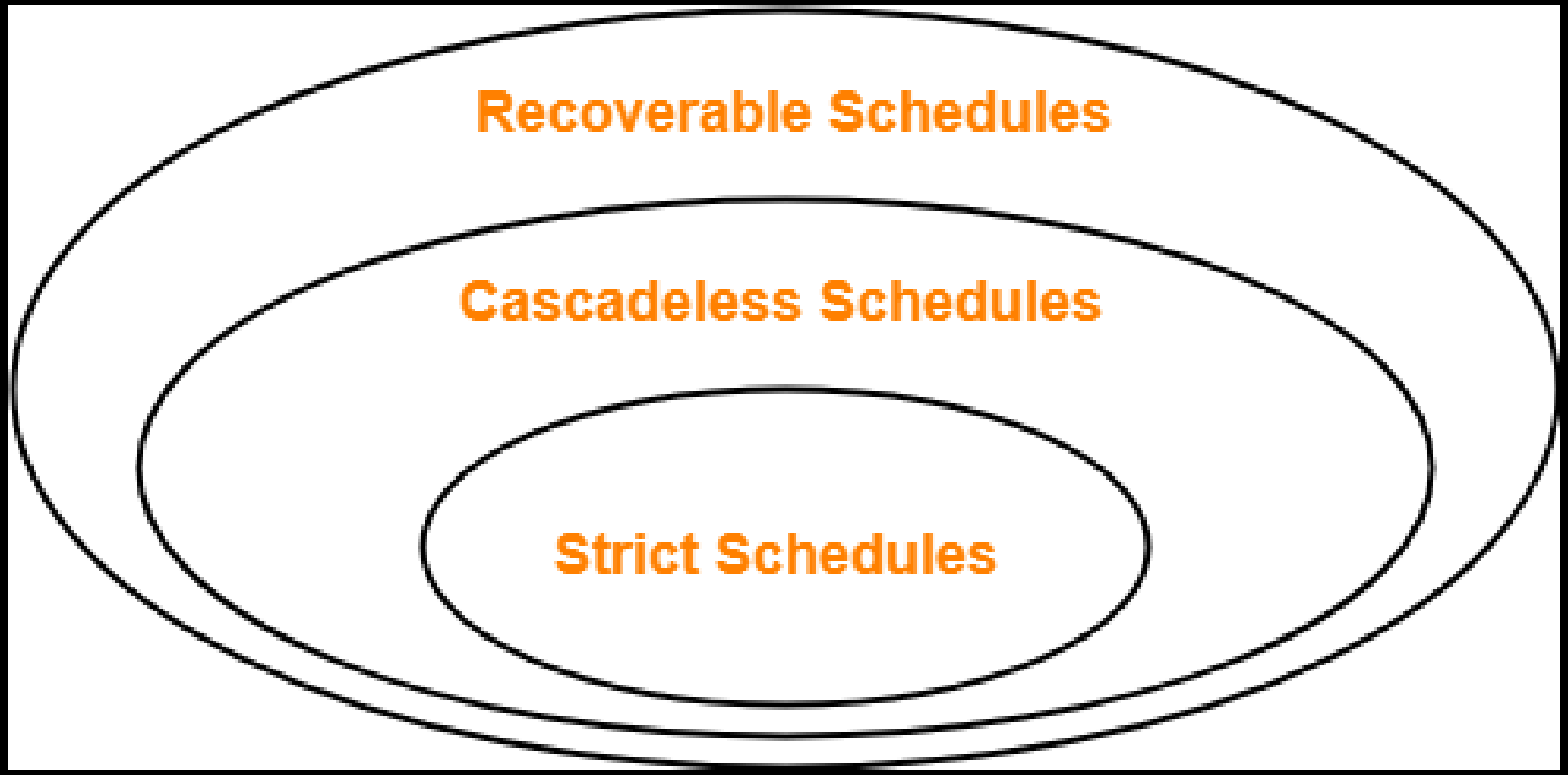
Strict Schedule

- A schedule in which for each pair of transactions T_i and T_j, such that if T_j reads a data item previously written by T_i then the commit or abort of T_i must appear before read and write operation of T_j.

S_1	
T_1	T_2
R(a)	
W(a)	
	W(a)
C	
	R(a)
	C

S_2	
T_1	T_2
R(a)	
W(a)	
C	
	W(a)
	R(a)
	C

S_3	
T_1	T_2
R(a)	
	R(b)
W(a)	
	W(b)
C	
	R(a)
	C


Recoverable Schedules
Cascadeless Schedules
Strict Schedules

Log based Recovery

- The system log, or transaction log, records all the changes or activities happening in the database, ensuring that transactions are durable and can be recovered in the event of a failure.

- Different types of log records represent various stages or actions in a transaction.
  - <T_i start>: This log entry indicates that the transaction T_i has started its execution.
  - < T_i, X_j, V_1, V_2>: This log entry documents a write operation. It states that transaction T_i has changed the value of data item X_j from V_1 to V_2.
  - < T_i commit>: This log entry marks the successful completion of transaction T_i.
  - < T_i abort>: This log entry denotes that the transaction T_i has been aborted, either due to an error or a rollback operation.

- Before any write operation modifies the database, a log record of that operation needs to be created. This is to ensure that in case of a failure, the system can restore the database to a consistent state using the log records.

Aspect	Deferred Database Modification	Immediate Database Modification
Write Operation Timing	Only at the commit point.	As soon as changes occur.
I/O Operations	Reduced, as changes are batched and written at once during the commit, saving on intermediate I/O operations.	Increased, as each change triggers immediate write operations, leading to more frequent I/O operations.
Recovery Complexity	Simpler, as uncommitted changes are not reflected in the database, making rollback easier in case of failures.	More complex, as it may require undoing changes from uncommitted transactions that have been written to the database.

Shadow Paging Recovery Technique

In the shadow paging recovery technique, the database maintains two page tables during a transaction: the current page table (reflecting the state before the transaction began) and the shadow page table (which tracks changes made during the transaction). Here's how it operates:

- **Initialization**: When a transaction begins, the database creates a shadow copy of the page table. The database pages themselves are not duplicated; only the page table entries are duplicated.

- **Modifications**: As the transaction progresses, any changes are reflected in the current page table, while the shadow page table retains the original state. If a page is modified, a new copy of the page is created, and the current page table is updated to point to this new version, thereby ensuring that the shadow page table still points to the original unmodified page.

- **Commit**: Upon transaction commit, the shadow page table is discarded, and the current page table becomes the new committed state of the database. The database atomically switches to using the current page table, ensuring that changes are installed all at once.

- **Recovery**: In case of a system failure before the transaction commits, the database can easily recover by discarding the current page table and reverting back to the shadow page table, thereby restoring the database to its state before the transaction began.

Data fragmentation

- Data fragmentation in the context of a Database Management System (DBMS) refers to the process of breaking down a database into smaller, manageable pieces, and distributing these pieces across a network of computers. This is a significant component in distributed database systems.
  - **Horizontal Fragmentation**: In DBMS, horizontal fragmentation divides a table into sections based on rows. Each fragment contains a subset of rows, usually segmented based on certain conditions or attributes. This can enhance performance by facilitating parallel processing and quicker data retrieval, especially useful in distributed database systems.
  - **Vertical Fragmentation**: This fragmentation type divides a database table by columns, with different sets of columns stored in separate fragments. This approach is employed when different users require access to varied data attributes, promoting efficient data handling and minimizing data transfer times, which is advantageous in scenarios where queries only necessitate a subset of attributes.
  - **Hybrid Fragmentation**: Hybrid or mixed fragmentation in DBMS combines both horizontal and vertical strategies, optimizing data storage and retrieval for complex access patterns. This method can potentially offer performance improvements by utilizing the merits of both horizontal and vertical fragmentation, and is typically implemented in databases with complex, multifaceted data access requirements.

Distributed database

- A distributed database is a database in which storage devices are not all attached to a common processor. It may be stored in multiple computers, located in the same physical location, or dispersed over a network of interconnected computers.

- **Data Replication**
  - ***Advantages:*** Increased Availability, Improved Performance, Enhanced Reliability
  - ***Disadvantages:*** Storage Costs, Maintenance Complexity, Write Complexity.

- **Data Fragmentation**
  - ***Advantages:*** Efficient Data Access, Distributed Processing, Localized Management.
  - ***Disadvantages:*** Complexity, Dependency on Network, Reconstruction Issues.

CONCURRENCY CONTROL

Here we will study those protocol which guarantee to generate schedule which always satisfy desirable properties like conflict serializability. Along with we desire the following properties from schedule generating protocols

- Concurrency should be as high as possible, as this is our ultimate goal because of which we are making all the effort.
- The time taken by a transaction should also be less.
- Easy to understand and implement.

- **Time stamping based method: -** Where before entering the system, a specific order is decided among the transaction, so in case of a clash we can decide which one to allow and which to stop.

- **Lock based method: -** where we ask a transaction to first lock a data item before using it. So that no different transaction can use a data at the same time, removing any possibility of conflict.
  - 2 phase locking
    1. Basic 2pl
    2. Conservative 2pl
    3. Rigorous 2pl
    4. Strict 2pl
  - Graph based protocol

- **Validation based protocol –** Majority of transactions are read only transactions, the rate of conflicts among the transaction may be low, thus many of transaction, if executed without the supervision of a concurrency control scheme, would nerveless leave the system in a consistent state.

TIME STAMP ORDERING PROTOCOL

- Basic idea of time stamping is to decide the order between the transaction before they enter in the system using a stamp (time stamp), in case of any conflict during the execution order can be decided using the time stamp.

- Let’s understand how this protocol works, here we have two idea of timestamping, one for the transaction, and other for the data item.

- Time stamp for transaction,
  - With each transaction t_i, in the system, we associate a unique fixed timestamp, denoted by $TS(t_i)$.
  - This timestamp is assigned by database system to a transaction at time transaction enters into the system.
  - If a transaction has been assigned a timestamp $TS(t_i)$ and a new transaction t_j , enters into the system with a timestamp $TS(t_j)$, then always $TS(t_i) < TS(t_j)$.

- Two things are to be noted

1. First time stamp of a transaction remain fixed throughout the execution

2. Second it is unique means no two transaction can have the same timestamp.

- Time stamp with data item, in order to assure such scheme, the protocol maintains for each data item Q two timestamp values:

    1. **W-timestamp(Q)** is the largest time-stamp of any transaction that executed write(Q) successfully.

    2. **R-timestamp(Q)** is the largest time-stamp of any transaction that executed read(Q) successfully.

- These timestamps are updated whenever a new read(Q) or write(Q) instruction is executed.

- Suppose a transaction Ti request a ***read(Q)***

1. If **TS(Ti) < W-timestamp(Q),** then Ti needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and Ti is rolled back.

2. If **TS(Ti)≥ W-timestamp(Q),** then the read operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).

- Suppose that transaction Ti issues ***write(Q).***
  1. If **TS(Ti) < R-timestamp(Q),** then the value of Q that Ti is producing was needed previously, and the system assumed that that value would never be produced. Hence, the write operation is rejected, and Ti is rolled back.

  2. If **TS(Ti) < W-timestamp(Q),** then Ti is attempting to write an obsolete value of Q. Hence, this write operation is rejected, and Ti is rolled back.

  3. If **TS(Ti) ≥ R-timestamp(Q),** then the write operation is executed, and W-timestamp(Q) is set to max(W-timestamp(Q), TS(Ti)).

  4. If **TS(Ti) ≥ W-timestamp(Q),** then the write operation is executed, and W-timestamp(Q) is set to max(W-timestamp(Q), TS(Ti)).

	Conflict Serializability	View Serializability	Recoverability	Cascadelessness	Deadlock Freedom
Time Stamp Ordering	YES	YES	NO	NO	YES
Thomas Write Rule					
Basic 2PL					
Conservative 2PL					
Rigorous 2PL					
Strict 2PL					

THOMAS WRITE RULE

- Thomas write is an improvement in time stamping protocol, which makes some modification and may generate those protocols that are even view serializable, because it allows greater potential concurrency.

- It is a Modified version of the timestamp-ordering protocol in which Blind write operations may be ignored under certain circumstances.

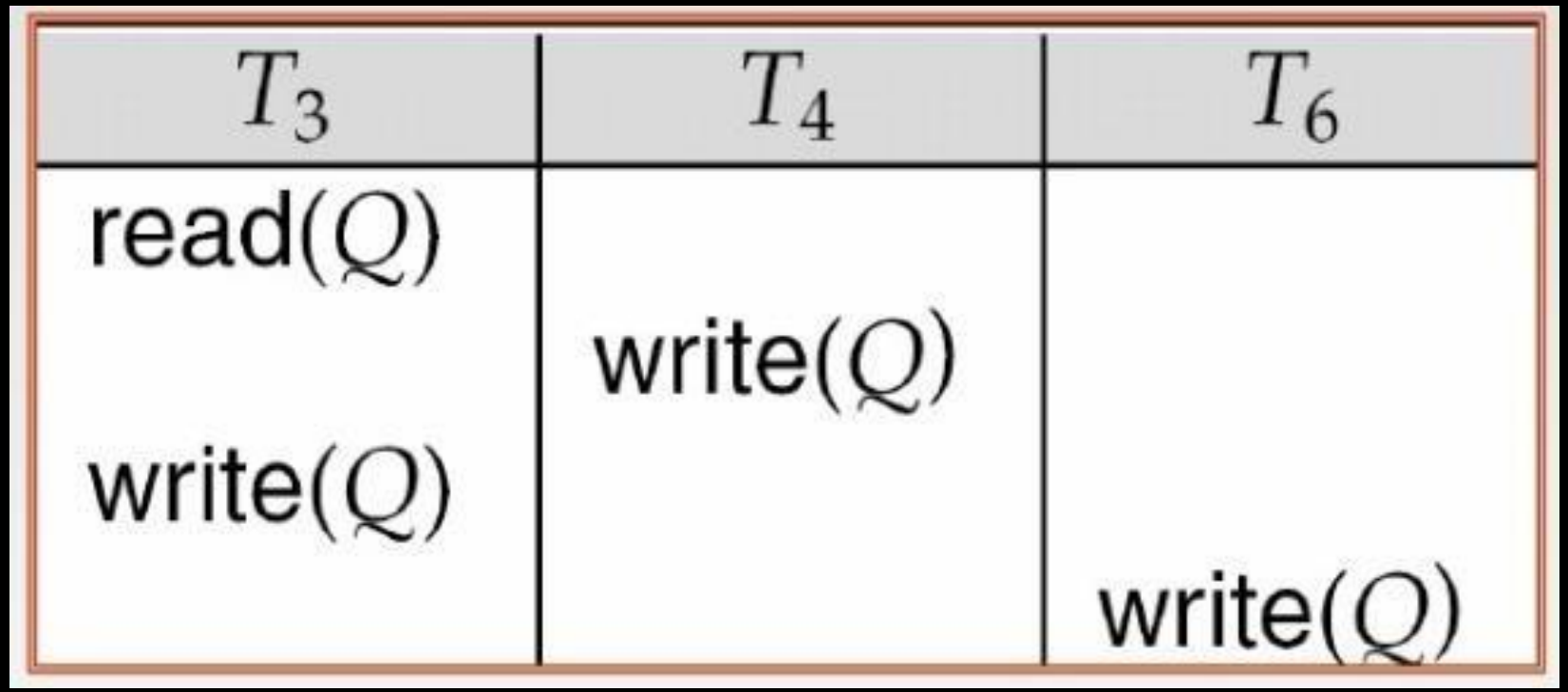
- The protocol rules for read operations remain unchanged. while for write operation, there is slightly change in Thomas write rule than timestamp ordering protocol.

When T_i attempts to write data item Q,

- **if TS(T_i) < W-timestamp(Q),** then T_i is attempting to write an obsolete value of {Q}. Rather than rolling back T_i as the timestamp ordering protocol would have done, this {write} operation can be ignored.

- This modification is valid as the any transaction with **TS(T_i) < W-timestamp(Q),** the value written by this transaction will never be read by any other transaction performing Read(Q) ignoring such obsolete write operation is considerable.

- Thomas' Write Rule allows greater potential concurrency. Allows some view-serializable schedules that are not conflict serializable.



	Conflict Serializability	View Serializability	Recoverability	Cascadelessness	Deadlock Freedom
Time Stamp Ordering	YES	YES	NO	NO	YES
Thomas Write Rule	NO	YES	NO	NO	YES
Basic 2PL					
Conservative 2PL					
Rigorous 2PL					
Strict 2PL					

Lock Based Protocols

- To ensure isolation is to require that data items be accessed in a mutually exclusive manner i.e. while one transaction is accessing a data item, no other transaction can modify that data item. Locking is the most fundamental approach to ensure this.

- Lock based protocols ensure this requirement. Idea is first obtain a lock on the desired data item then if lock is granted then perform the operation and then unlock it.

- In general, we support two modes of lock because, to provide better concurrency.

- ***Shared mode***
  - If transaction T_i has obtained a shared-mode lock (denoted by S) on any data item Q, then T_i can read, but cannot write Q, any other transaction can also acquire a shared mode lock on the same data item(this is the reason we called this shared mode).

- **Exclusive mode**
  - If transaction T_i has obtained an exclusive-mode lock (denoted by X) on any data item Q, then T_i can both read and write Q, any other transaction cannot acquire either a shared or exclusive mode lock on the same data item. (this is the reason we called this exclusive mode)

Lock –Compatibility Matrix

- Conclusion shared is compatible only with shared while exclusive is not compatible either with shared or exclusive.
- To access a data item, transaction T_i must first lock that item, if the data item is already locked by another transaction in an incompatible mode, or some other transaction is already waiting in non-compatible mode, then concurrency control manager will not grant the lock until all incompatible locks held by other transactions have been released. The lock is then granted.

		Current State of lock of data items		
		Exclusive	Shared	Unlocked
Requested Lock	Exclusive	N	N	Y
	Shared	N	Y	Y
	Unlock	Y	Y	-

- Lock based protocol *do not ensure serializability* as granting and releasing of lock do not follow any order and any transaction any time may go for lock and unlock. Here in the example below we can see, that even this transaction in using locking but neither it is conflict serializable nor independent from deadlock.

T_1	T_2
LOCK-X(A)	
READ(A)	
WRITE(A)	
UNLOCK(A)	
	LOCK-S(B)
	READ(B)
	UNLOCK(B)
LOCK-X(B)	
READ(B)	
WRITE(B)	
UNLOCK(B)	
	LOCK-S(A)
	READ(A)
	UNLOCK(A)

- If we do not use locking, or if we unlock data items too soon after reading or writing them, we may get inconsistent states, as there exists a possibility of dirty read. On the other hand, if we do not unlock a data item before requesting a lock on another data item, concurrency will be poor.

- We shall require that each transaction in the system follow a set of rules, called a **locking protocol**, indicating when a transaction may lock and unlock each of the data items for e.g. 2pl or graph based locking.

- Locking protocols restrict the number of possible schedules.

Two phase locking protocol(2PL)

- The protocol ensures that each transaction issue lock and unlock requests in two phases, note that each transaction will be 2 phased not schedule.

- **Growing phase**- A transaction may obtain locks, but not release any locks.

- **Shrinking phase**- A transaction may release locks, but may not obtain any new locks.

- Initially a transaction is in growing phase and acquires lock as needed and in between can perform operation reach to lock point and once a transaction releases a lock, it can issue no more lock requests i.e. it enters the shrinking phase.

T_1	T_2
LOCK-X(A)	
READ(A)	
WRITE(A)	
	LOCK-S(B)
	READ(B)
LOCK-X(B)	
READ(B)	
WRITE(B)	
	LOCK-S(A)
	READ(A)
	UNLOCK(B)
UNLOCK(A)	
UNLOCK(B)	
	UNLOCK(A)

	Conflict Serializability	View Serializability	Recoverability	Cascadelessness	Deadlock Freedom
Time Stamp Ordering	YES	YES	NO	NO	YES
Thomas Write Rule	NO	YES	NO	NO	YES
Basic 2PL	YES	YES	NO	NO	NO
Conservative 2PL					
Rigorous 2PL					
Strict 2PL					

Properties

- 2PL ensures conflict serializability, and the ordering of transaction over lock points is itself a serializability order of a schedule in 2PL.
- If a schedule is allowed in 2PL protocol then definitely it is always conflict serializable. But it is not necessary that if a schedule is conflict serializable then it will be generated by 2pl. Equivalent serial schedule is based on the order of lock points.
- View serializability is also guaranteed.
- Does not ensure freedom from deadlock
- May cause non-recoverability.
- Cascading rollback may occur.

Conservative 2PL

- The idea is there is no growing phase transaction start directly from lock point, i.e. transaction must first acquire all the required locks then only it can start execution. If all the locks are not available then transaction must release the acquired locks and must wait.
  - Shrinking phase will work as usual, and transaction can unlock any data item anytime.
  - we must have a knowledge in future to understand what is data required so that we can use it

	Conflict Serializability	View Serializability	Recoverability	Cascadelessness	Deadlock Freedom
Time Stamp Ordering	YES	YES	NO	NO	YES
Thomas Write Rule	NO	YES	NO	NO	YES
Basic 2PL	YES	YES	NO	NO	NO
Conservative 2PL	YES	YES	NO	NO	YES
Rigorous 2PL					
Strict 2PL					

RIGOROUS 2PL

- Requires that all locks be held until the transaction commits.
- This protocol requires that locking be two phase and also all the locks taken be held by transaction until that transaction commit.
- Hence there is no shrinking phase in the system.

	Conflict Serializability	View Serializability	Recoverability	Cascadelessness	Deadlock Freedom
Time Stamp Ordering	YES	YES	NO	NO	YES
Thomas Write Rule	NO	YES	NO	NO	YES
Basic 2PL	YES	YES	NO	NO	NO
Conservative 2PL	YES	YES	NO	NO	YES
Rigorous 2PL	YES	YES	YES	YES	NO
Strict 2PL					

STRICT 2PL

- that all exclusive-mode locks taken by a transaction be held until that transaction commits. This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits, preventing any other transaction from reading the data.

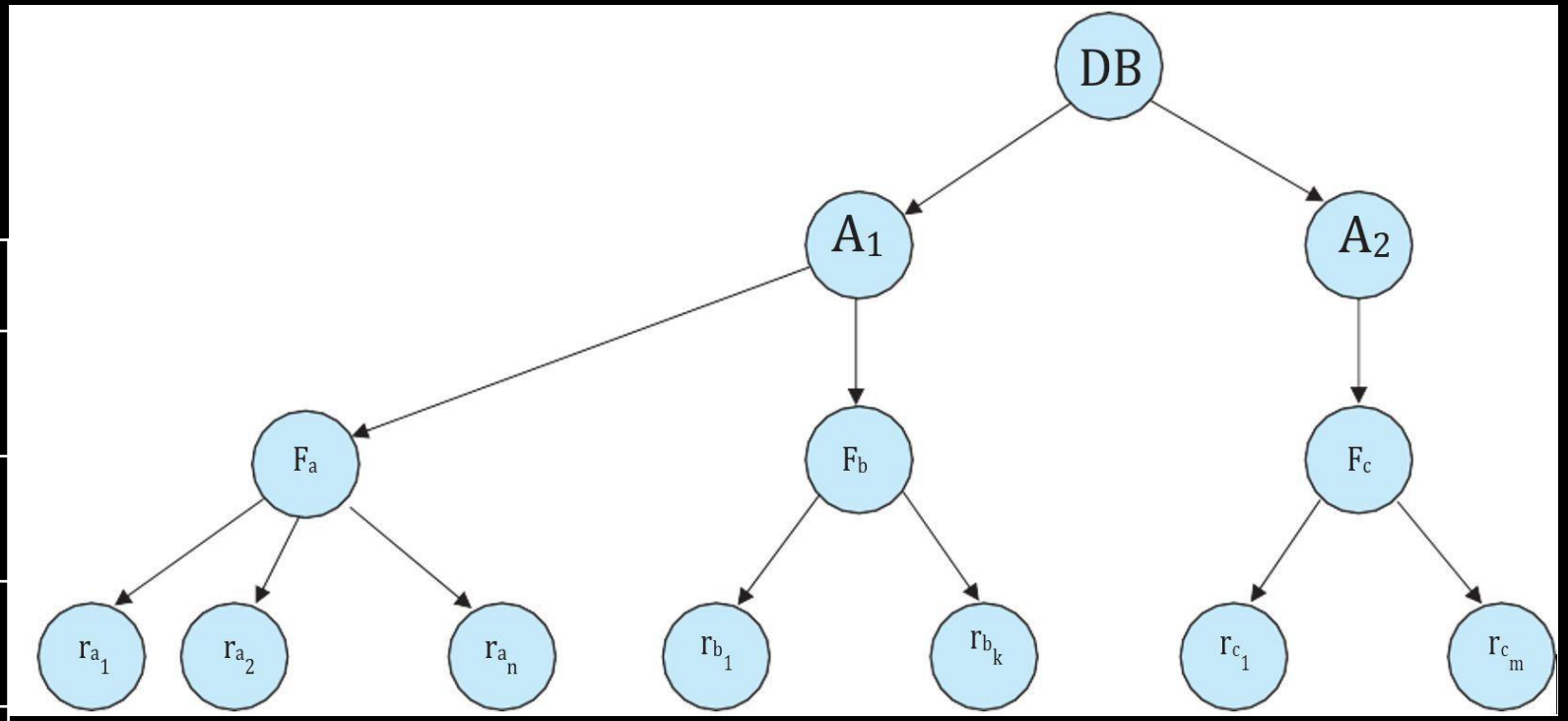
- This protocol requires that locking be two phase and also that exclusive –mode locks taken by transaction be held until that transaction commits.

- So it is simplified form of rigorous 2pl

	Conflict Serializability	View Serializability	Recoverability	Cascadelessness	Deadlock Freedom
Time Stamp Ordering	YES	YES	NO	NO	YES
Thomas Write Rule	NO	YES	NO	NO	YES
Basic 2PL	YES	YES	NO	NO	NO
Conservative 2PL	YES	YES	NO	NO	YES
Rigorous 2PL	YES	YES	YES	YES	NO
Strict 2PL	YES	YES	YES	YES	NO

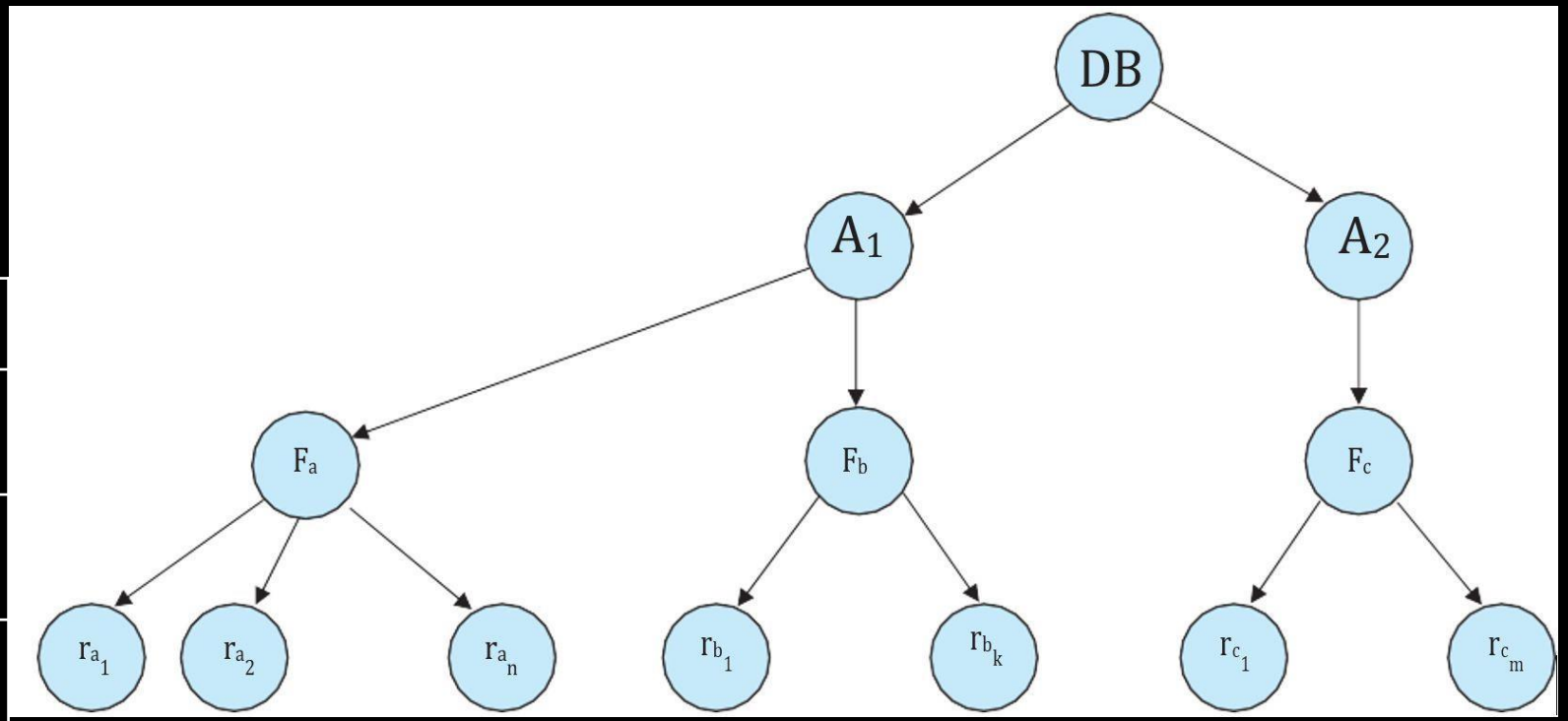
Multiple Granularity

	IS	IX	S	X
IS				
IX				
S				
X				



Multiple Granularity

	IS	IX	S	X
IS	true	true	true	false
IX	true	true	false	false
S	true	false	true	false
X	false	false	false	false



Knowledge Gate Website

Validation-Based Protocols

- In cases where a majority of transactions are read-only transactions, the rate of conflicts among transactions may be low.

- Thus, many of these transactions, if executed without the supervision of a concurrency-control scheme, would nevertheless leave the system in a consistent state.

- A concurrency-control scheme imposes overhead of code execution and possible delay of transactions. It may be better to use an alternative scheme that imposes less overhead.

- A difficulty in reducing the overhead is that we do not know in advance which transactions will be involved in a conflict. To gain that knowledge, we need a scheme for *monitoring* the system.

- The **validation protocol** requires that each transaction T_i executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction. The phases are, in order:

- **Read phase**. During this phase, the system executes transaction T_i. It reads the values of the various data items and stores them in variables local to T_i. It performs all write operations on temporary local variables, without updates of the actual database.

- **Validation phase**. The validation test (described below) is applied to trans- action T_i. This determines whether T_i is allowed to proceed to the write phase without causing a violation of serializability. If a transaction fails the validation test, the system aborts the transaction.

- **Write phase**. If the validation test succeeds for transaction T_i, the temporary local variables that hold the results of any write operations performed by T_i are copied to the database. Read-only transactions omit this phase.