UNIT – V

1) Lock – based Protocol
2) Timestamp Based Protocols
3) Validation based Protocols
4) Deadlock Handling
5) Failure Classification
6) Storage Structure
7) Recovery and Atomicity
8) Log-Based Recovery
9) Recovery with Concurrent Transactions

Concurrency Control

- Concurrency Control ensures that Database transactions are performed concurrently and accurately to produce correct results.
- Concurrent access is quite easy if all users are just reading data. There is no way they can interfere with one another. Though for any practical Database, it would have a mix of READ and WRITE operations and hence the concurrency is a challenge.
- DBMS Concurrency Control is used to address such conflicts, which mostly occur with a multi-user system.
- Therefore, Concurrency Control is the most important element for proper functioning of a Database Management System where two or more database transactions are executed simultaneously, which require access to the same data.

Concurrency Control

- Different concurrency control protocols offer different benefits between the amount of concurrency they allow.
- Following are the Concurrency Control techniques in DBMS:
  1) Lock-Based Protocols
  2) Two Phase Locking Protocol
  3) Timestamp-Based Protocols
  4) Validation-Based Protocols

1. Lock Based Protocols

- **Lock Based Protocols** in DBMS is a mechanism in which a transaction cannot Read or Write the data until it acquires an appropriate lock.
- Lock based protocols help to eliminate the concurrency problem in DBMS for simultaneous transactions by locking a particular transaction to a single user.
- Locks in DBMS help synchronize to access the database items by concurrent transactions.
- **Locking** is one of the most widely used mechanisms to ensure **Serializabilty.**
- A Transaction must be obtained **a Read or write Lock** on a data item before it can perform **Read or write Operation**.

1.Lock based protocol

The basic rules for Locking are given below

Read Lock (or) Shared Lock(S)

- If a Transaction has a Read lock on a data item, it can read the item but not update it
- If a transaction has a Read lock on the data item, other transaction can obtain Read Lock on the data item but no Write Locks.
- So, the Read Lock is also called a **shared lock**

Write Lock (or) Exclusive Lock (X)

- If a transaction has a write Lock on a data item, it can both read and update the data item.
- If a transaction has a write Lock on the data item, then other transactions cannot obtain either a Read lock or write lock on the data item.
- So, the Write Lock is also known as **Exclusive Lock**

1.Lock based protocol

Lock Compatibility Table

	S	X
S	✔	✗
X	✗	✗

- S in a Row specifies that a transaction locks a data item in shared mode. So, other transactions can lock the data item in shared mode but not in exclusive mode.
- X in a Row specifies that a transaction locks a data item in Exclusive mode. So, other transactions cannot lock the data item in both shared or exclusive mode.

1.Lock based protocol

- **Using the above specified Locking Mechanisms alone does not guarantee Serializabilty of Schedules.**
- **For Example** : if we take 2 transactions T1 and T2

T1
S(Y)
R(Y)
Unlock (Y)
X(X)
R(X)
X := X+ Y
W(X)
Unlock (x)

T2
S(X)
R(X)
Unlock (X)
X(Y)
R(Y)
Y := X+ Y
W(Y)
Unlock(y)

- This is an Example of using the above specified Locking Technique .
- **Scheduling these transactions in an interleaving Fashion will not ensure Serializabilty .**

Two Phase Locking Protocol

- One way of Guaranteeing Serializabilty is to use an additional protocol Known as **two Phase Locking or 2PL.**
- ***A transaction is said to follow 2PL protocol if all Locking operations precede the first Unlock operation in the transaction.***
- In 2PL, Every Transaction can be divided into two Phases

  Growing phase and

  Shirking Phase
- In the **Growing Phase** , the transaction acquires all locks needed but cannot release any Locks.
- In the **Shrinking Phase**, the Transaction releases all the acquired locks but cannot acquire new Locks.

Two Phase Locking Protocol

- **The above Specified Transactions T1 and T2 are the example transactions that do not follow the 2PL. This is because X(X) operation follows unlock (Y) operation in T1 and similarly, the X(Y) operation follows unlock (X) operation in T2.**

T1
S(Y)
R(Y)
Unlock (Y)
X(X)
R(X)
X := X+ Y
W(X)
Unlock (x)

T2
S(X)
R(X)
Unlock (X)
X(Y)
R(Y)
Y := X+ Y
W(Y)
Unlock(y)

- If Concurrency Control Module enforces 2PL on the above specified Transactions T1 and T2, the transactions can be rewritten as T11 and T21 as shown below

Two Phase Locking Protocol

T11
S(Y)
R(Y)
X(X)
Unlock (Y)
R(X)
X := X+Y
W(X)
Unlock (X)

T21
S(X)
R(X)
X(Y)
Unlock (X)
R(Y)
Y := X+Y
W(Y)
Unlock (Y)

Both transactions followed 2PL, so if these transactions are executed in a interleaving fashion definitely, that particular Non serial Schedule is a Serializable Schedule.

It can be proved that, if Every transaction in a schedule follows a 2PL Protocol, the Schedule is guaranteed to be Serializable.

i.e., 2PL Ensure Conflict Serializabilty. In addition to being Serializable, Schedules should be Cascade less. Cascading Roll back may occur under 2PL.

Cascading roll backs can be avoided by a modification to a 2PL called Strict 2PL protocol

2. Timestamp-Based Protocols

- ***Lock based protocols ensure conflict serializability but it causes 2 problems***
  - **Deadlock and**
  - **Starvation**
- **The alternative approaches to control concurrency are**
  1) **Timestamp-Based Protocols**
  2) **Validation Based Protocols**

3) Timestamp-Based Protocols

- The Timestamp methods for concurrency control does not need any locks and therefore there are no deadlocks
- Locking methods generally prevent conflicts by making transactions to wait.
- Timestamp methods do not make the transactions wait.
- In this method, Transactions involved in a conflict are simply rolled back and restarted.

2. Timestamp-Based Protocols

- The most commonly used concurrency protocol is the timestamp based protocol. This protocol uses either system time or logical counter as a timestamp.
- Lock-based protocols manage the order between the conflicting pairs among transactions at the time of execution, whereas timestamp-based protocols start working as soon as a transaction is created.
- Every transaction has a timestamp associated with it, and the ordering is determined by the age of the transaction. A transaction created at 0002 clock time would be older than all other transactions that come after it. For example, any transaction 'y' entering the system at 0004 is two seconds younger and the priority would be given to the older one.
- In addition, every data item is given the latest read and write-timestamp. This lets the system know when the last ‘read and write’ operation was performed on the data item.

2. Timestamp-Based Protocols

Timestamp Ordering Protocol

- The timestamp-ordering protocol ensures serializability among transactions in their conflicting read and write operations.
- This is the responsibility of the protocol system that the conflicting pair of tasks should be executed according to the timestamp values of the transactions.

1) **Timestamp of a transaction is a unique identifier determined by system clock or logical counter**

   The timestamp of transaction T_i is denoted as $TS(T_i)$.

2) **Read Timestamp of a data item is the timestamp of a transaction where the last read has done**

   Read time-stamp of data-item X is denoted by R-TS(X).

3) **Write Timestamp of a data item is the timestamp of a transaction where the last write has done**

   Write time-stamp of data-item X is denoted by W-TS(X).

2. Timestamp-Based Protocols

Timestamp ordering protocol works as follows –

If a transaction Ti issues a read(X) operation –

If TS(Ti) < W-TS(X)

Operation rejected.

If TS(Ti) >= W-TS(X)

Operation executed.

All data-item timestamps updated.

If a transaction Ti issues a write(X) operation –

If TS(Ti) < R-TS(X)

Operation rejected.

If TS(Ti) < W-TS(X)

Operation rejected and Ti rolled back.

Otherwise, operation executed.

2. Timestamp-Based Protocols

Conflicting Operations

1) (W,R)
2) (R,W)
3) (W,W)

Timestamp Ordering protocol tries to preserve the order of conflicting operations

1) (W,R)

- To Perform Read operation in T2, get TS(T2) and W_TS(A)
- TS(T2) = 4, W_TS(A) = 3.
- Since we want to read updated data, TS(T2) (where we want to read) should be the younger Transaction) should be greater than W_TS(A)
- Since TS(T2) > W_TS(A), we can perform Read Operation, else Abort and restart the Transaction

T1	T2	T3
3	4	5
W(A)		
	R(A)	

T1	T2	T3	T4(restart)
10	11	12	13
		W(A)	
	R(A)		R(A)
	Abort		

2. Timestamp-Based Protocols

2) (R,W)

- To perform Write operation in T2, get TS(T2) and R_TS(A)
- TS(T2) = 4, R_TS(A) = 3.
- Since we want to write a data item A in TS(T2) , TS(T2) should be greater than R_TS(A)
- Since TS(T2) > R_TS(A), we can perform Write Operation, else Abort and restart the Transaction

T1	T2	T3
3	4	5
R(A)		
	W(A)	

T1	T2	T3	T4(restart)
3	4	5	6
		R(A)	
	W(A)		W(A)
	Abort		

2. Timestamp-Based Protocols

2) (W,W)

- To perform Write operation in T2, get TS(T2) and W_TS(A)
- TS(T2) = 3 , W_TS(A) = 2.
- Since we want to write a data item A in TS(T2) , TS(T2) should be greater than W_TS(A)
- Since TS(T2) > W_TS(A), we can perform Write Operation, else Abort and restart the Transaction

T1	T2	T3
2	3	4
W(A)		
	W(A)	

T1	T2	T3	T4(restart)
3	4	5	6
		W(A)	
	W(A)		W(A)
	Abort		

3. Validation Based Protocol

- Validation phase is also known as optimistic concurrency control technique.
- In the validation based protocol, the transaction is executed in the following three phases:

  Read phase: In this phase, the transaction T is read and executed. It is used to read the value of various data items and stores them in temporary local variables. It can perform all the write operations on temporary variables without an update to the actual database.

  Validation phase: In this phase, the temporary variable value will be validated against the actual data to see if it violates the serializability.

  Write phase: If the validation of the transaction is validated, then the temporary results are written to the database or system otherwise the transaction is rolled back.

3. Validation Based Protocol

- Here each phase has the following different timestamps:
  1) **Start(Ti):** It contains the time when Ti started its execution.
  2) **Validation (T_i):** It contains the time when Ti finishes its read phase and starts its validation phase.
  3) **Finish(Ti):** It contains the time when Ti finishes its write phase.

3. Validation Based Protocol

- To manage the concurrency between transactions T1 and T2, the validation test process for T1 should validate all the T1 operations should follow TS(T1) < TS(T2) where TS is the timestamp and one of the following condition should be satisfying

1) Finish T1 < Start T2

In this condition, T1 completes all the execution processes before the T2 starts the operations. It regulates maintaining the serializability.

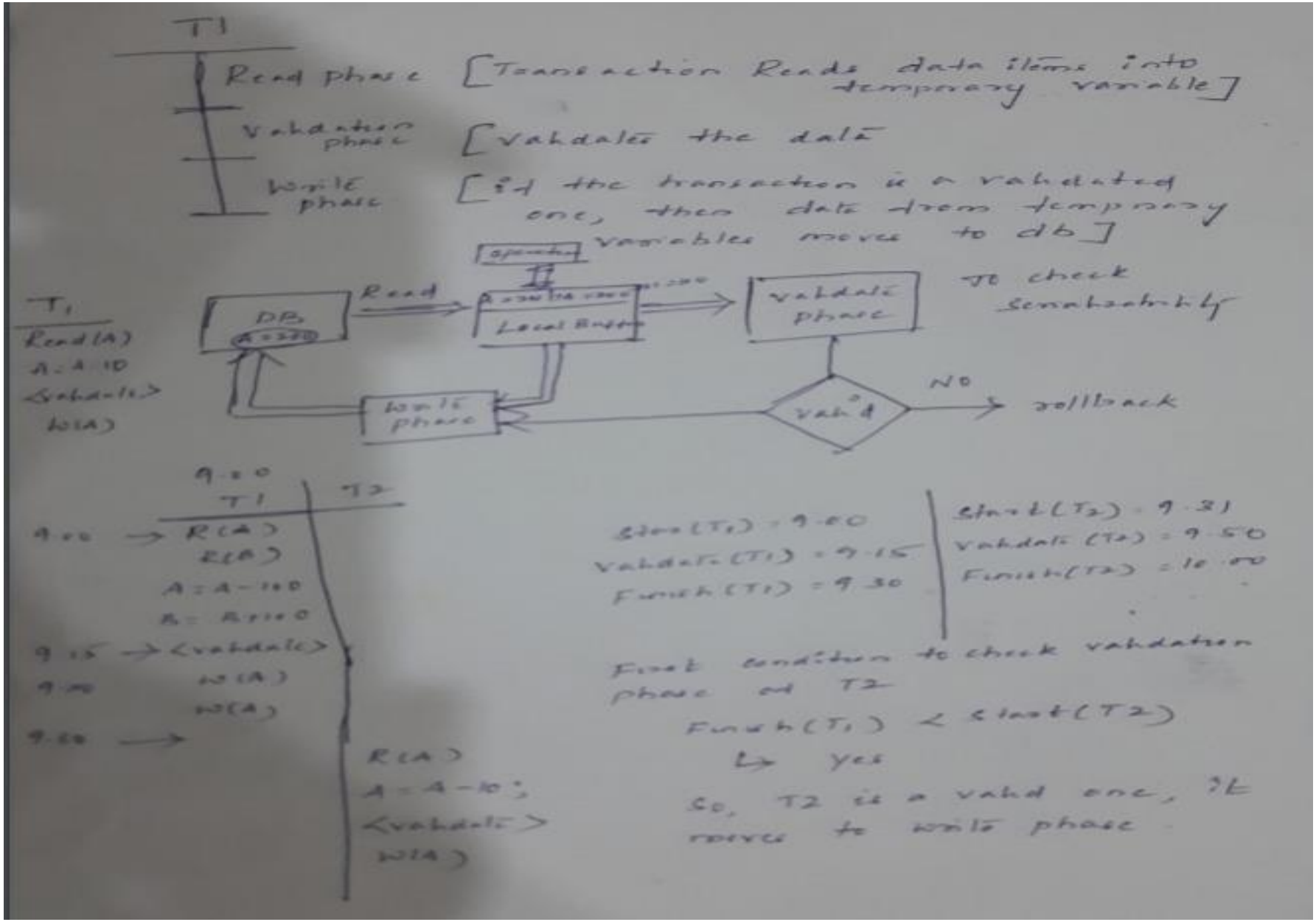
2) Start(T2) <Finish(T1) <Validate(T2)

The validation phase of T2 should occur after the finish phase of T1. This scenario is useful for concurrent transaction serializability.

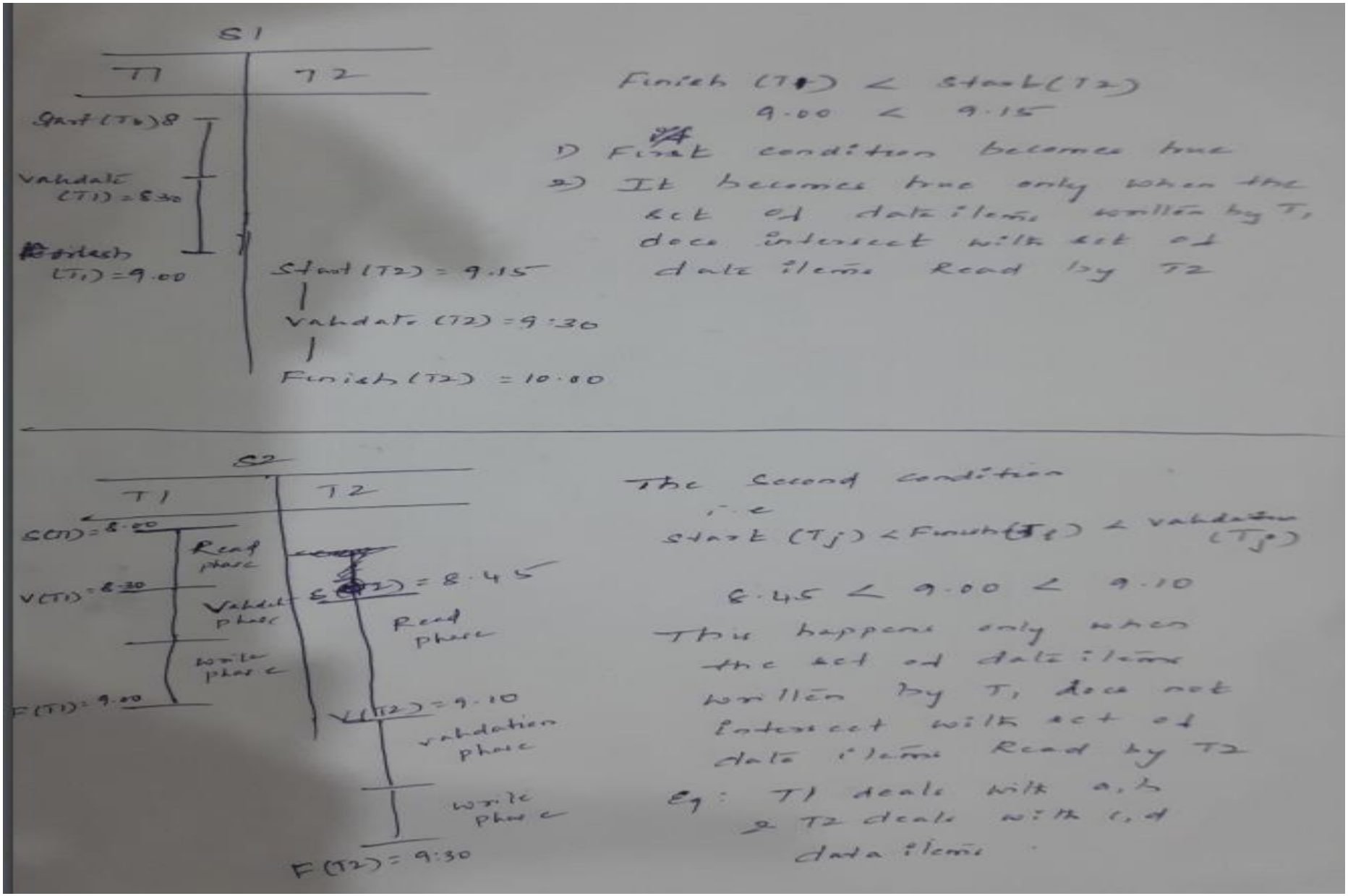
The Transactions are able to access the mutually exclusive database resource at a particular timestamp while validating the protocol conditions.

.

3. Validation Based Protocol



3. Validation Based Protocol



4. Deadlock Handling

Dead Lock

- ***A System is in a Dead lock state if there exists a set of Transactions such that every transaction in the set is waiting for another transaction in the set.***

  For Example, there exists a set of waiting transactions {T0, T1 ,… , Tn} such that,

  T0 is waiting for a data item that T1 holds

  And T1 is waiting for a data item that T2 Holds

  And -----

  And Tn-1 is waiting for a data item that Tn holds

  And Tn is waiting for a data item that T0 holds.

- None of the transactions can make progress such a situation.
- The only remedy to this undesirable situation is to Roll back some of the transactions involved in the Dead lock.
- **There are two principle methods for dealing with the Dead Lock problem**
  - **a) Dead Lock detection and Recovery**
  - **b) Dead Lock prevention**

4. Deadlock Handling

a) Dead Lock Detection and Recovery :

- A Deadlock exists in the system if and only if the ***wait-for-graph*** contains a ***Cycle.***
- The Lock manager maintains a structure called a wait-for-graph to detect deadlock cycles.

Procedure to draw wait-for-graph

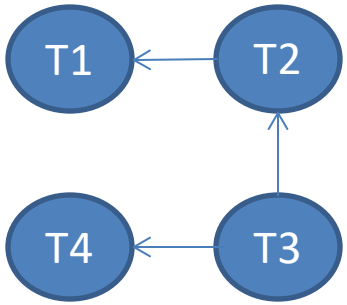
- The **Nodes** correspond to active transactions and
- An **Arc** from Ti to Tj if Ti is waiting for Tj to release a lock
- **The Lock Manager adds Edges to the Graph when it queues lock requests and removes edges when it grants lock requests.**
- Consider the Schedule S, the last step, shown below [X(B) and X(A)], creates a cycle in the wait-for-graph.

4. Deadlock Handling

T1	T2	T3	T4
S(A)			
R(A)			
	X(B)		
	W(B)		
S(B)			
	X(C)		
		S(C)	
		R(C)	
		X(D)	S(D)

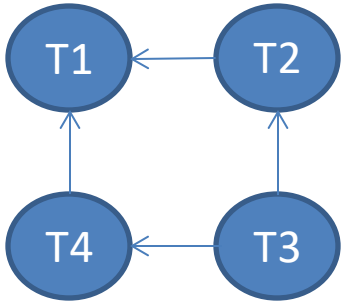
T1	T2	T3	T4
S(A)			
R(A)			
	X(B)		
	W(B)		
S(B)			
	X(C)		
		S(C)	
X(D)		R(C)	
		X(D)	S(D)

Wait-for-graph for Schedule A



Schedule A has No cycle, so it has no Deadlock

Wait-for-graph for Schedule B



Schedule B has a cycle, so it has a Deadlock

4. Deadlock Handling

b) Dead Lock Prevention

- Prevention is commonly used if the probability that the system would enter a deadlock state is relatively high; otherwise, Detection and Recovery are more efficient.
- Two different Deadlock Prevention schemes using Time stamps have been proposed
  - **1) Wait-die Scheme**
  - **2) Wound-Wait scheme.**
- Transaction Timestamp is a unique identifier assigned to each transaction. The Timestamps are typically based on the order in which transactions are started**.**
- The Lower the Timestamp, the higher is the transaction's priority ; i.e.. the oldest transaction has the highest priority.
- If a transaction Ti requests a Lock and transaction Tj holds a Conflicting lock, the lock manager can use one of the following two policies.
  - **Wait –Die Scheme :** If Ti has higher priority, it is allowed to wait; otherwise , it is aborted.
  - **Wound-Wait scheme :** If Ti has Higher Priority, abort Tj; otherwise, Ti Waits.

2) Starvation

- Consider the following situation. Suppose a transaction T2 has a shared lock on a data item and another transaction T1 requests an exclusive lock on the same data item.
- As we have seen, T1 will have to wait until T2 releases the lock. Meantime, assume that another transaction T3 requests a shared lock on the data item.
- Since the lock request T3 is compatible to the lock granted to T2, T3 will be granted the shared lock on the data item.
- At this point even if T2 release the lock, T1 will have to wait until T3 also releases the lock. The transaction T1 can wait for an exclusive lock endlessly if other transactions continue to request and acquire shared locks on the data item.
- We say that the transaction T1 is starved, as it is not making any progress.

5. Failure Classification

Crash Recovery

- DBMS is a highly complex system with hundreds of transactions being executed every second.
- The durability and robustness of a DBMS depends on its complex architecture and its underlying hardware and system software.
- If it fails or crashes amid transactions, it is expected that the system would follow some sort of algorithm or techniques to recover lost data.

Failure Classification

To see where the problem has occurred, we generalize a failure into various categories, as follows –

a) **Transaction failure**

- A transaction has to abort when it fails to execute or when it reaches a point from where it can't go any further.
- This is called transaction failure where only a few transactions or processes are hurt.

5. Failure Classification

Reasons for a transaction failure could be –

Logical errors – Where a transaction cannot complete because it has some code error or any internal error condition.

System errors – Where the database system itself terminates an active transaction because the DBMS is not able to execute it, or it has to stop because of some system condition. For example, in case of deadlock or resource unavailability, the system aborts an active transaction.

b) System Crash

- There are problems – external to the system – that may cause the system to stop abruptly and cause the system to crash.
- For example, interruptions in power supply may cause the failure of underlying hardware or software failure.
- Examples may include operating system errors.

5. Failure Classification

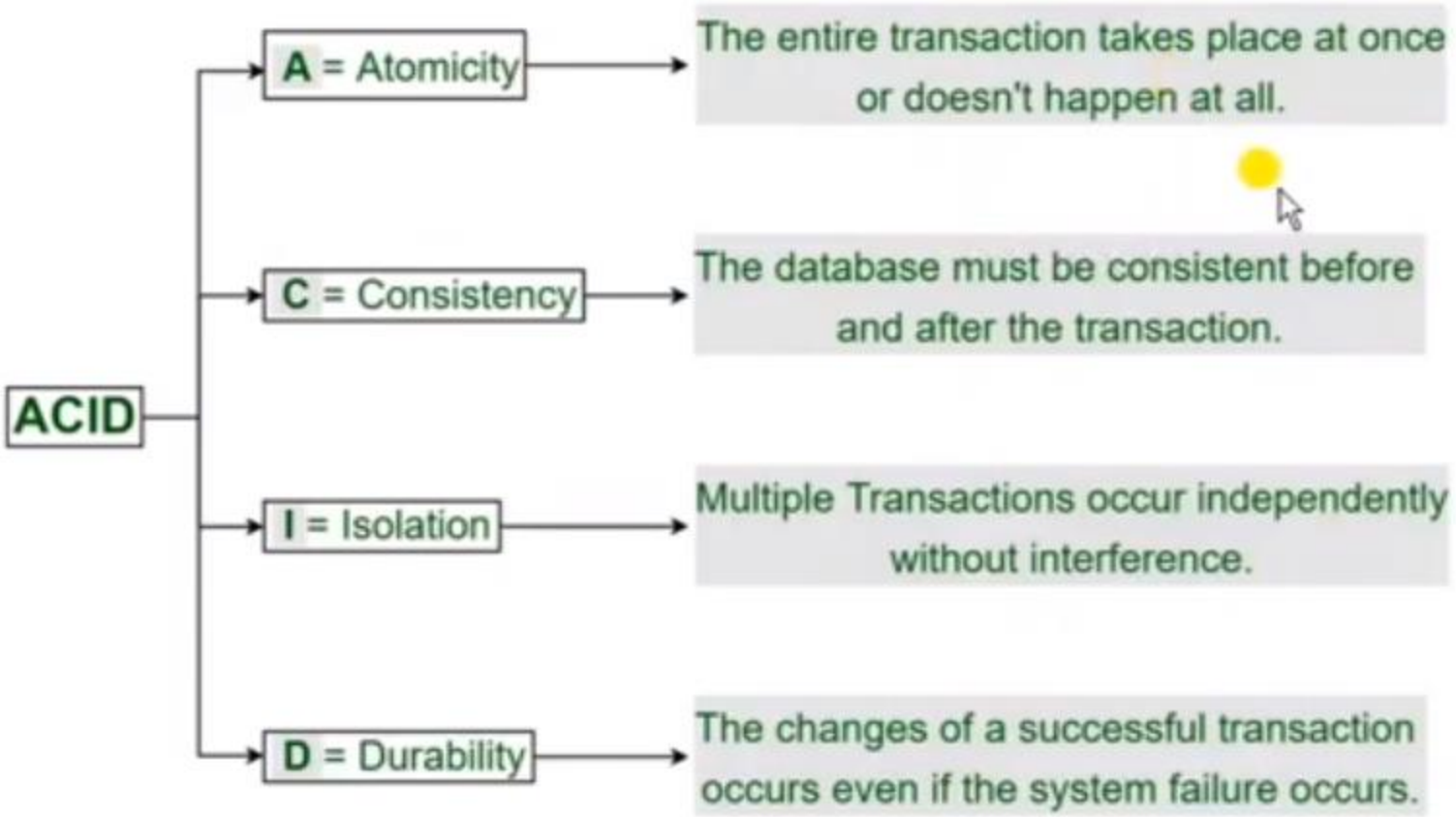
c) Disk Failure

- In early days of technology evolution, it was a common problem where hard-disk drives or storage drives used to fail frequently.
- Disk failures include formation of bad sectors, unreachability to the disk, disk head crash or any other failure, which destroys all or a part of disk storage.

6. Storage Structure

1) **Volatile storage:**
   - ✓ does not survive system crashes
   - ✓ examples: main memory, cache memory .
   - ✓ They are fast but can store only a small amount of information.
2) **Nonvolatile storage:**
   - ✓ survives system crashes
   - ✓ examples: disk, tape, flash memory, non-volatile (battery backed up) RAM.
   - ✓ They are huge in data storage capacity, but slower in accessibility.
3) **Stable storage:**
   - ✓ a mythical form of storage that survives all failures
   - ✓ approximated by maintaining multiple copies on distinct nonvolatile media

ACID Properties in DBMS



7. Recovery and Atomicity

- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
- Consider transaction Ti that transfers $50 from account A to account B; goal is either to perform all database modifications made by Ti or none at all.
- Several output operations may be required for Ti (to output A and B). A failure may occur after one of these modifications have been made but before all of them are made.
- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- Two approaches to ensure Atomicity and Durability are :
  1) log-based recovery, and
  2) shadow-paging

8. Log Based Recovery

- A log is kept on stable storage.
- The log is a sequence of log records, and maintains a record of update activities on the database.
- When transaction Ti starts, it registers itself by writing a <Ti start>log record ! Before Ti executes write(X), a log record <Ti, X, V1, V2> is written, where V1 is the value of X before the write, and V2 is the value to be written to X.
- Log record notes that Ti has performed a write on data item Xj Xj had value V1 before the write, and will have value V2 after the write.
- When Ti finishes it last statement, the log record <Ti commit> is written. ! We assume for now that log records are written directly to stable storage (that is, they are not buffered)
- Two approaches using logs
  1) Deferred database modification
  2) Immediate database modification

8. Log Based Recovery

1) **Deferred database modification** – All logs are written on to the stable storage and the database is updated when a transaction commits.
2) **Immediate database modification** – Each log follows an actual database modification. That is, the database is modified immediately after every operation.

9. Recovery with Concurrent Transactions

Recovery with Concurrent Transactions

- When more than one transaction are being executed in parallel, the logs are interleaved.
- At the time of recovery, it would become hard for the recovery system to backtrack all logs, and then start recovering.
- To ease this situation, most modern DBMS use the concept of 'checkpoints'.

Checkpoint

- Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system.
- As time passes, the log file may grow too big to be handled at all. Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk.
- Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.

9. Recovery with Concurrent Transactions

Recovery

When a system with concurrent transactions crashes and recovers, it behaves in the following manner –

1) The recovery system reads the logs backwards from the end to the last checkpoint.
2) It maintains two lists, an undo-list and a redo-list.
3) If the recovery system sees a log with $<T_n$, Start> and $<T_n$, Commit> or just $<T_n$, Commit>, it puts the transaction in the redo-list.
4) If the recovery system sees a log with $<T_n$, Start> but no commit or abort log found, it puts the transaction in undo-list.

All the transactions in the undo-list are then undone and their logs are removed. All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs.