Database Systems
(CS 355 / CE 373)

Dr. Umer Tariq
Assistant Professor,
Dhanani School of Science & Engineering,
Habib University

Acknowledgements

- Many slides have been borrowed from the official lecture slides accompanying the textbook:

  Database System Concepts, (2019), Seventh Edition,

  Avi Silberschatz, Henry F. Korth, S. Sudarshan

  McGraw-Hill, ISBN 9780078022159

  The original lecture slides are available at:

  https://www.db-book.com/

- Some of the slides have been borrowed from the lectures by Dr. Immanuel Trummer (Cornell University). Available at: (www.itrummer.org)

Outline: Week 12

- Interaction of Transaction Isolation and Atomicity Properties
  - Recoverable Schedules
  - Cascading Rollback
  - Cascadeless Schedules
- Transaction Isolation Levels
- Concurrency-Control Mechanisms
  - Locks

DBMS: The Concept of *Transactions*

DB

Accounts

Number	Balance	C-ID

Account(C-ID, Number, Balance)

Application Code.

TRANSFER

OnButton Clicked OK

(SELECT Balance Read(A)
FROM Accounts
WHERE Number = '.... 1245')
=> BalanceVar

BalanceVar = BalanceVar - 50

(Update Accounts Write(A)
Set Balance = BalanceVar
WHERE Number = '....1245';)

(SELECT Balance
FROM Accounts Read(B)
WHERE Number = '....5679')

(=> BalanceVar
Balance = BalanceVar + 50

(Update Accounts Write(B)
Set Balance = BalanceVar
WHERE Number = '....5679')

UI

Name | Transfer

Current Account_1245 10,000

Savings Account_5679 50,000

From 1245

To 5679

Amount 50

OK

DBMS: The Concept of *Transactions*

TRANSFER (Transfer.py)

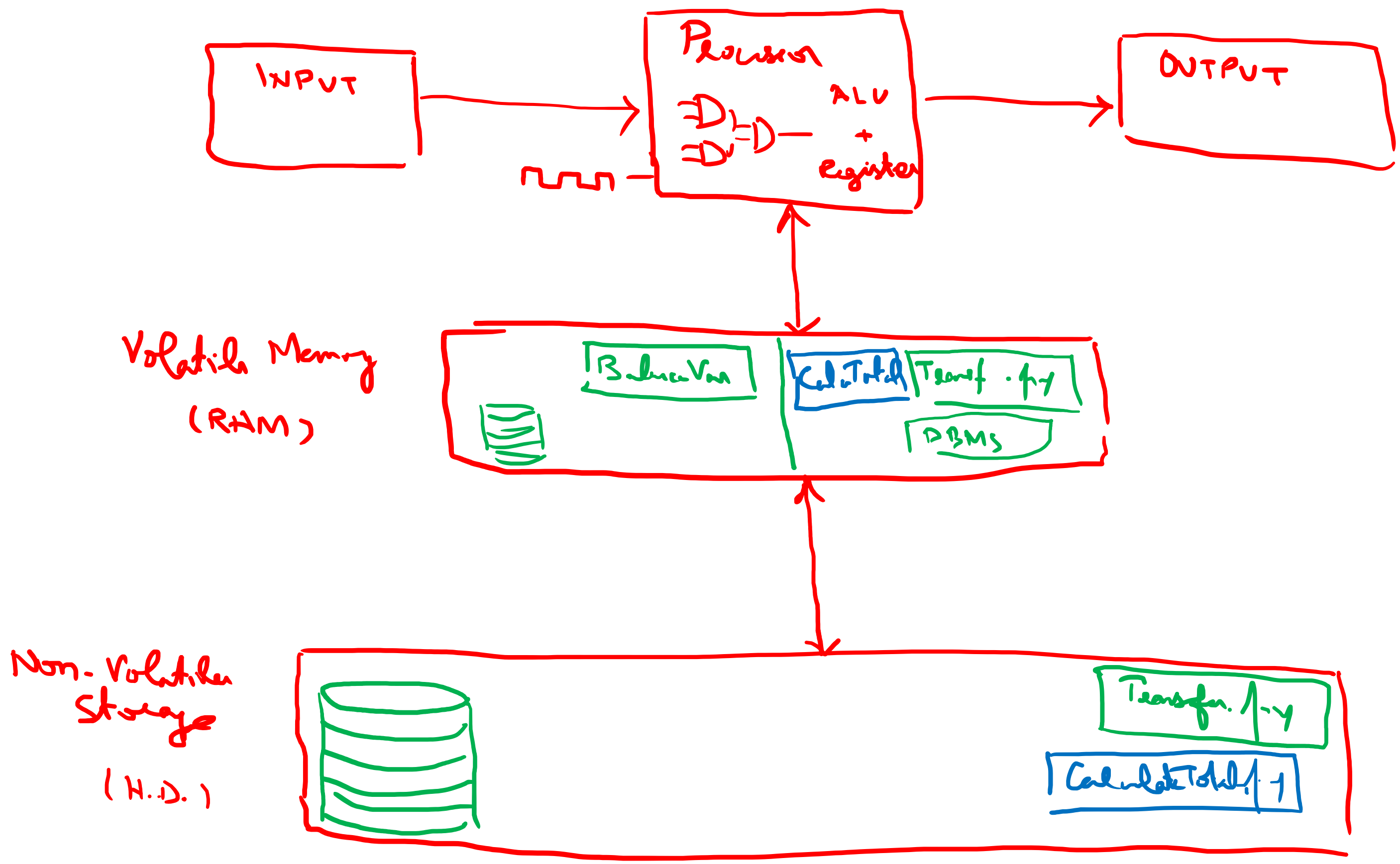
BEGIN TRANSACTION

1. Read (A)
2. A = A − 50
3. Write (A)
4. Read (B)
5. B = B + 50
6. Write (B)

END TRANSACTION

DBMS: The Concept of *Transactions*

Draw a "COMPUTER"?



DBMS: The Concept of *Transactions*

- Transaction
  - A transaction is unit of program execution that consists of multiple database operations but appears as a single, indivisible unit from the point of view of the database user/application.
  - A transaction executes in its entirety or not at all.

- Example
  - A transaction to transfer Rs. 50 from account A to account B

$$
\begin{aligned}
T_i:\ & \text{read}(A); \\
& A := A - 50; \\
& \text{write}(A); \\
& \text{read}(B); \\
& B := B + 50; \\
& \text{write}(B).
\end{aligned}
$$

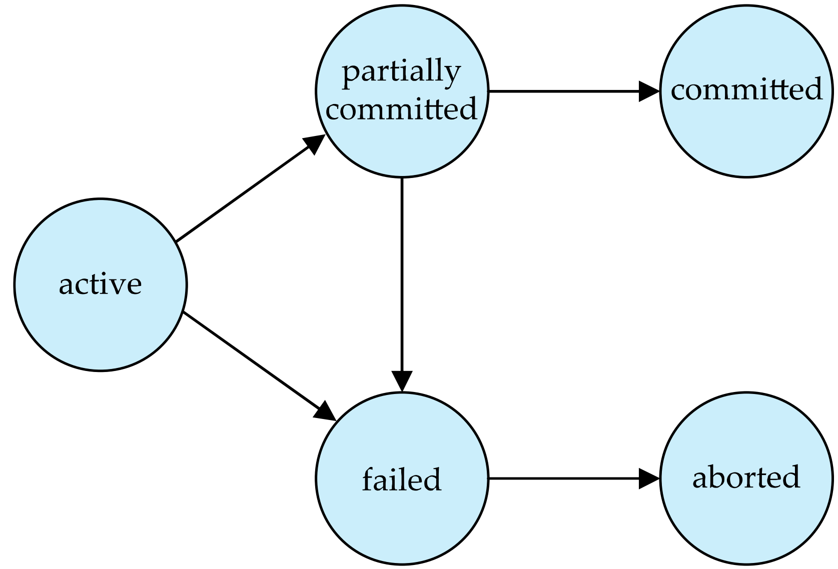
Transaction Properties: ACID

- A transaction must have the following four properties:
  - **A**tomicity,
  - **C**onsistency,
  - **I**solation,
  - **D**urability.

- These form the acronym **ACID** properties.

Transaction States

- Active
  - the initial state; the transaction stays in this state while it is executing.
- Partially committed
  - after the final statement has been executed.
- Failed
  - after the discovery that normal execution can no longer proceed.
- Aborted
  - after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
- Committed
  - after successful completion.



TRANSFER:-

1. **read**(A)
2. $A := A - 50$
3. **write**(A)
4. **read**(B)
5. $B := B + 50$
6. **write**(B)

Send.email( )

Serial vs Concurrent Execution of Transactions

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

T_1
read(A)
$A := A - 50$
write(A)
read(B)
$B := B + 50$
write(B)
commit

T_2
read(A)
$temp := A * 0.1$
$A := A - temp$
write(A)
read(B)
$B := B + temp$
write(B)
commit

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

- Restricting ourselves to executing transactions serially (i.e. one after the other) makes it easy to achieve isolation among transactions.
- However, concurrent execution of transactions provides significant performance benefits:
  - Increased throughput
  - Reduced average response times

Concurrent Schedule: Consistent vs Inconsistent State

T_1
read(A)
$A := A - 50$
write(A)
read(B)
$B := B + 50$
write(B)
commit

T_2
read(A)
$temp := A * 0.1$
$A := A - temp$
write(A)
read(B)
$B := B + temp$
write(B)
commit

A = 100, B = 100

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

A = 45 B = 155

var A_1 = 100
var A_1 = 50
DB-A = 50
var A = 50
temp = 5
var A_2 = 45
DB-A = 45!
var B_1 = 100
var B_1 = 150
DB_B = 150
var B_2 = 150
var B_2 = 155
DB_B = 155

var A_1 = 100
var A_1 = 50
var A_2 = 100
temp = 10
var A_2 = 90
DB_A = 90!
var B_2 = 100
DB-A = 50!
var B_1 = 100
var B_1 = 150
DB-B = 150
var B_2 = 110
DB-B = 110!

A = 100, B = 100

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

A = 50 B = 110

Serializable Schedule

- A serial schedule is always consistent, i.e. it maintains the consistent state of the database.
- However, the same cannot be guaranteed for a concurrent schedule.
- If a concurrent schedule can be shown to have the same effect as a serial schedule, (in other words it is shown to be equivalent to a serial schedule), then it can ensure the consistency of the database.
- Such a concurrent schedule is called a **<u>serializable schedule</u>**.

T_1
read(A)
$A := A - 50$
write(A)
read(B)
$B := B + 50$
write(B)
commit

T_2
read(A)
$temp := A * 0.1$
$A := A - temp$
write(A)
read(B)
$B := B + temp$
write(B)
commit

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

Serializable Schedules: Equivalence

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

Figure 17.4 Schedule 3—a concurrent schedule equivalent to schedule 1.

E Q U I V A L E N T T O

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

Figure 17.2 Schedule 1—a serial schedule in which T_1 is followed by T_2.

<u>If a concurrent schedule can be shown to be equivalent to a serial schedule, we conclude that this concurrent schedule maintains the consistency of the database.</u>

Conflict-Equivalent Schedules

- If a schedule S can be transformed into a schedule S′ by a series of swaps of nonconflicting instructions, we say that S and S′ are **conflict equivalent.**

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

EQUIVALENT TO

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

Transaction Isolation and Atomicity

- So far, we have studied concurrent schedules while assuming implicitly that there are no transaction failures
- We now address the effect of transaction failures during concurrent execution of transactions.

Nonrecoverable Schedules

$T_6 \rightarrow T_7$

- Consider the partial schedule shown:
  - We call this a partial schedule because we have not included a commit or abort operation for T6.

- Notice the following:
  - T_7 has read the value of A written by T_6 . Therefore, we say that T_7 is <u>*dependent*</u> on T_6
  - T_7 commits immediately after executing the *read*(*A*) instruction.
  - T_7 commits while T_6 is still in the active state.

T_6	T_7
read(A)	
write(A)	
	$\boxed{\text{read}(A)}$
	$\boxed{\text{commit}}$
× read(B)	
...	
commit	

- Suppose that T_6 fails after the *read(B)* statement.
  - We must abort T_7 to ensure atomicity of T_6 (as any effect of value of A written by T_6 must be rolled back.)
  - However, T_7 has already been committed and cannot be aborted.
  - Thus, we have a situation where it is impossible to recover from the failure of T_6
  - This is an example of <u>***nonrecoverable schedule***</u>.

Recoverable Schedules

- A recoverable schedule is one where
  - For each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i , the commit operation of T_i appears before the commit operation of T_j

- For the example shown here to be recoverable, T_7 would have to delay committing until after T_6 commits.

T_6	T_7
read(A)	
write(A)	
	read(A)
	~~commit~~
read(B)	
commit	
	commit

Cascading Rollback

- Even if a schedule is recoverable, to recover correctly from the failure of a transaction T_i , we may have to roll back several transactions.

- Consider the partial schedule shown here:
  - Transaction T_8 writes a value of A that is read by transaction T_9. Transaction T_9 writes a value of A that is read by transaction T_{10}.
  - Suppose that, at this point (indicated as *abort*) T_8 fails.
  - T_8 must be rolled back. Since T_9 is dependent on T_8, T_9 must be rolled back. Since T_{10} is dependent on T_9, T_{10} must be rolled back.
  - This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called **cascading rollback**

T_8	T_9	T_{10}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
abort		
Commit		
	commit	
		commit

Concurrent Schedule:

Serializable ? Recoverable ? Cascadeless ?

Cascadeless Schedules

- Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work.
- It is desirable to restrict the schedules to those where cascading rollbacks cannot occur.
-
- Such schedules are called **<u>cascadeless schedules</u>**.
- Formally,
  - a **cascadeless schedule** is one where, for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j .
- Note that a cascadeless schedule is also a recoverable schedule.

T_8	T_9	T_{10}
read(A)		
read(B)		
write(A)		
	~~read(A)~~	
	~~write(A)~~	
		~~read(A)~~

commit		
	read(A)	
	write(A)	
	commit	
		read(A)

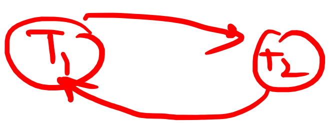
Transaction Isolation Levels: Real-World Applicability

- Online Store Example (with two customers trying to buy the same item).

A, B, C.

ItemAvailable-A = 15

T_1	T_2
	Read(ItemAvailable-A)
Read(ItemAvailable-A)	
	UI
UI + Payment	+ Payment
ItemAvailable-A--;	
Write(ItemAvailable-A)	
	ItemAvailable-A--;
	Write(ItemAvailable-A)



Transaction Isolation Levels: Background

- Transaction isolation (and serializability) is a useful concept because it allows programmers to ignore issues related to concurrency when they code transactions

- However, the protocols used to ensure serializability of concurrent transaction executions may allow too little concurrency for certain applications.

- In these cases, weaker levels of isolation/consistency are used.
  - The use of weaker levels of isolation places additional burdens on programmers for ensuring database correctness.
  - An application designer may decide to accept a weaker isolation level in order to improve system performance.

Transaction Isolation Levels: SQL Standard

The SQL standard allows a transaction to specify its isolation level:

- **Serializable**
  - usually ensures serializable execution.
- **Repeatable read**
  - allows only committed data to be read and further requires that, between two reads of a data item by a transaction, no other transaction is allowed to update it.
- **Read committed**
  - allows only committed data to be read, but does not require repeatable reads.
- **Read uncommitted**
  - allows uncommitted data to be read. It is the lowest isolation level allowed by SQL.
- All the isolation levels above additionally disallow "dirty writes": They disallow writes to a data item that has already been written by another transaction that has not yet committed or aborted.

Transaction Isolation Levels: SQL Standard

- Many database systems run, by default, at the read committed isolation level.

- In SQL, it is possible to set the isolation level explicitly. For example, through the statement:

set transaction isolation level serializable

Review: Concurrent Execution of Transactions: Role of Concurrency-Control Schemes

- Concurrency-control schemes
  - Mechanisms to achieve isolation among concurrently-executing transactions
  - Mechanisms to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
- Will study these schemes after studying the notion of 'correctness of concurrent executions'

T_1
read(A)
$A := A - 50$
write(A)
read(B)
$B := B + 50$
write(B)
commit

T_2
read(A)
$temp := A * 0.1$
$A := A - temp$
write(A)
read(B)
$B := B + temp$
write(B)
commit

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

Transaction Isolation and Concurrency Control Schemes

- Concurrency-control protocols of DBMS allow the concurrent execution of transaction in such a manner that the resulting transaction schedules are
  - Serializable
  - Recoverable
  - Cascadeless

- Concurrency control protocols (generally) do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids non-serializable schedules.

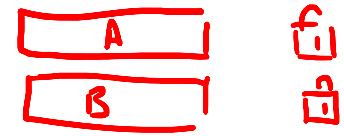
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.

- Tests for serializability help us understand why a concurrency-control protocol of DBMS is correct.

Concurrency Control Schemes: Overview

- There are various concurrency-control schemes that a DBMS can use to ensure that even when multiple transactions are executed concurrently, only acceptable transaction schedules are generated.

- Some of the main concurrency control schemes can be categorized based on their use of the following concepts
  - Locks
  - Timestamps
  - Multiple Versions of Data and Snapshot Isolation



Concurrency-Control Schemes: Locks

- One way to ensure isolation is to require that data items be accessed in a mutually exclusive manner
  - While one transaction is accessing a data item, no other transaction can modify that data item.
- The most common method used to implement this requirement is to allow a transaction to access a data item only if it is holding a <u>**lock**</u> on that item.
  - A lock is a mechanism which provides access privileges to a transaction on a shared resource,
  - While the lock is acquired, no other transaction can access that resource.
  - In order to release the resource, i.e. to make it available to other transactions, a transaction has to <u>unlock</u> the resource.

T_1: lock-X(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(A).

T_2: lock-S(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display($A + B$).

Concurrency Control Schemes: Locks

T_1	T_2	concurrency-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$	~~lock(B)~~	× ×
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

T_3

lock(B)

waiting

lock(B)

Concurrency Control Schemes: Lock Modes

- There could be various possible modes in which the data item is locked. For example:
  - **Shared**. If a transaction T_i has obtained a shared-mode lock (***S***) on item Q, then T_i can read, but cannot write Q.
  - **Exclusive**. If a transaction T_i has obtained an exclusive-mode lock (***X***) on item Q, then T_i can both read and write Q.
- Every transaction must first request the **concurrency-control manager** for a lock in an appropriate mode on data item Q depending on the type of operations that it will perform on Q.
- The transaction can proceed with the operation only after the concurrency-control manager grants the lock to the transaction.
- The use of these two lock modes allow multiple transactions to read a data item but limits write access to just one instruction at a time.

Concurrency Control Schemes: Compatibility of Lock Modes

- Given a set of lock modes, we can define a **compatibility function** on them as follows:
  - Let *A* and *B* represent arbitrary lock modes.
  - Suppose that a transaction T_i requests a lock of mode *A* on item *Q* on which transaction T_j ($T_i \neq T_j$) currently holds a lock of mode *B*.
  - If transaction T_i can be granted a lock on *Q* immediately, in spite of the presence of the mode *B* lock, then we say mode *A* is **compatible** with mode *B*.

- Such a compatibility function between a set of lock modes can be represented conveniently by a matrix (Lock-compatibility matrix)
  - An element ***comp***(*A*, *B*) of the matrix has the value true if and only if mode *A* is compatible with mode *B*.
  - Compatibility matrix of shared-mode (S) and exclusive-mode (X) locks:

	S	X
S	true	false
X	false	false

Analysis of Transaction Schedules with Locks

- Despite the acquisition of locks the following schedule results in an inconsistent state.

T_1: lock-X(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(A).

T_2: lock-S(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display($A + B$).

A =100 B = 100

T_1	T_2	concurrency-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B) $B := B - 50$ write(B) unlock(B)	B = 50	
	lock-S(A)	
		grant-S(A, T_2)
	read(A) unlock(A) lock-S(B)	
		grant-S(B, T_2)
	read(B) unlock(B) display($A + B$)	
A + B = 150		
lock-X(A)		
		grant-X(A, T_1)
read(A) $A := A + 50$ write(A) unlock(A)		

Analysis of Transaction Schedules with Locks

- With unlocking delayed, transactions T_3 and T_4 lead to a consistent schedule
  - A "**Locking Protocol**" dictates when a transaction may lock and unlock each of the data item. The choice of a locking protocol restricts the number of possible transaction schedules.

T_1: lock-X(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(A).

T_2: lock-S(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display($A + B$).

T_3: lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).

T_4: lock-S(A);
read(A);
lock-S(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).

The Two-Phase Locking Protocol

- One protocol that ensures serializability is the two-phase locking protocol.
- This protocol requires that each transaction issue lock and unlock requests in two phases:
  1. Growing phase. A transaction may obtain locks, but may not release any lock.
  2. Shrinking phase. A transaction may release locks, but may not obtain any new locks.

- Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests

T_1: lock-X(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(A).

T_2: lock-S(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display($A + B$).

T_3: lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).

T_4: lock-S(A);
read(A);
lock-S(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).

The Two-Phase Locking Protocol: Example

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

The Two-Phase Locking Protocol: Exercise

18.2 Consider the following two transactions:

T_{34}: read(A);
read(B);
if $A = 0$ **then** $B := B + 1$;
write(B).

T_{35}: read(B);
read(A);
if $B = 0$ **then** $A := A + 1$;
write(A).

Add lock and unlock instructions to transactions T_{31} and T_{32} so that they observe the two-phase locking protocol.

T_{34}:

lock_S(A)
read(A)
lock_X(B)
read(B)
if A = 0 then B := B+1
write(B)
unlock(A)
unlock(B)

T_{35}:

lock_S(B)
read(B)
lock_X(A)
read(A)
if B = 0 then A := A+1
write(A)
unlock(A)
unlock(B)

The Two-Phase Locking Protocol: Exercise

T1(Without locks)	T2 (Without Locks)
Read(A)	Read(A)
Read(B)	Read(S)
Read(C)	Read(T)
A=B+C	A=S+T
Write(A)	Write(A)

The Two-Phase Locking Protocol: Exercise

T1(Without locks)	T2 (Without Locks)
Read(A)	Read(A)
Read(B)	Read(S)
Read(C)	Read(T)
A=B+C	A=S+T
Write(A)	Write(A)

T1(With Locks)	T2(With Locks)
Lock-X(A)	Lock-X(A)
Read(A)	Read(A)
Lock-S(B)	Lock-S(S)
Read(B)	Read(S)
Lock-S(C)	Lock-S(T)
Read(C)	Read(T)
A=B+C	A=S+T
Write(A)	Write(A)
Unlock-X(A)	Unlock-X(A)
Unlock-S(B)	Unlock-S(S)
Unlock-S(C)	Unlock-S(T)