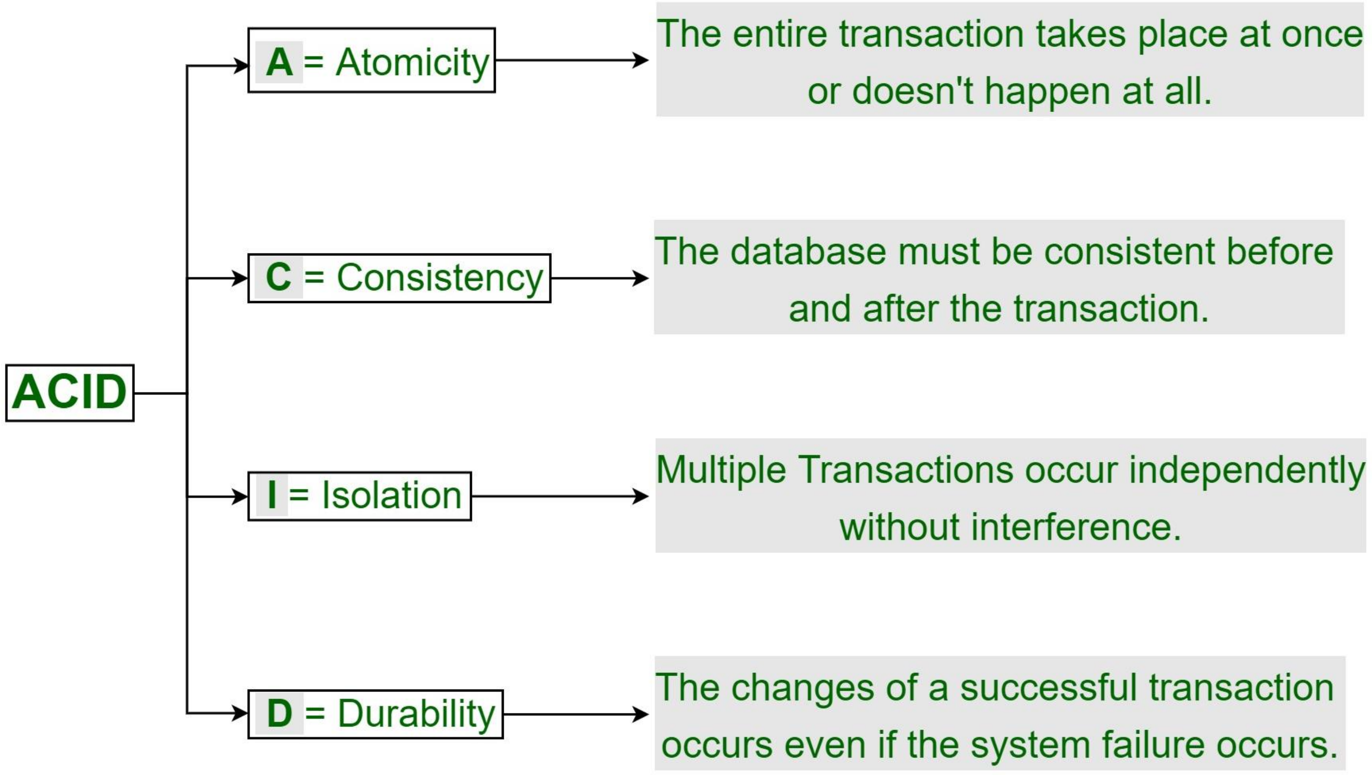
Serializability

CS 341 Database Systems

ACID Properties of Transactions



Recovery Mechanism

To recover the state of the database we can use:

- A log file recording every database operation.
- Checkpoints recording the state of all active transactions.
  - Then: develop an algorithm for transactions to **UNDO**,
  - and those that we need to **REDO**, to effect recovery.
  - at intervals, the system will:
    - Flush its buffers – buffers are forced to write changes to secondary storage.
    - Write out a checkpoint record to log indicating which transactions are in *progress*.

Three classic problems

Problem: Two or more transactions read / write on the same part of the database.

Although transactions execute correctly, results may **interleave** in different ways => 3 classic problems.

- **Lost Update**
- **Uncommitted Dependency**
- **Inconsistent Analysis**

Read Write Conflicts

Table 10.11 Read/Write Conflict Scenarios: Conflicting Database Operations Matrix

	Transactions		
	T1	**T2**	**Result**
Operations	Read	Read	No conflict
	Read	Write	Conflict
	Write	Read	Conflict
	Write	Write	Conflict

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	read(bal_x)	100
t_3	read(bal_x)	$bal_x = bal_x + 100$	100
t_4	$bal_x = bal_x - 10$	write(bal_x)	200
t_5	write(bal_x)	commit	90
t_6	commit		[illegible]

Lost Update WW

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		read(bal_x)	100
t_3		$bal_x = bal_x + 100$	100
t_4	begin_transaction	write(bal_x)	200
t_5	read(bal_x)	:	200
t_6	$bal_x = bal_x - 10$	rollback	100
t_7	write(bal_x)		190
	commit		190

Uncommitted Dependency WR

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	read(bal_x)	read(bal_x)	100	50	25	0
t_4	$bal_x = bal_x - 10$	sum = sum + bal_x	100	50	25	100
t_5	write(bal_x)	read(bal_y)	90	50	25	100
t_6	read(bal_z)	sum = sum + bal_y	90	50	25	150
t_7	$bal_z = bal_z + 10$		90	50	25	150
t_8	write(bal_z)		90	50	35	150
t_9	commit	read(bal_z)	90	50	35	150
t_{10}		sum = sum + bal_z	90	50	35	185
t_{11}		commit	90	50	35	185

Inconsistent Analysis RW

ANSI/ISO Transaction Isolation Levels

- Transaction isolation levels refer to the degree to which transaction data is "**protected or isolated**" from other ***concurrent*** transactions.
- The isolation levels are described based on what data other transactions can see (read) during execution. More precisely, the *transaction isolation levels are described by the type of "reads" that a transaction allows or does not allow.*

Types of Read Operations

- Dirty reads
- Non-repeatable (fuzzy) reads
- Phantom reads

Preventable Read Phenomena by Isolation Level

Table 10.15 Transaction Isolation Levels

	Isolation Level	Allowed			Comment
		Dirty Read	Nonrepeatable Read	Phantom Read	
Less restrictive	Read Uncommitted	Y	Y	Y	Reads uncommitted data, and allows nonrepeatable reads and phantom reads.
	Read Committed	N	Y	Y	Does not allow uncommitted data reads but allows nonrepeatable reads and phantom reads.
	Repeatable Read	N	N	Y	Only allows phantom reads.
More restrictive	Serializable	N	N	N	Does not allow dirty reads, nonrepeatable reads, or phantom reads.
Oracle/SQL Server Only	Read Only/Snapshot	N	N	N	Supported by Oracle and SQL Server. The transaction can only see the changes that were committed at the time the transaction started.

Exclusive VS Shared Locks

Exclusive Lock

- An **exclusive lock** exists when access is reserved specifically for the transaction that locked the object. The exclusive lock must be used when the potential for conflict exists.
- An exclusive lock is issued when a transaction wants ***to update (write)*** a data item and no locks are currently held on that data item by any other transaction.

Shared Lock

- A **shared lock** exists when concurrent transactions are granted read access on the basis of a common lock. A shared lock produces no conflict as long as all the concurrent transactions are read-only.
- A shared lock is issued when a transaction wants to **read data** from the database and no exclusive lock is held on that data item.

Solved: Lost Update

Time	User 1 (Trans A)	User2 (Trans B)
1	Retrieve t (get S-lock on t)	
2		Retrieve t (get S-lock on t)
3	Update t (request X-lock on t)	
4	wait	Update t (request X-lock on t)
5	wait	wait
6	wait	wait
7		

- No update lost but → Deadlock

Solved:
Uncommitted Dependency

Time	User 1 (Trans A)	User 2 (Trans B)
1		Update t (get X-lock on t)
2	Retrieve t (request S-lock on t)	-
3	wait	-
4	wait	-
5	wait	Commit / Rollback (releases X-lock on t)
6	Resume: Retrieve t (get S-lock on t)	
7	-	
8		

Solved: Inconsistent Analysis

Time	User 1 (Trans A)	User 2 (Trans B)
1	Retrieve Acc1 : (get S-lock) Sum = 40	
2	Retrieve Acc2 : (get S-lock) Sum = 90	
3		Retrieve Acc3: (get S-lock)
4		Update Acc3: (get X-lock) 30 → 20
5		Retrieve Acc1: (get S-lock)
6		Update Acc1: (request X-lock) wait
7	Retrieve Acc3: (request S-lock) wait	wait wait wait

- Inconsistent Analysis is prevented → Deadlock

Serializability

Serializability

- Objective of a concurrency control protocol is **to schedule transactions** in such a way as to avoid any interference.
- Possible solution is to ***Run all transactions serially.*** This is often too restrictive as *it limits degree of concurrency or parallelism in system.*

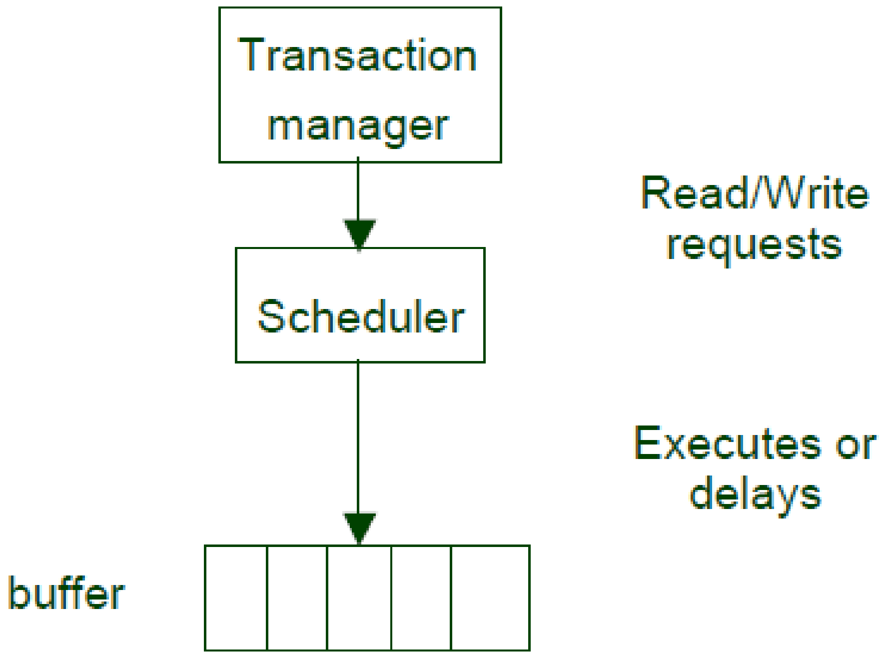
Serializability *identifies those executions of transactions that are guaranteed to ensure database consistency.*

Schedules

- **Schedule** – *a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed*
  - A schedule for a set of transactions must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - By default, transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

The Scheduler

- The scheduler component of a DBMS must ensure that the individual steps of different transactions preserve consistency.



Serializability - some definitions

- **Schedule:** *time-ordered sequence of the important actions taken by one or more transitions.*
- **Serial Schedule:** *a schedule where the operations of each transaction are executed consecutively without any interleaved operations from other transactions.* No guarantee that results of all serial executions of a given set of transactions will be identical.
- **Non-serial Schedule:** Schedule where operations from a set of concurrent transactions are interleaved.

Serializability - some definitions

- Objective of **serializability** :

  Find *non-serial* schedules that allow transactions to execute concurrently without interfering with one another.

- In other words, find *non-serial schedules that are equivalent to some serial schedule*. Such a schedule is called **serializable**.

Serializability - some definitions

In serializability, ordering of read/writes is important:

a) If two transactions only **read** a data item, they do **not conflict** and order is not important.

b) If two transactions either **read or write completely separate data items**, they do **not conflict** and order is not important.

c) If one transaction **writes a data item and another reads or writes same data item**, they **conflict** and **order of execution is important.**

Conflict Serializability

Time	T_7	T_8
t_1	begin_transaction	
t_2	read($\mathbf{bal}_x$)	
t_3	write($\mathbf{bal}_x$)	
t_4		begin_transaction
t_5		read($\mathbf{bal}_x$)
t_6		write($\mathbf{bal}_x$)
t_7	read($\mathbf{bal}_y$)	
t_8	write($\mathbf{bal}_y$)	
t_9	commit	
t_{10}		read($\mathbf{bal}_y$)
t_{11}		write($\mathbf{bal}_y$)
t_{12}		commit

(a) Schedule S_1

T_7	T_8
begin_transaction	
read($\mathbf{bal}_x$)	
write($\mathbf{bal}_x$)	
	begin_transaction
	read($\mathbf{bal}_x$)
read($\mathbf{bal}_y$)	
	write($\mathbf{bal}_x$)
write($\mathbf{bal}_y$)	
commit	
	read($\mathbf{bal}_y$)
	write($\mathbf{bal}_y$)
	commit

(b) Schedule S_2

T_7	T_8
begin_transaction	
read($\mathbf{bal}_x$)	
write($\mathbf{bal}_x$)	
read($\mathbf{bal}_y$)	
write($\mathbf{bal}_y$)	
commit	
	begin_transaction
	read($\mathbf{bal}_x$)
	write($\mathbf{bal}_x$)
	read($\mathbf{bal}_y$)
	write($\mathbf{bal}_y$)
	commit

(c) Schedule S_3

Figure 22.7 Equivalent schedules: (a) nonserial schedule S_1; (b) nonserial schedule S_2 equivalent to S_1; (c) serial schedule S_3, equivalent to S_1 and S_2.

S2 – swap non-conflicting pairs (Schedule (b))

Time	T7	T8
T1	Begin transaction	
T2	Read (balx)	
T3	Write (balx)	
T4		Begin transaction
T5		Read (balx)
T6	Read (baly)	
T7		**Write (balx)**
T8	**Write (baly)**	
T9	**Commit**	
T10		Read (baly)
T11		Write (baly)
T12		**Commit**

S2 – swap non-conflicting pairs

Time	T7	T8
T1	Begin transaction	
T2	Read (balx)	
T3	Write (balx)	
T4		Begin transaction
T5		Read (balx)
T6	Read (baly)	
T7	**Write (baly)**	
T8	**Commit**	
T9		**Write (balx)**
T10		Read (baly)
T11		Write (baly)
T12		**Commit**

S2 – swap non-conflicting pairs

Time	T7	T8
T1	Begin transaction	
T2	Read (balx)	
T3	Write (balx)	
T4		Begin transaction
T5		**Read (balx)**
T6	**Read (baly)**	
T7	**Write (baly)**	
T8	**Commit**	
T9		**Write (balx)**
T10		Read (baly)
T11		Write (baly)
T12		**Commit**

S2 – swap non-conflicting pairs

Time	T7	T8
T1	Begin transaction	
T2	Read (balx)	
T3	Write (balx)	
T4		
T5	**Read (baly)**	Begin transaction
T6		**Read (balx)**
T7	**Write (baly)**	
T8	**Commit**	
T9		**Write (balx)**
T10		Read (baly)
T11		Write (baly)
T12		**Commit**

S2 – swap non-conflicting pairs

Time	T7	T8
T1	Begin transaction	
T2	Read (balx)	
T3	Write (balx)	
T4		
T5	**Read (baly)**	Begin transaction
T6		**Read (balx)**
T7	**Write (baly)**	
T8	**Commit**	
T9		**Write (balx)**
T10		Read (baly)
T11		Write (baly)
T12		**Commit**

S2 – swap non-conflicting pairs

Time	T7	T8
T1	Begin transaction	
T2	Read (balx)	
T3	Write (balx)	
T4		
T5	**Read (baly)**	
T6	**Write (baly)**	Begin transaction
T7	**Commit**	**Read (balx)**
T8		
T9		**Write (balx)**
T10		Read (baly)
T11		Write (baly)
T12		**Commit**

Serial Schedule

Time	T7	T8
T1	Begin transaction	
T2	Read (balx)	
T3	Write (balx)	
T4	**Read (baly)**	
T5	**Write (baly)**	
T6	**Commit**	
T7		Begin transaction
T8		**Read (balx)**
T9		**Write (balx)**
T10		Read (baly)
T11		Write (baly)
T12		**Commit**

Conflict Serializability

Time	T_7	T_8
t_1	begin_transaction	
t_2	read(**bal**$_x$)	
t_3	write(**bal**$_x$)	
t_4		begin_transaction
t_5		read(**bal**$_x$)
t_6		write(**bal**$_x$)
t_7	read(**bal**$_y$)	
t_8	write(**bal**$_y$)	
t_9	commit	
t_{10}		read(**bal**$_y$)
t_{11}		write(**bal**$_y$)
t_{12}		commit

(a) Schedule S_1

T_7	T_8
begin_transaction	
read(**bal**$_x$)	
write(**bal**$_x$)	
	begin_transaction
	read(**bal**$_x$)
read(**bal**$_y$)	
	write(**bal**$_x$)
write(**bal**$_y$)	
commit	
	read(**bal**$_y$)
	write(**bal**$_y$)
	commit

(b) Schedule S_2

T_7	T_8
begin_transaction	
read(**bal**$_x$)	
write(**bal**$_x$)	
read(**bal**$_y$)	
write(**bal**$_y$)	
commit	
	begin_transaction
	read(**bal**$_x$)
	write(**bal**$_x$)
	read(**bal**$_y$)
	write(**bal**$_y$)
	commit

(c) Schedule S_3

Figure 22.7 Equivalent schedules: (a) nonserial schedule S_1; (b) nonserial schedule S_2 equivalent to S_1; (c) serial schedule S_3, equivalent to S_1 and S_2.

Test for Conflict Serializability
Precedence Graph

- Create a node for each transaction. (T1, T2 ,T3…)
- Create a directed edge **Ti → Tj,**
  - if Tj reads the value of an item written by Ti. **(WR)**
  - if Tj writes a value into an item after it has been read by Ti. **(RW)**
  - if Tj writes a value into an item after it has been written by Ti. **(WW)**
- If an edge Ti → Tj exists in the precedence graph for S, then in any serial schedule equivalent to S, Ti must appear before Tj.
- If the precedence graph contains a **cycle**, the schedule is **not conflict serializable.**

Two Concurrent update transactions that are not conflict serializable

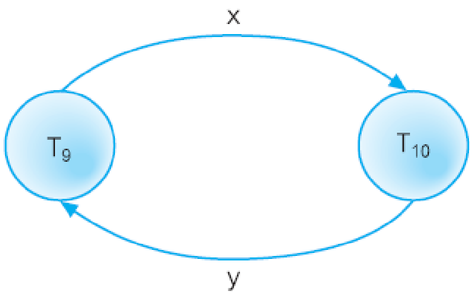
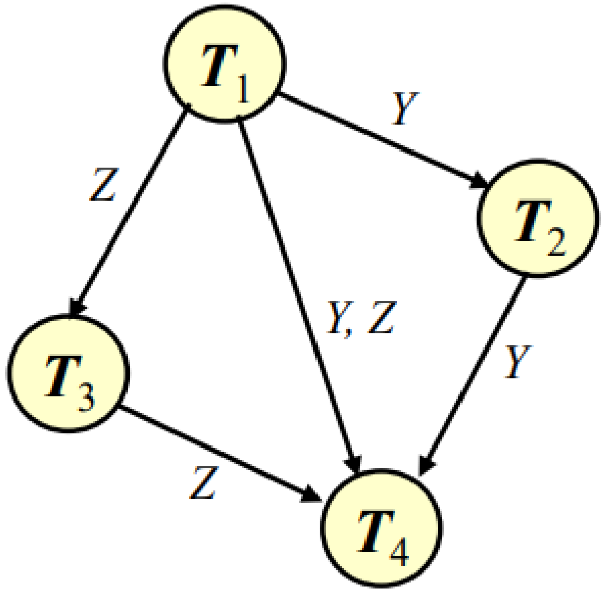


Figure 22.9 Precedence graph for Figure 22.8 showing a cycle, so schedule is not conflict serializable.

Time	T_9	T_{10}
t_1	begin_transaction	
t_2	read(bal_x)	
t_3	$bal_x = bal_x + 100$	
t_4	write(bal_x)	begin_transaction
t_5		read(bal_x)
t_6		$bal_x = bal_x * 1.1$
t_7		write(bal_x)
t_8		read(bal_y)
t_9		$bal_y = bal_y * 1.1$
t_{10}		write(bal_y)
t_{11}	read(bal_y)	commit
t_{12}	$bal_y = bal_y - 100$	
t_{13}	write(bal_y)	
t_{14}	commit	

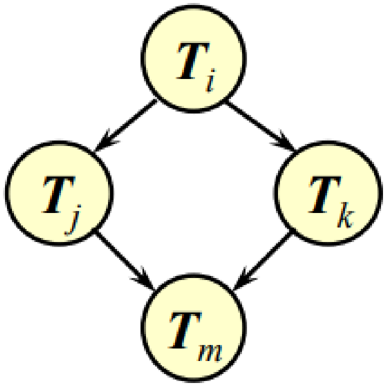
Precedence Graph

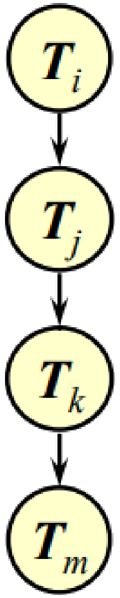
T_1	T_2	T_3	T_4	T_5
	$r(X)$			
$r(Y)$				
$r(Z)$				
				$r(V)$
				$r(W)$
	$r(Y)$			
	$w(Y)$			
		$w(Z)$		
$r(U)$				
			$r(Y)$	
			$w(Y)$	
			$r(Z)$	
			$w(Z)$	
$r(U)$				
$w(U)$				

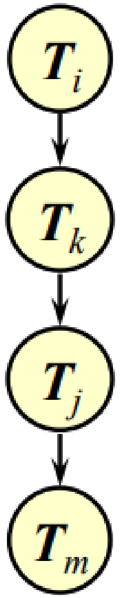


Serializable

- **A schedule is serializable if and only if its precedence graph is acyclic.**
- If precedence graph is acyclic, the serializability order can be obtained by a topological sorting of the graph.
- A topological sort is a linear ordering of vertices in a directed acyclic graph (DAG). Given a DAG G = (V, E), a topological sort algorithm returns a sequence of vertices in which the vertices never come before their predecessors on any paths.





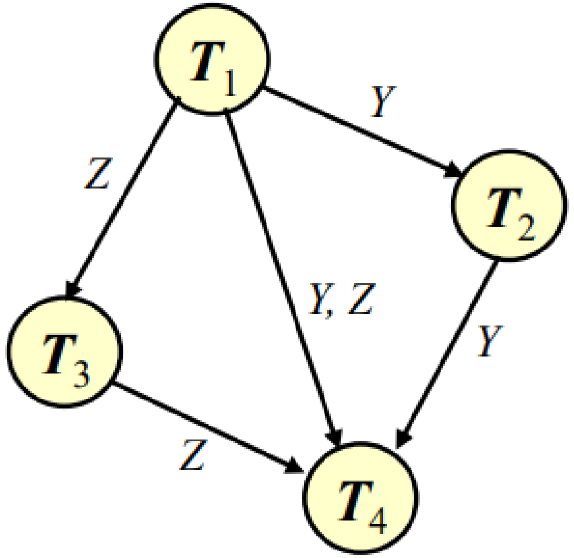


No cycles → Serializable

T1 → T2 → T3 → T4

Or?

T_1	T_2	T_3	T_4	T_5
	$r(X)$			
$r(Y)$				
$r(Z)$				
				$r(V)$
				$r(W)$
	$r(Y)$			
	$w(Y)$			
		$w(Z)$		
$r(U)$				
			$r(Y)$	
			$w(Y)$	
			$r(Z)$	
			$w(Z)$	
$r(U)$				
$w(U)$				



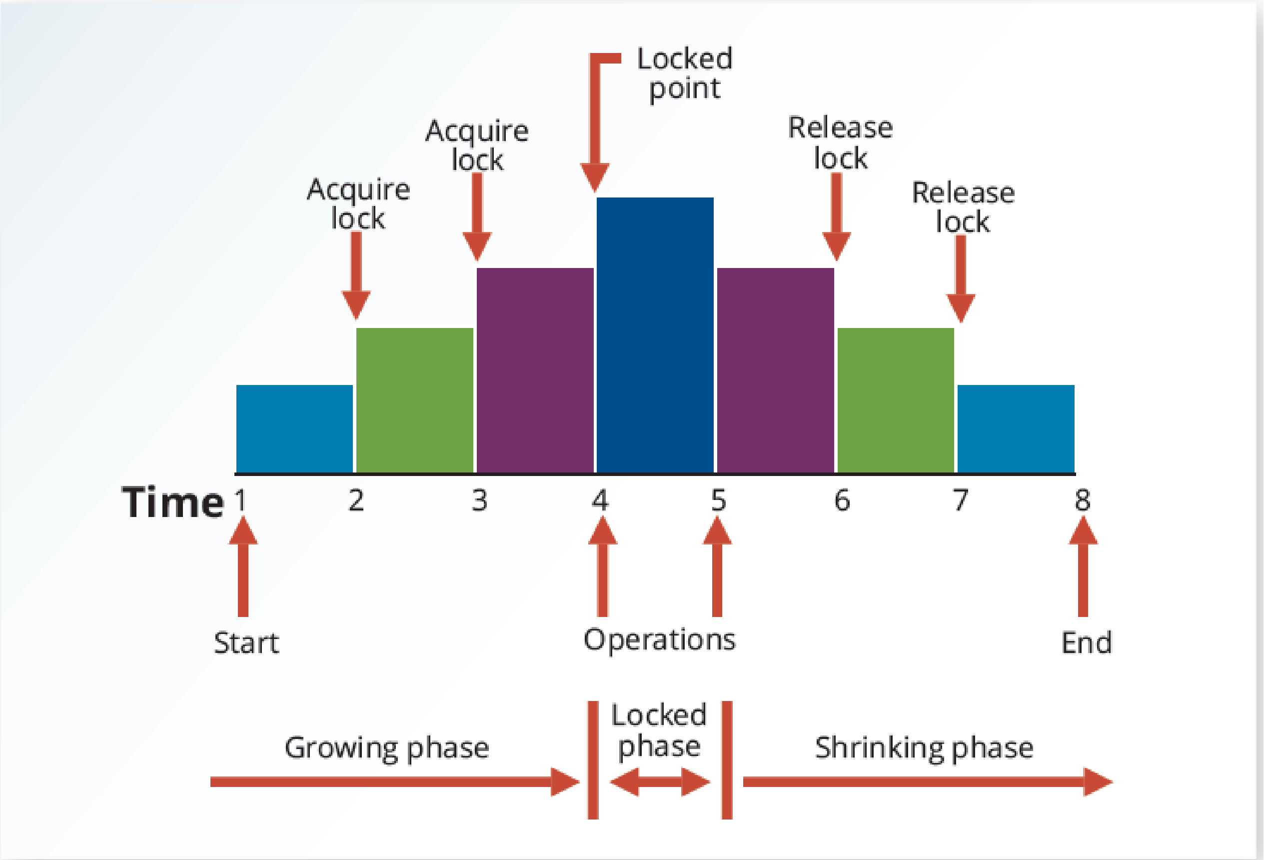
2-Phase Locking Protocol (2PL)

2PL defines how transactions acquire and relinquish locks. It guarantees *serializability*, but it does not prevent deadlocks.

1. A **growing phase**, in which a transaction acquires all required locks without unlocking any data. After all locks have been acquired, the transaction is in its *locked point.*

2. A **shrinking phase**, in which a transaction releases all locks and cannot obtain a new lock.

Figure 10.7 Two-Phase Locking Protocol

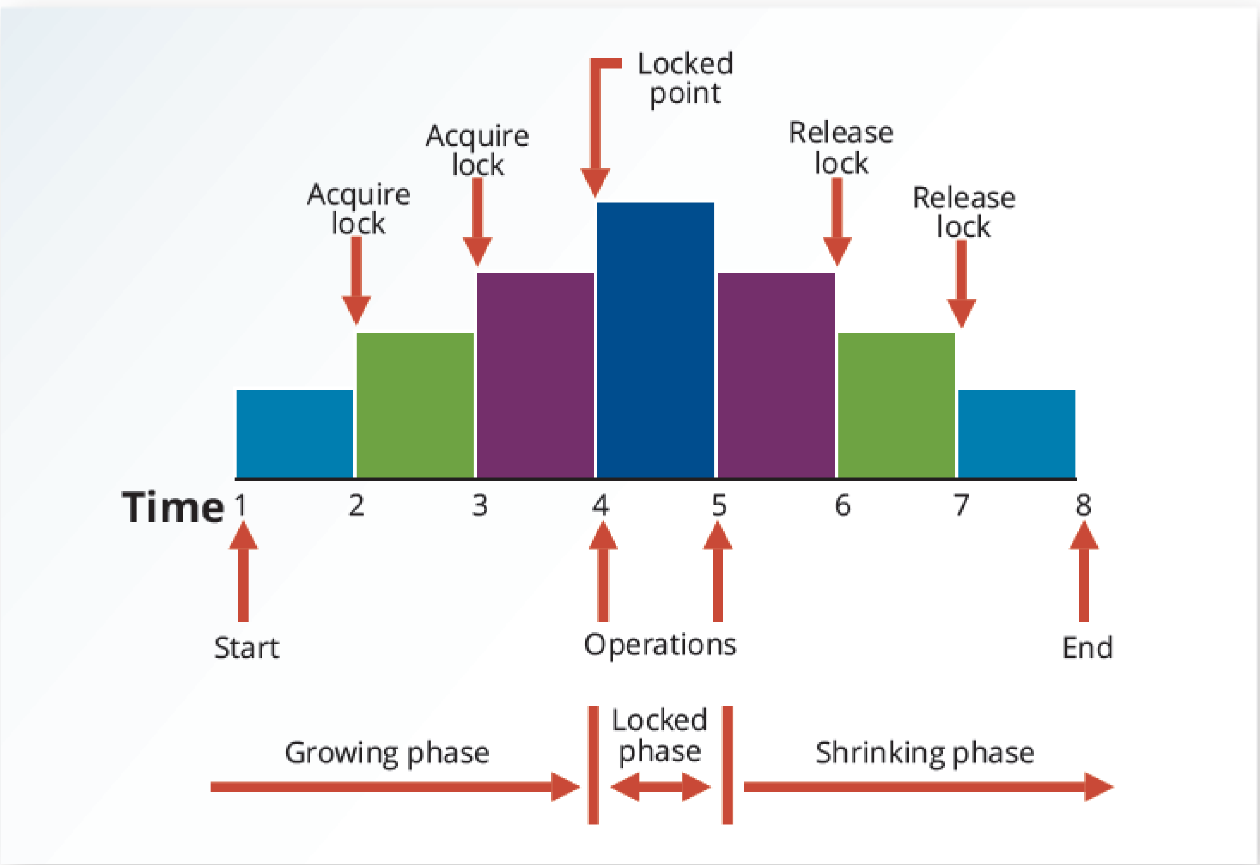


2-Phase Locking Protocol (2PL)

A transaction follows the two-phase locking protocol if all locking operations **precede the first unlock operation** in the transaction.

- There is no requirement that all locks be obtained simultaneously.
- Normally, the transaction acquires some locks, does some processing, and goes on to acquire additional locks as needed.
- However, it never releases any lock until it has reached a stage where no new locks are needed. *In Strict 2PL, all locks released on commit.*

Figure 10.7 Two-Phase Locking Protocol



Preventing the lost update problem

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	write_lock(bal_x)	100
t_3	write_lock(bal_x)	read(bal_x)	100
t_4	WAIT	$bal_x = bal_x + 100$	100
t_5	WAIT	write(bal_x)	200
t_6	WAIT	commit/unlock(bal_x)	200
t_7	read(bal_x)		200
t_8	$bal_x = bal_x - 10$		200
t_9	write(bal_x)		190
t_{10}	commit/unlock(bal_x)		190

Figure 22.15 Preventing the lost update problem.

Preventing the uncommitted dependency problem

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		write_lock(bal_x)	100
t_3		read(bal_x)	100
t_4	begin_transaction	$bal_x = bal_x + 100$	100
t_5	write_lock(bal_x)	write(bal_x)	200
t_6	WAIT	rollback/unlock(bal_x)	100
t_7	read(bal_x)		100
t_8	$bal_x = bal_x - 10$		100
t_9	write(bal_x)		90
t_{10}	commit/unlock(bal_x)		90

Figure 22.16 Preventing the uncommitted dependency problem.

Preventing the inconsistent analysis problem

To prevent this problem from occurring, T5 must precede its reads by exclusive locks, and T6 must precede its reads with shared locks.

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	write_lock(bal_x)		100	50	25	0
t_4	read(bal_x)	read_lock(bal_x)	100	50	25	0
t_5	$bal_x = bal_x - 10$	WAIT	100	50	25	0
t_6	write(bal_x)	WAIT	90	50	25	0
t_7	write_lock(bal_z)	WAIT	90	50	25	0
t_8	read(bal_z)	WAIT	90	50	25	0
t_9	$bal_z = bal_z + 10$	WAIT	90	50	25	0
t_{10}	write(bal_z)	WAIT	90	50	35	0
t_{11}	commit/unlock(bal_x, bal_z)	WAIT	90	50	35	0
t_{12}		read(bal_x)	90	50	35	0
t_{13}		sum = sum + bal_x	90	50	35	90
t_{14}		read_lock(bal_y)	90	50	35	90
t_{15}		read(bal_y)	90	50	35	90
t_{16}		sum = sum + bal_y	90	50	35	140
t_{17}		read_lock(bal_z)	90	50	35	140
t_{18}		read(bal_z)	90	50	35	140
t_{19}		sum = sum + bal_z	90	50	35	175
t_{20}		commit/unlock(bal_x, bal_y, bal_z)	90	50	35	175

Figure 22.17 Preventing the inconsistent analysis problem.

Deadlock

- **Deadlock** occurs when 2 or more transaction are in a simultaneous **wait state**.
- A deadlock occurs when two transactions wait indefinitely for each other to unlock data. For example, a deadlock occurs when two transactions, T1 and T2, exist in the following mode:
- T1 = access data items X and Y
- T2 = access data items Y and X

- Note that deadlocks are possible only when one of the transactions wants to obtain an exclusive lock on a data item; no deadlock condition can exist among shared lock.

Deadlock

Time	T_{17}	T_{18}
t_1	begin_transaction	
t_2	write_lock($\mathbf{bal_x}$)	begin_transaction
t_3	read($\mathbf{bal_x}$)	write_lock($\mathbf{bal_y}$)
t_4	$\mathbf{bal_x} = \mathbf{bal_x} - 10$	read($\mathbf{bal_y}$)
t_5	write($\mathbf{bal_x}$)	$\mathbf{bal_y} = \mathbf{bal_y} + 100$
t_6	write_lock($\mathbf{bal_y}$)	write($\mathbf{bal_y}$)
t_7	WAIT	write_lock($\mathbf{bal_x}$)
t_8	WAIT	WAIT
t_9	WAIT	WAIT
t_{10}	⋮	WAIT
t_{11}	⋮	⋮

Deadlock Resolution

The system must **detect** and **break** **deadlocks** by the following strategies:

- Choosing one transaction as a **victim** and **rolling it back**.
- **Timing out** the transaction and returning an error.
- **Automatically restarting** the transaction hoping not to get deadlock again.
- Return an **error code** back to the **victim** and leaving it up to program to handle situation.