

7. Define transaction and its properties. Explain the two-phase locking protocol for concurrency control in DBMS. [1+4+3]
8. How does log based recovery method work? [4+4]

7. a) Consider a schedule S: r1(y); r3(z); w1(y); w2(z); r3(y); w2(y). State whether it is conflict serializable schedule or not. Determine the equivalent serial schedule if it is serializable. [4]
- b) Explain how graph based protocol maintains concurrent execution of transactions. [4]

Briefly explain ACID properties of a transaction. How two-phase locking protocol does ensures conflict-serializable schedule? [3+4]

How does the check point recovery method work? [4+4]

- a) During execution, a transaction passes through several states until it finally commits or aborts. Explain all possible sequences of states through which a transaction may pass. [5]
- b) Explain the following briefly: [2×2]
- (i) Two phase locking protocol
 - (ii) Wait & die scheme for deadlock prevention

- a) What is transaction? What are the ideal properties of a transaction? [4]
- b) Describe strict two-phase locking protocol (2PL). [1+4]
- Define term Recovery and Atomicity in database. [3]

- a) What are the possible transaction states? Which of the following schedule is conflict serializable? For each serializable schedule, determine the equivalent serial schedule. [1+3]
- (i) r1(X);r3(X);w1(X);r2(X);w3(X) [2+4]
- (ii) r1(X);r3(X);w3(X);w1(X);r2(X)
- b) How deadlocks arises in transaction processing? [2]

Define transaction and explain various states of a transaction with a transition diagram. Describe about two phase locking protocol for concurrent transaction along with its limitations. [4+4]

Explain the possible transaction states in DBMS. Explain the concept of conflict serializability with an example. [4+4]

Explain ACID properties of a database transaction. Describe how conflict serializability differs from the view serializability for concurrent execution of transactions. [4+4]

7. a) Define ACID properties of a transaction. Describe the concept of conflict serializability for concurrent execution of transactions. [4+4]
- b) How two phase looking protocol helps in concurrency control? Explain. [4]

- c) Explain about the concept of view serializability for concurrent execution of transaction. [2+4]
- b) How deadlocks arise while processing transactions? Explain the deadlock prevention strategies. [2+4]

Describe the different types of locks used for concurrency control. Draw the lock compatibility matrix.

d) B-tree structure used for indexing. [4+4]

Explain different states of a transaction along with state transition diagram. Explain conflict Serializability with example. [4+4]

Explain briefly two phase locking protocol for Concurrency Control. [4]

a) What is a transaction? What are the properties a transaction should satisfy in a database system? [1+3]

b) What do you mean by serializability of a schedule? What do you understand by granularity of locking for concurrency control? [2+2]

8. Explain Atomicity and Isolation properties of a database transaction. Describe the concept of conflict serializability for concurrent execution of transactions. [4+4]

8. a) List the ACID properties. Explain the usefulness of each. [4]

b) During its execution, a transaction passes through several states, until it finally commits or aborts. List all possible sequences of states through which a transaction may pass. Explain why each state transition may occur. [4]

c) How two phase locking protocol helps in avoiding deadlock? Explain with examples. [4]

a) Explain conflict serializability with example. [8]

b) Differentiate between fine granularity and coarse granularity locking in multiple granularity locking protocol. [4]

Transaction

A transaction is a logical unit of work that comprises one or more database operations (such as insert, delete, or update). A transaction ensures that these operations either all succeed (commit) or all fail (rollback), thus maintaining the integrity of the database.

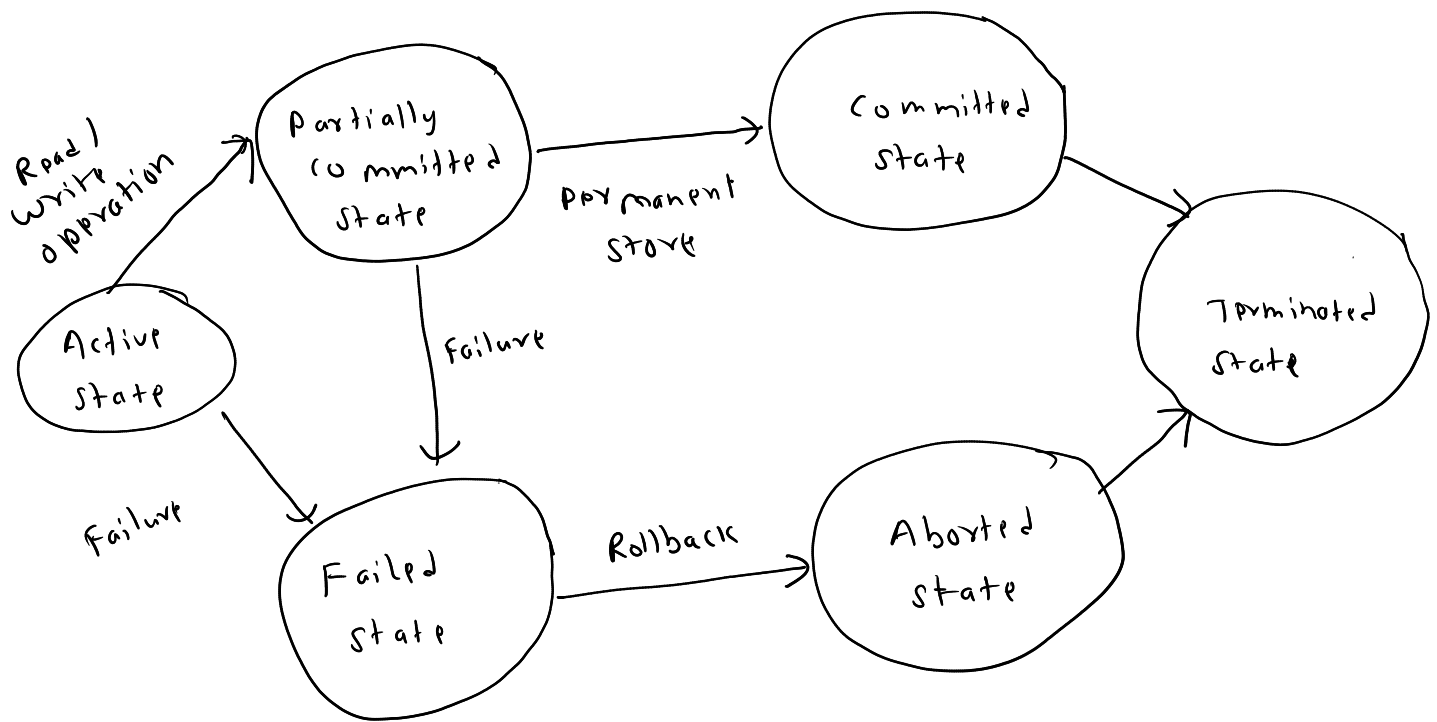


Fig : states of Transaction in DBMS

1. Active State

- > The transaction is initially in the active state when it starts executing.
- > It remains active as long as it is executing its operations (e.g., reading or writing data in the database).

2. Partially Committed State

- > Once the transaction completes its final operation (e.g., the last database update), it moves to the partially committed state.
- > Here, the changes may not yet be permanently recorded in the database.

-> The system now has to ensure that all the effects of the transaction are properly written to disk and made durable.

3. Committed State

-> If the transaction successfully completes all operations and the system ensures the changes are safely stored, the transaction moves to the committed state.

-> At this point, the effects of the transaction are guaranteed to be permanently applied in the database (they survive any subsequent failures).

4. Terminated State

-> Once a transaction is either committed or aborted, it is said to be terminated.

-> A terminated transaction is essentially out of the system's control; the DBMS does not keep track of it further in terms of execution.

5. Failed State

-> If at any point during the active state the transaction encounters an error, it enters the failed state.

-> No further operations of the transaction will be carried out in this state.

6. Aborted State

-> After a failure, the transaction typically goes through an abort or rollback process.

-> In the aborted state, any changes made by the transaction are undone (rolled back), restoring the database to the state before the transaction began.

-> Depending on the DBMS and the application, an aborted transaction may be restarted or it may simply be terminated.

This model helps ensure the ACID properties of transactions.

ACID properties

-> ACID is an acronym that stands for Atomicity, Consistency, Isolation, and Durability. These properties ensure that database transactions are processed reliably and maintain the integrity of the database even in the face of errors, power failures, or other unexpected events.

1. Atomicity

- > Atomicity guarantees that each transaction is treated as a single, indivisible unit. This means that either all the operations within the transaction are executed successfully, or none of them are.
- > If any part of the transaction fails, the entire transaction is rolled back, leaving the database unchanged. This prevents the database from ending up in a partial or inconsistent state.

2. Consistency

- > Consistency ensures that a transaction brings the database from one valid state to another valid state, following all defined rules, constraints, and triggers.
- > After a transaction is executed, the database remains in a state that adheres to all its integrity constraints. Any transaction that violates these constraints is not allowed to commit.

3. Isolation

- > Isolation guarantees that concurrent transactions do not interfere with each other. Each transaction executes as if it were the only transaction running in the system.
- > Even when transactions are executed concurrently, the final outcome is the same as if they were processed sequentially.

4. Durability

- > Durability ensures that once a transaction is committed, its changes are permanently recorded in the database. The results of the transaction survive any subsequent system failures.
- > Even in the event of a crash or power failure, committed transactions will not be lost.

Concurrent Execution

- > Multiple transactions (or database operations) run at the same time. This interleaving of operations allows the system to execute more than one transaction simultaneously.
- > Instead of one transaction waiting for another to finish, their operations are mixed (or interleaved) so that they can all make progress.

Concurrency Control

- > Concurrency control is the process of managing multiple transactions executing at the same time without them interfering with one another. It ensures that the database remains consistent and that the transactions do not conflict with each other.
- > Without proper control, simultaneous transactions might lead to inconsistencies (for example, if two transactions try to update the same piece of data at once).

Advantages of Concurrent Execution

- > Better Use of Resources
- > Reduced Waiting Time

Schedules

- > A schedule is a plan or order in which the operations (like read, write, commit, or abort) of one or more transactions are executed.
- > A schedule contains all instructions from each transaction.
- > If a transaction finishes correctly, its last instruction will be a commit.
- > If something goes wrong, its last instruction will be an abort.

Types of Schedules

1. Serial Schedule

- > Each transaction runs one after the other, without any overlapping operations from different transactions.
- > There is no interference between transactions because they are completely separate.
- > If Transaction 1 (T1) is executed fully and then Transaction 2 (T2) is executed, it's a serial schedule.

T ₁	T ₂
Read (A) $A = A + 20$ Write (A) Read (B) $B = B + 50$ Write (B)	
	Read A temp = A * 4 $A = A - \text{temp}$ Write (A)

(A serial schedule where T₁ is followed by T₂)

T ₁	T ₂
	Inst ₁
Inst ₁	

T₂
followed by T₁

2. Non-Serial Schedule

- > The operations of the transactions are interleaved (mixed together) but in such a way that the final effect on the database is the same as if the transactions had been run serially.
- > Even though T1 and T2 have their operations mixed, if the end result is the same as if T1 was executed completely first and then T2, then the non-serial schedule is considered equivalent to a serial schedule.
- > A schedule might interleave operations in a way that the final result is incorrect or inconsistent. For instance, if a schedule causes the sum of two values (like $A+B$) not to be preserved, then it is considered a faulty schedule.

T ₁	T ₂
Read A $A = A - 50$ Write (A)	
	Read(A) Temp = A * 0.1 $A = A - 0.1$ Write (A)
Read(B) $B = B + 50$ Write (B)	
	Read B $B = B + \text{temp}$ Write (B)

← non serial schedule

T₁ and T₂ if
serially executed
the output is
same as serial
(preserved value at A
and B from being lost
midway)

Serializability

-> Serializability is a concept in database concurrency control that ensures correctness in executing transactions. A schedule (sequence of transactions) is serializable if it is equivalent to some serial schedule, meaning transactions execute one after another without interleaving.

Types of Serializability

There are two main types of serializability:

- > Conflict Serializability
- > View Serializability

The main objective of serializability is to allow transactions to execute concurrently without interference while ensuring that the database state remains consistent and equivalent to a serial execution.

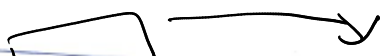
1. Conflict Serializability

-> A schedule is conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.

Conflicting Operations

Two operations conflict if:

- > They belong to different transactions.
- > They operate on the same data item.
- > At least one of them is a WRITE operation.



$I_1 = \text{read}(Q), I_2 = \text{read}(Q)$	I_1 and I_2 don't conflict
$I_1 = \text{read}(Q), I_2 = \text{write}(Q)$	They conflict
$I_1 = \text{write}(Q), I_2 = \text{read}(Q)$	They conflict
$I_1 = \text{write}(Q), I_2 = \text{write}(Q)$	They conflict

example

T ₁	T ₂
R(x)	
w(x)	
	w(x)

conflicting operation

w(x) (T₁) conflict
w(x) (T₂)
swap
order
changes

→ This schedule cannot be swapped into a serial order, meaning it is not conflict serializable.

T ₁	T ₂
R(A)	
w(A)	
	R(A)
	w(A)

Not
conflicting
operations

R(B)
w(B)

w(A) T₁ →
conflict with
R(A) (T₂)
w(A) T₁ →
conflict
w(A) (T₂)

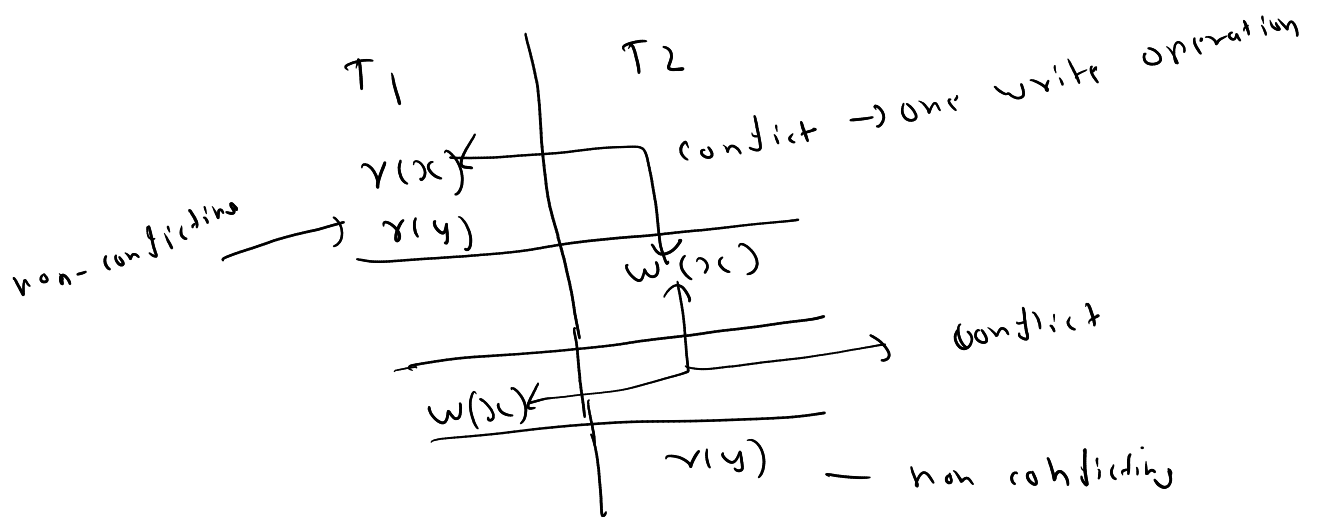
If we swap non-conflicting operations, we get:

T ₁	T ₂
R(A)	
w(A)	
R(B)	
w(B)	
	R(A)
	w(A)

This is equivalent to serial execution T₁ → T₂, so it is conflict serializable.

Precedence graph theory for testing conflict serializability

① $S_1 : r_1(x) r_1(y) w_2(x) w_1(x) r_2(y)$



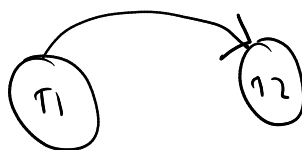
if at least one write operation then conflict

(no conflict between read and read)

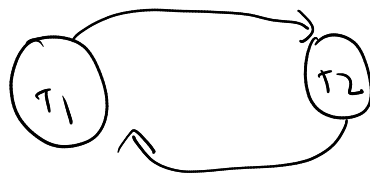
① make two nodes for each transaction



② For the conflicting $r_1(x) w_2(x)$, $r_1(x)$ happens before so draw edge from T_1 to T_2



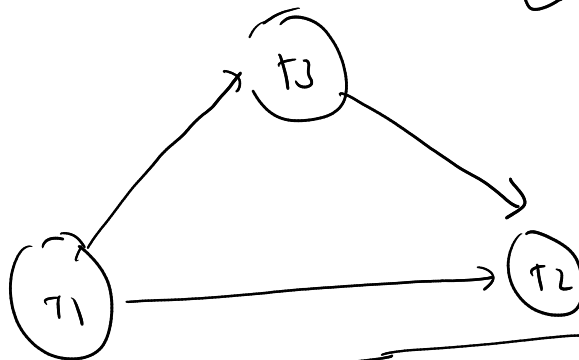
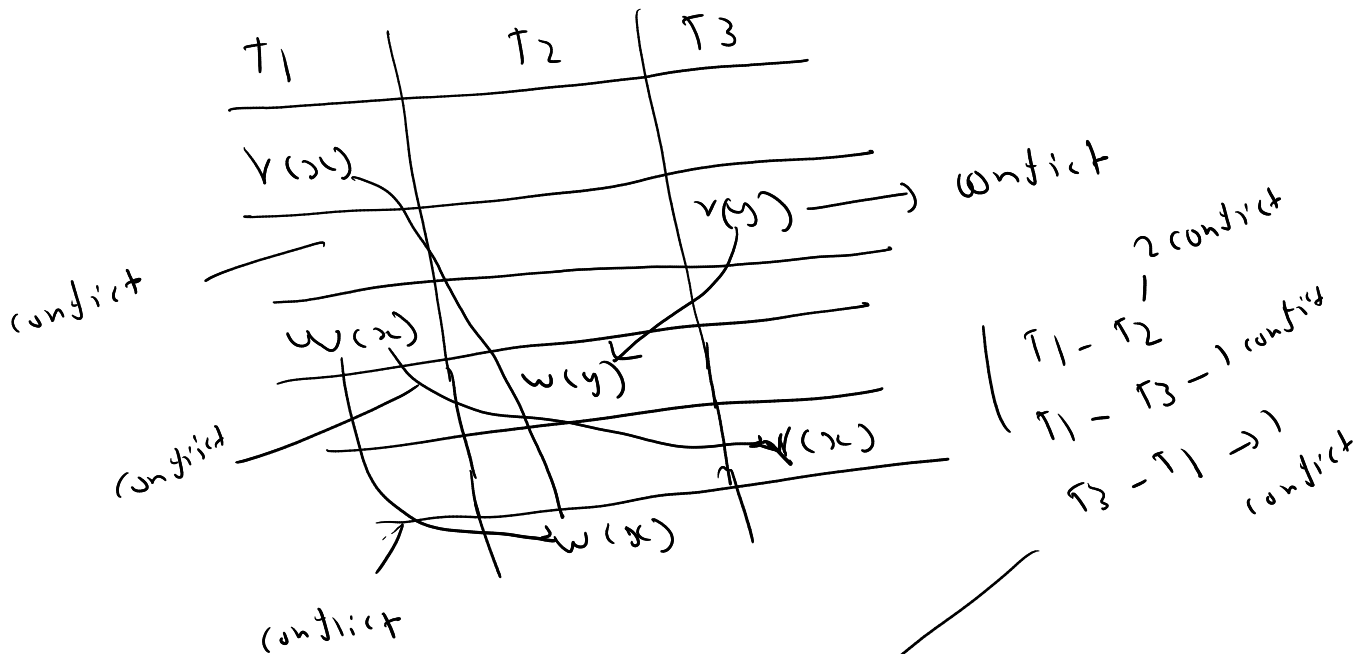
③ For the conflicting pair $w_2(x) w_1(x)$ where $w_2(x)$ happens before $w_1(x)$ draw edge from T_2 to T_1



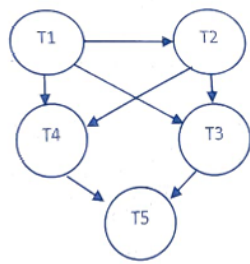
→ since the graph is cyclic, we conclude that it is not conflict serializable.

Cyclic → not conflict serializable

$\{x : 2 \text{ } S_1 : r_1(x) r_3(y) w_1(x), w_2(y), r_3(x), w_2(x)\}$



→ not cyclic so it is conflict serializable.



$T_1 \rightarrow T_2$

$T_1 \rightarrow T_4$

$T_2 \rightarrow T_3$

$T_2 \rightarrow T_4$

$T_3 \rightarrow T_5$

$T_4 \rightarrow T_5$

$T_1 \rightarrow T_2 \rightarrow T_3$

$(T_1 \rightarrow T_4)$

$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_5$

cyclic

not cyclic does not reach T_1 at last
so conflict serializable

2. View Serializability

-> A schedule is view serializable if it is view-equivalent to a serial schedule.

View Equivalence

Two schedules are view-equivalent if they satisfy the following conditions:

-> **Initial Read:** If a transaction T_1 reads a value from the database in one schedule, it must read the same initial value in the other schedule.

-> **Read-Write Dependency:** If a transaction T_1 reads a value written by another transaction T_2 in one schedule, then T_1 must read the same value from the same transaction T_2 in the other schedule.

-> **Final Write:** The transaction that performs the last write on a data item in one schedule must be the same as in the other schedule.

If a schedule satisfies these conditions with some serial schedule, it is view serializable.

example:

Schedule (S)

T1	T2
R(A)	
	W(A)
R(A)	
	W(A)

Serial schedule (S')

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)

- T1 reads A in both
- T1 reads A written by T2 in both
- T2 writes A at the end in both schedules

Differences between conflict serializability and view serializability:

- Conflict serializability is easy to achieve but view serializability is difficult to achieve.
- Every conflict serializable schedule is view serializable but the reverse is not true.
- It is easy to test conflict serializability but expensive to test view serializability.
- Most of the concurrency control schemas used in practice are conflict serializability.

Locked Base Protocol

-> The Locked Base Protocol is a concurrency control mechanism used in databases to ensure data consistency when multiple transactions access the same data. It uses locking mechanisms to manage concurrent transactions, preventing conflicts like dirty reads, lost updates, or uncommitted dependencies.

Types of Locks

1. Shared Lock (S-lock)

- > Allows multiple transactions to read a data item simultaneously.
- > No transaction can modify the data while a shared lock is held.
- > Other transactions can also acquire a shared lock on the same data.

2. Exclusive Lock (X-lock)

- > Allows only one transaction to access the data for both read and write.
- > Prevents other transactions from acquiring any lock (shared or exclusive) on the same data item.
- > Ensures that no other transaction reads or modifies the data while the lock is held.

Two-Phase Locking (2PL)

- > 2PL is a concurrency control protocol that guarantees a schedule is conflict serializable. This means that even though transactions may interleave their operations, the final outcome is the same as if the transactions were executed one after the other.

2PL divides the execution of a transaction into two distinct phases:

1. Growing Phase:

- > In this phase, the transaction is allowed to request and acquire locks on data items
- > During the growing phase, no locks are released. The transaction collects all the locks it will need before starting to release any.
- > It can "upgrade" a lock (e.g., from Shared to Exclusive) if it needs to write after reading.

2. Shrinking Phase:

- > Once the transaction releases its first lock, it enters the shrinking phase.
- > In this phase, the transaction can only release locks; it cannot acquire any new ones.
- > It can "downgrade" a lock (e.g., from Exclusive to Shared) if allowed.

Example of 2PL

Let's say we have two transactions, T1 and T2, working on two bank accounts, A and B:

T1: Transfer \$50 from A to B.

T2: Check the balance of A and B.

Steps:

T1: Growing Phase:

- > T1 locks A with an Exclusive lock (X(A)) to subtract \$50.
- > T1 locks B with an Exclusive lock (X(B)) to add \$50.

T2: Growing Phase:

- > T2 wants to read A and B, so it requests Shared locks (S(A) and S(B)).
- > But T1 already has X(A) and X(B), so T2 waits.

T1 Shrinking Phase:

- > T1 finishes the transfer, releases $X(A)$, then $X(B)$.
- > Now the shrinking phase starts—no new locks for T1.

T2 Continues:

- > T2 gets $S(A)$ and $S(B)$, reads the balances, and releases the locks.

Timeline:

T1: Lock $X(A)$ → Lock $X(B)$ → Update A and B → Release $X(A)$ → Release $X(B)$.

T2: Waits → Lock $S(A)$ → Lock $S(B)$ → Read → Release $S(A)$ → Release $S(B)$.

This ensures T2 sees the updated balances after T1 is done, avoiding confusion.

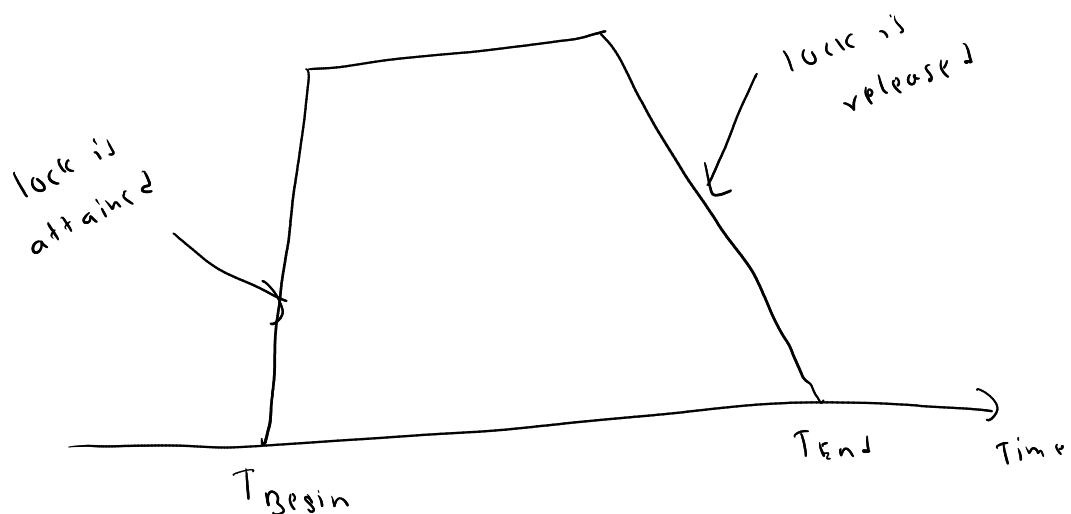


Fig: two phase locking

Strict Two-Phase Locking (Strict-2PL)

-> Strict-2PL is a variant of the basic 2PL protocol. It adds a stronger rule regarding when locks can be released, which helps to eliminate a problem called cascading aborts—where one transaction's failure might force other transactions to roll back.

How Does Strict-2PL Work?

1. Growing Phase (Similar to 2PL):

-> The transaction acquires locks as needed, just like in the standard 2PL.

2. Lock Holding Until Commit:

-> No Early Release: Once a lock is acquired, it is held until the transaction commits or aborts.

-> Single Release Point: All locks are released at the very end of the transaction, rather than gradually.

Example of Strict-2PL

Consider a transaction T3 that needs to update data items A and B:

- > T3 begins and requests locks on A and B.
- > It acquires the necessary locks (e.g., X lock for writing) on both items.
- > T3 performs its read and write operations while holding the locks.
- > Once T3 has finished all its operations, it commits the transaction.
- > All locks are released at commit time.

Advantage:

Because T3 holds all its locks until commit, if it aborts, no other transaction would have read uncommitted data. This means that cascading aborts (where one transaction's rollback forces others to rollback) are prevented.

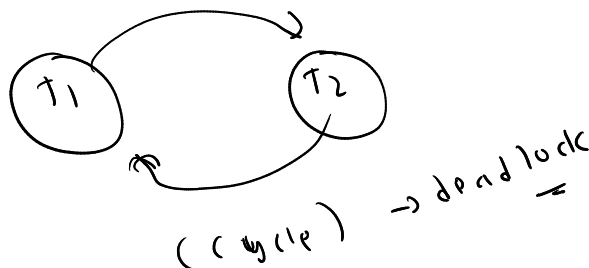
2. **Hold and Wait:** A transaction is holding at least one resource and is waiting to acquire additional resources held by other transactions.
3. **No Preemption:** Resources cannot be forcibly taken away from a transaction; they must be released voluntarily.
4. **Circular Wait:** A cycle of transactions exists, where each transaction waits for a resource held by the next transaction in the cycle.

Because T1 is waiting for the lock on Grade, which is held by T2, and T2 is waiting for the lock on Student, which is held by T1, neither transaction can proceed. This forms a circular wait—a classic deadlock.

Deadlock Detection and avoidance

1. **Wait-for Graph (WFG):**

- > The DBMS constructs a directed graph where each node represents a transaction.
- > A directed edge from T1 to T2 means T1 is waiting for a resource locked by T2.
- > If there is a cycle in this graph, a deadlock exists.
- > Once a cycle is detected, the DBMS picks a “victim” transaction to roll back. Often the victim is chosen based on criteria such as smallest amount of work done or least priority.
- > After rolling back the victim, its locks are released, allowing the other transactions to proceed.



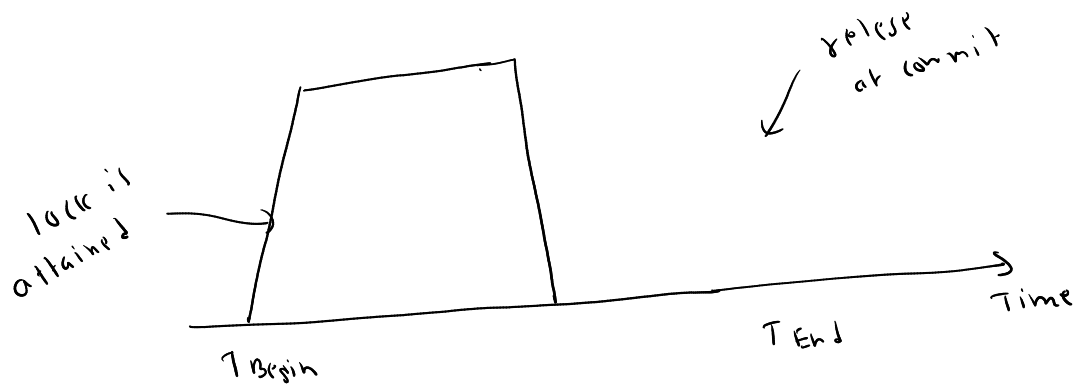


Fig: Strict two-phase locking

Deadlock

-> A deadlock occurs in a DBMS when two or more transactions are waiting indefinitely for each other to release locks on resources. In other words, each transaction holds a lock on some resource(s) that the other transaction(s) needs, and they form a cycle of waiting. No transaction can proceed, because no one is willing to release the resource it already holds.

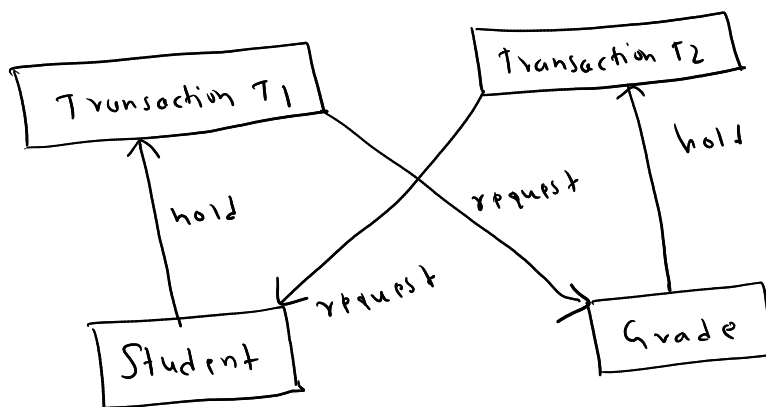


Fig: Deadlock

Key Conditions for Deadlock

1. **Mutual Exclusion:** Resources (data items) are locked in an exclusive mode —only one transaction can have the lock at a time.

Deadlock Prevention

1. Wait-Die Scheme

- > Transactions are assigned timestamps when they start. The idea is to let older transactions (with earlier timestamps) wait for younger ones, while younger transactions that request resources held by older ones are aborted (or "die").
- > If an older transaction requests a resource that is held by a younger transaction, it is allowed to wait.
- > If a younger transaction requests a resource held by an older transaction, it is aborted immediately. The younger transaction will later restart with a new timestamp. ●
- > This method prevents circular waiting by ensuring that transactions only wait in one direction (older waiting for younger), thus avoiding a deadlock cycle.

2. Wound-Wait Scheme

- > This scheme also uses timestamps but reverses the waiting logic compared to the Wait-Die scheme.
- > If an older transaction requests a resource that is currently held by a younger transaction, the younger transaction is aborted (or "wounded") immediately, freeing up the resource for the older one.
- > If a younger transaction requests a resource held by an older transaction, the younger transaction is allowed to wait.

3. Timestamp Ordering

- > This method uses the idea of ordering transactions based on the time they were initiated. Each transaction receives a unique timestamp.
- > Transactions are executed in order based on their timestamps. When a conflict arises (for example, two transactions want to access the same resource), the system compares timestamps.

-> The transaction with the earlier timestamp is given priority, and the one with the later timestamp is forced to wait or may even be rolled back.