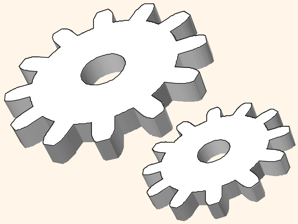
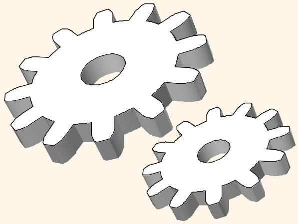
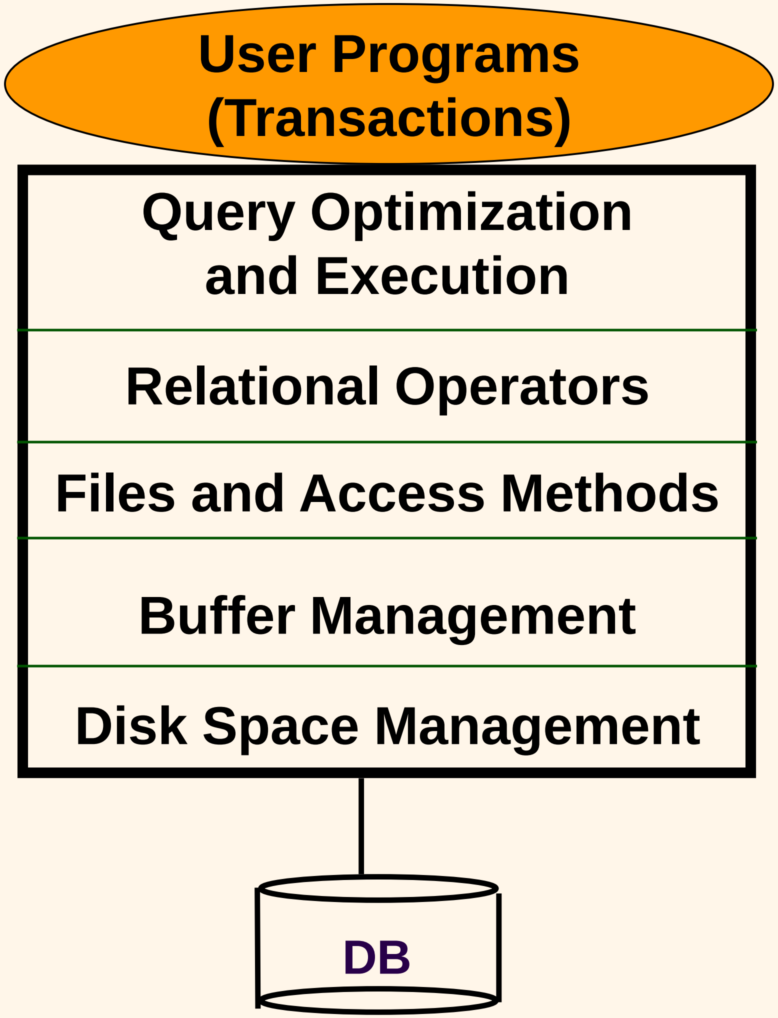


COMP7640
Database Systems & Administration

Transactions &
Concurrency Control

Where Are We Now?





User Programs (Transactions)

Query Optimization and Execution

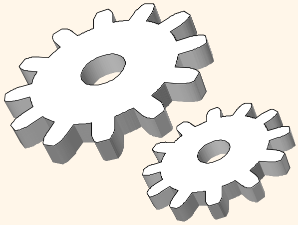
Relational Operators

Files and Access Methods

Buffer Management

Disk Space Management

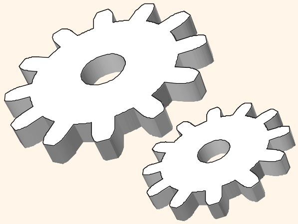
DB



Transactions

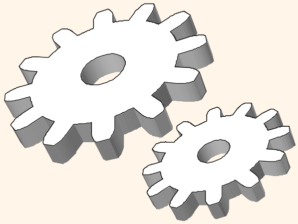
- A **transaction** is a unit of execution
- A transaction requires *accessing/manipulating* data items in database systems, e.g.,
  - Read the balance in a bank account
  - Update the balance in a bank account
- A database system receives transactions from *user programs* written by high level program languages, e.g., SQL, C++, Java

Transactions



- A transaction comprises action(s)
  - Read operation
    - **read**(X): transfers the data item *from the database* to the variable X.
  - Write operation
    - **write**(X): transfers the variable X *to the database*.
  - Commit operation
    - **commit**: completes the transaction.
  - Abort operation
    - **abort**: terminates and undo actions done
  - Arithmetic operations (+, -, ×, and ÷)

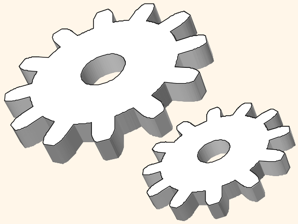
A Transaction Example



- Transfer \$50 from account A to account B:

1. **read**(A)	from database disk to the main memory
2. $A = A - 50$	done in the main memory
3. **write**(A)	from the main memory to the database
4. **read**(B)	from database disk to the main memory
5. $B = B + 50$	done in the main memory
6. **write**(B)	from the main memory to the database
7. **commit**	ask the database to complete this transaction

Transactions: Challenge I



- Database systems need to handle *many* transactions
    - Concurrent execution of multiple transactions
- A high *throughput* (*transactions per second*) is required

Example 1 [a]:

The total number of credit card transactions was 207.65 million for Q3/2020, representing a 1.9% increase from the previous quarter and a 3.1% decrease from the same period in 2019. The total value of credit card transactions was HK$147.8 billion for Q3/2020, representing a 2.9% increase from the previous quarter and a 21.7% decrease from the same period in 2019. Of the total transaction value, HK$120.9 billion (81.8%) was related to retail spending in Hong Kong, HK$17.3 billion (11.7%) in retail spending overseas and HK$9.6 billion (6.5%) in cash advances.

Example 2 [b]:

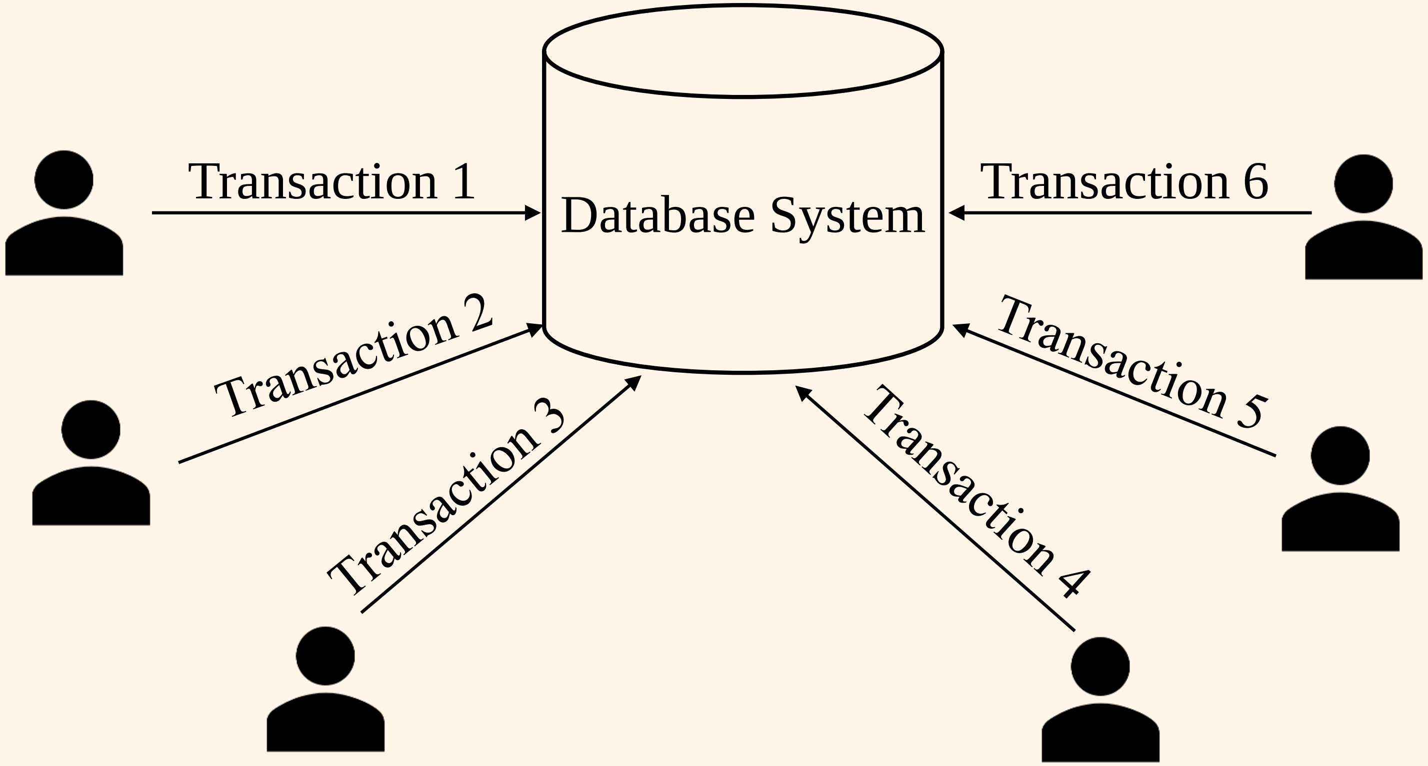
On Double 11 in 2019, Alibaba Cloud's cloud-native database product ApsaraDB for POLARDB set an all-new record, 87 million transactions per second (TPS). That's 87 million transactions that were completed in just the blink of an eye.

[a] https://www.hkma.gov.hk/eng/news-and-media/press-releases/2020/12/20201218-4/

[b] https://www.alibabacloud.com/blog/learn-how-alibaba-clouds-databases-could-support-87-million-transactions-per-second_595593?spm=a3c0i.14509290.7797739730.1.1641757eIwf0Mb

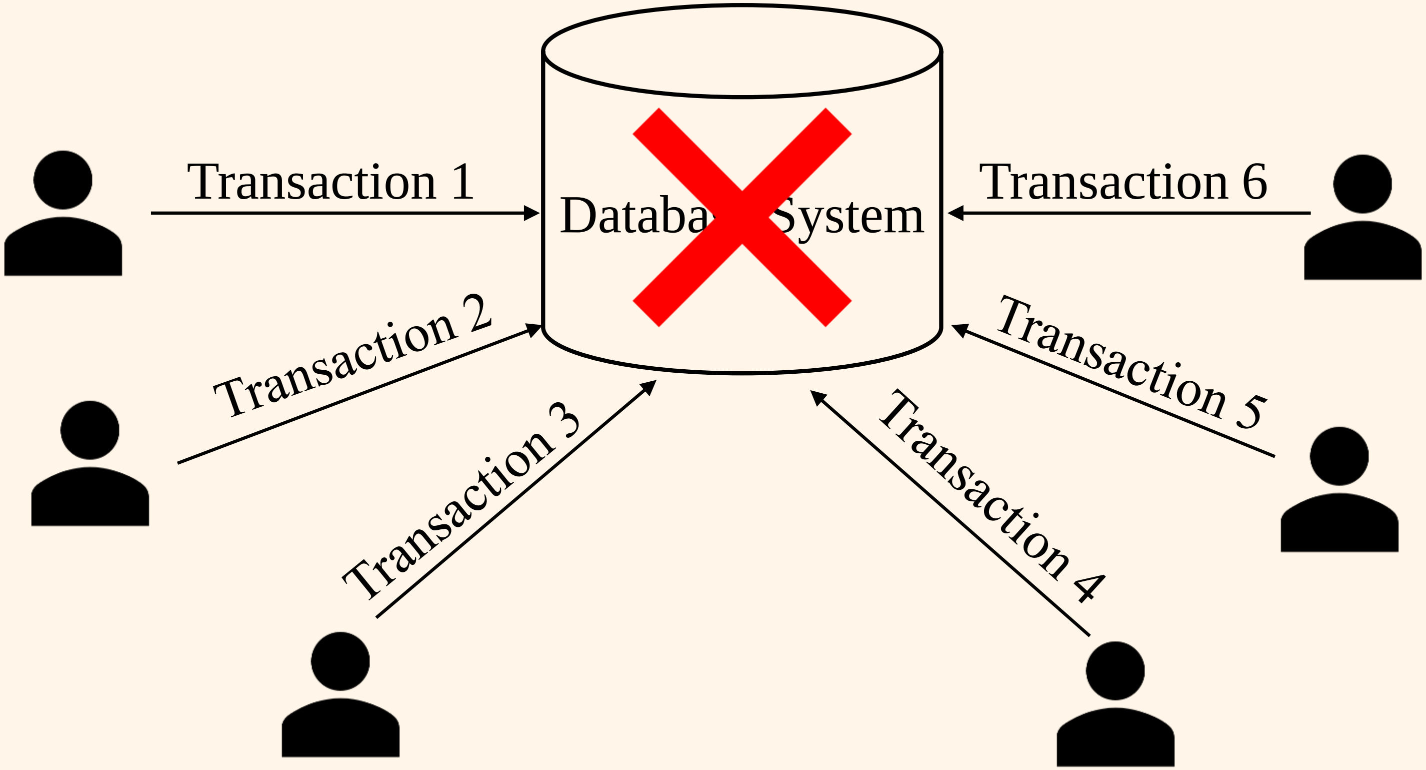
Transactions: Challenge II

- Concurrent execution of multiple transactions may produce *incorrect* states/results in database systems
  - e.g., multiple transactions modify the same data items
- Careful *scheduling* is required



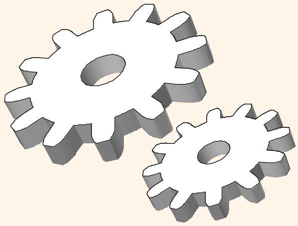
Transactions: Challenge III

- Database systems can crash suddenly due to
  - hardware failures, system crashes, electricity outage
- *Recovery* is required



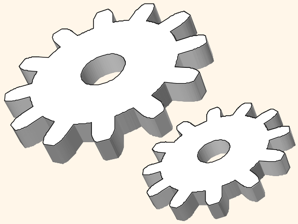
Transactions: ACID Properties

- Each transaction should achieve *four properties* when handling multiple transactions in database system
  - **Atomicity**
  - **Consistency**
  - **Isolation**
  - **Durability**



Transactions: ACID Properties

- **Atomicity:** Either committed or aborted
    - *All* operations in each transaction *must be executed* or *none of them are executed*
- **Consistency:** No constraint violation after each transaction finishes.
    - Depends on applications (e.g., We do not allow the bank account with balance smaller than zero.)

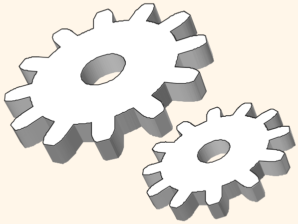


Transactions: ACID Properties

- **Isolation:** Concurrent transactions are *not* aware of each other. Each thinks that it is the *only* running transaction.

- **Durability:** If a transaction is committed, its changes to the database are *permanent*, even if there is a system failure.

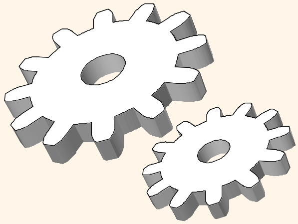
Example of Fund Transfer



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

- Atomicity: If any step fails, roll back.
- Consistency: Suppose that we have this constraint "all balances must not be negative".
  - If the constraint is violated after the transaction finishes, the transaction must be rolled-back, that is, the database state is restored to that before the transaction.

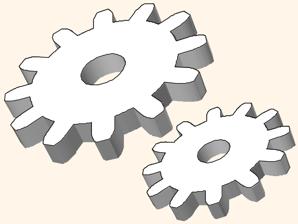
Example of Fund Transfer



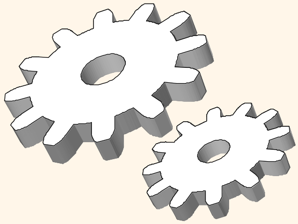
1. **read**(*A*)
2. *A* = *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* = *B* + 50
6. **write**(*B*)

- Isolation: Assume that another transaction also needs to access *A* and *B*. The two transactions should *not* affect each other.
- Durability: Once the transaction is committed (or complete), the money transfer is *permanent*.

Implementing ACID



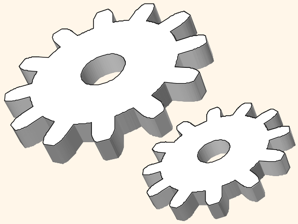
- **Atomicity**
  - No incomplete transaction.
  - Can guide us handle Challenge III
- **Consistency**
  - User's responsibility: through integrity constraints and triggers.
- **Isolation**
  - **Concurrency control**
  - Can guide us handle Challenge II
- **Durability**
  - Requires the **recovery** mechanism.
  - Can guide us handle Challenge III



Concurrency Control

- Concurrent execution of transactions
  - While a transaction is waiting for a page to be read in from disk, CPU can process another transaction (Fast CPU, relatively slow I/O)
  - Increases throughput (i.e., reduces response time)
  - Requires *careful scheduling to* fulfill the *isolation property*
- *Concurrency control* is to generate an execution *schedule* of transactions without *incorrect* states/results

Schedule



- **Schedule** – the order that the operations of *multiple* transactions are executed

Transaction T_1	*Transaction* T_2
1: **read**(A)	1: **read**(X)
2: **read**(B)	2: **read**(Y)
3: $A := A + B$	3: $Y := X + Y$
4: **write**(A)	4: **write**(Y)

⇒

A Schedule

T_1: **read**(A)

T_2: **read**(X)

T_1: **read**(B)

T_1: $A := A + B$

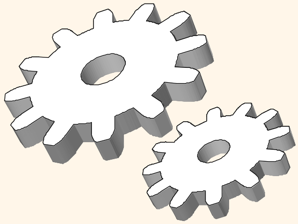
T_2: **read**(Y)

T_1: **write**(A)

T_2: $Y := X + Y$

T_2: **write**(Y)

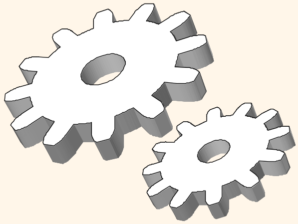
Schedule



- **Schedule** – the order that the operations of multiple transactions are executed
- Two *Basic* Requirements
  - A schedule should consist of *all actions* of all transactions
  - A schedule should preserve the *relative order of any two actions* in the *same* transaction

T1:	R(A), W(A),		R(C), W(C), Commit	
T2:		R(B), W(B),		Abort

Example

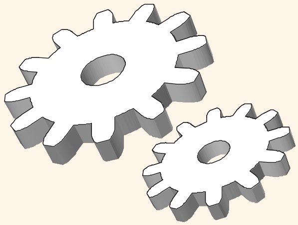


- *Transaction* T_1
  1. **read**(A)
  2. $A := A + 50$
  3. **write**(A)

- *Transaction* T_2
  1. **read**(A)
  2. $A := A + 50$
  3. **write**(A)

- "A" should be added 100 afterwards. Initially, A=100. Then, A should be 200 afterwards.
- See the next slide for different wrong schedules.

Example



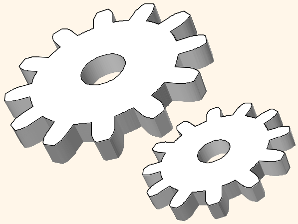
- Wrong schedules

T_1: **read**(A)
T_1: $A := A + 50$
T_1: **write**(A)
T_2: **read**(A)
T_2: **write**(A)

Miss one operation
T_2: $A := A + 50$

T_1: **read**(A)
T_1: **write**(A)
T_1: $A := A + 50$
T_2: **read**(A)
T_2: $A := A + 50$
T_2: **write**(A)

Does not preserve the order of
T_1: $A := A + 50$ and T_1: **write**(A)



Example

- Wrong schedules (Initially A=100)

T_1: **read**(A)

T_2: **read**(A)

T_1: $A := A + 50$

T_2: $A := A + 50$

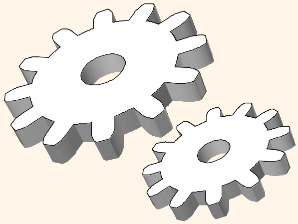
T_1: **write**(A)

T_2: **write**(A)

Fulfills all basic requirements:

- consist of *all* actions ✓
- preserve the *relative order* ✓

But the result is *incorrect*. (Why?)



Example

- Why Wrong schedules (Initially A=100)

T_1: **read**(A)

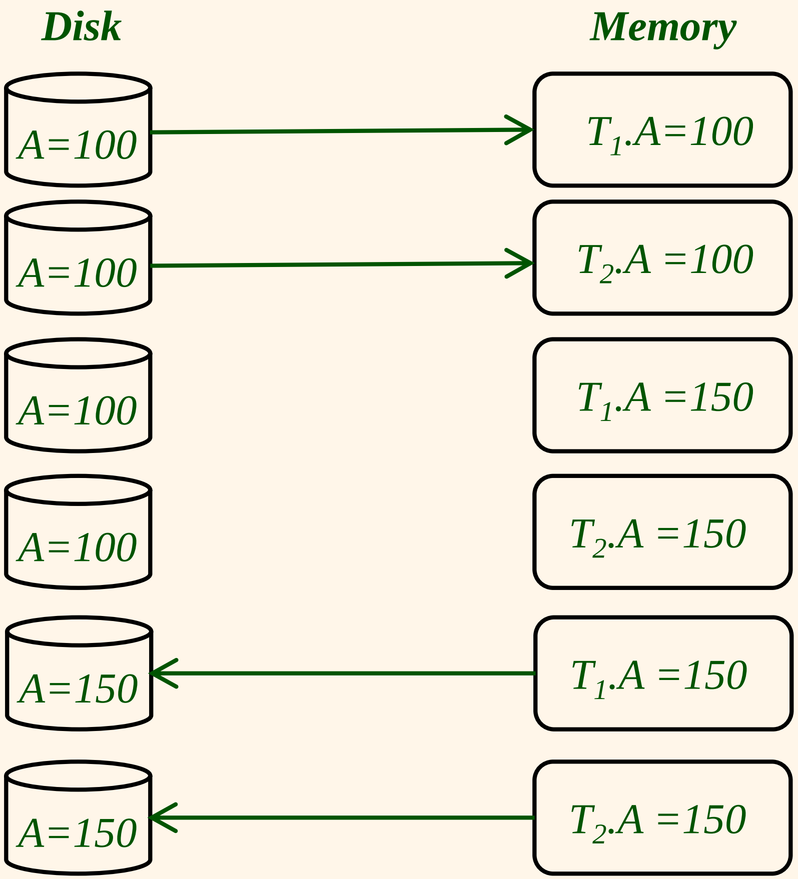
T_2: **read**(A)

T_1: $A := A + 50$

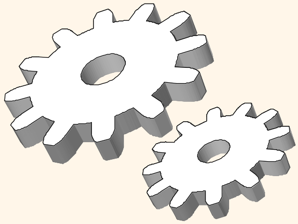
T_2: $A := A + 50$

T_1: **write**(A)

T_2: **write**(A)

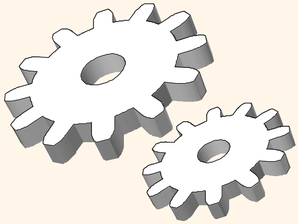


Schedule



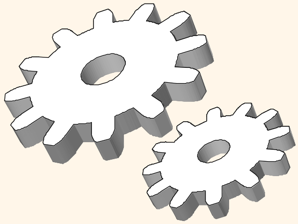
- **Schedule** – the order that the operations of multiple transactions are executed
- Requirements
  - A schedule should consist of *all* actions of all transactions
  - A schedule should preserve the *relative order* of any two actions in the same transaction
  - The result of the schedule should be correct
    - Ensure consistency when *accessing* the *same* data object
    - Ensure consistency when *updating* the *same* data object
    - **Solution**: Add *locks* on data objects

Lock



- When a transaction wants to read/write a data object, it must first acquire a *lock* on it.
  - Transactions can proceed *only* after the *necessary* locks are obtained.
- Must request Exclusive (X) lock on a data object *before writing* to the object.
- Must request Shared (S) lock/ Exclusive (X) lock on an object *before reading* the object.

Example of Lock



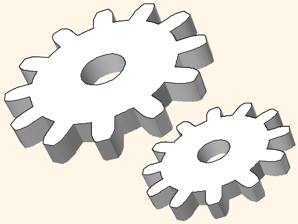
Transaction

R(*A*)
R(*B*)
B = B + A
W(B)

→

S(*A*) //obtain a shared lock
R(*A*) // read A
U(*A*) //unlock A
X(*B*) //obtain an exclusive lock
R(*B*) // read B
B = B + A
W(B) // write B
U(*B*) // unlock B

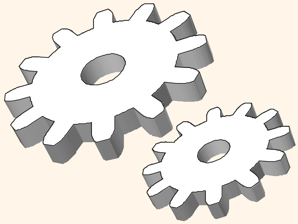
Lock Compatibility



	S	**X**
S	**True**	**False**
X	**False**	**False**

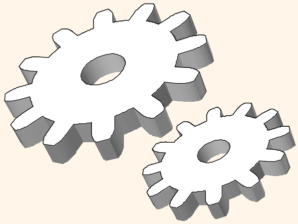
- A transaction obtains a lock only if the lock is compatible with those already on the object.
  - S lock can be granted only if *no* X lock has been granted on the object.
  - X lock can be granted only if *no* lock has been granted on the object.
- If a lock cannot be granted, the transaction that requests this lock must wait until other transactions release locks.

Lock Example



T_1	T_2
X(A);Read(A)	
A ←A+100;Write(A)	
X(B); U(A)	
	X(A);Read(A)
	A ←Ax2;Write(A); X(B) ***Wait***
Read(B);B ←B+100	
Write(B); U(B)	
	X(B);U(A);Read(B)
	B ←Bx2;Write(B);U(B);

Example of Scheduling with Locks



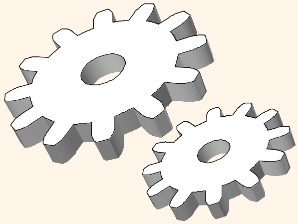
T_1	T_2
R(A)	**R**(A)
R(B)	**R**(B)
B = B + A	
W(B)	

T_1	T_2
S(A)	
X(B)	
	S(A)
	S(B)
	R(A)
R(A)	
U(A)	
R(B)	
W(B)	
U(B)	
	S(B)
	R(B)
	U(A)
	U(B)

Cannot grant this lock

Arithmetic operations can be omitted

Example of Scheduling with Locks



Initially A=10 and B=20

T_1	T_2
R(A)	**R**(A)
R(B)	**R**(B)
A = B + 10	B = A + 20
W(A)	**W**(B)

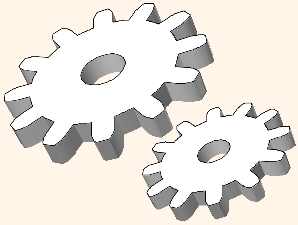
2 Possible **Correct** Results:

T_1 before T_2: A=30, B=50
T_2 before T_1: B=30, A=40

A=10 **B=20**

T_1	T_2
	S(A)
	R(A)
	U(A)
X(A)	
S(B)	
R(A)	
R(B)	
U(B)	
	X(B)
	R(B)
	W(B)
	U(B)
W(A)	
U(A)	

Scheduling with Locks May Still have Wrong Results

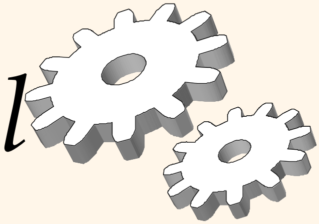


T_1	T_2
R(A)	**R**(A)
R(B)	**R**(B)
A = B + 10	B = A + 20
W(A)	**W**(B)

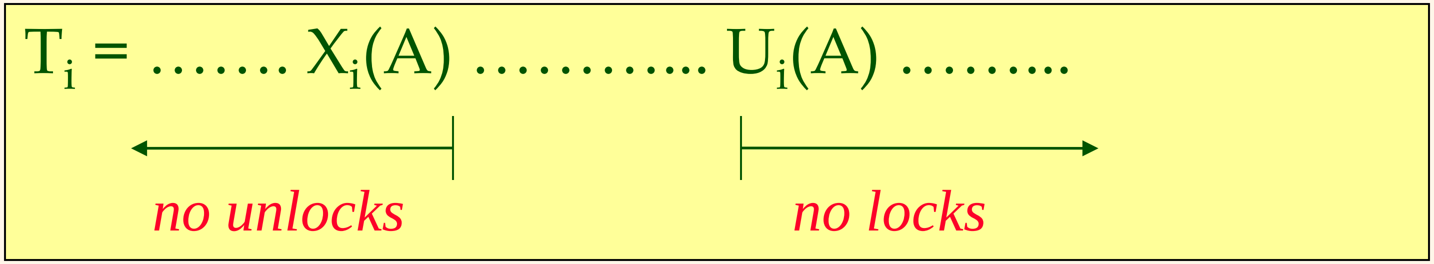
A=10 **B=20**

	T_1	T_2	
		S(A)	
		R(A)	$T_2.A=A=10$
		U(A)	
	X(A)		
	S(B)		
			$T_2.B = T_2.A + 20=30$
$T_1.A=A=10$	**R**(A)		
$T_1.B=B=20$	**R**(B)		
	U(B)		
$T_1.A = T_1.B + 10=30$			
		X(B)	
		R(B)	$T_2.B=B=20$
		W(B)	$B=T_2.B=30$
		U(B)	
$A=T_1.A=30$	**W**(A)		
	U(A)		

Two-Phase Locking (2PL) Protocol



- **Solution**: Use *Two-Phase Locking Protocol*
- A transaction cannot request additional locks once it releases any locks

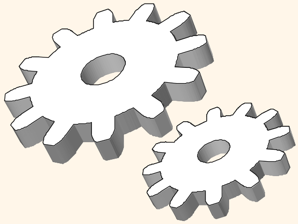


T_i = ……. $X_i(A)$ ……….... $U_i(A)$ ……....

no unlocks

no locks

2PL - Example



T_1	T_2
X(A)	
R(A)	
W(A)	
U(A)	
	X(A)
	R(A)
	W(A)
	U(A)

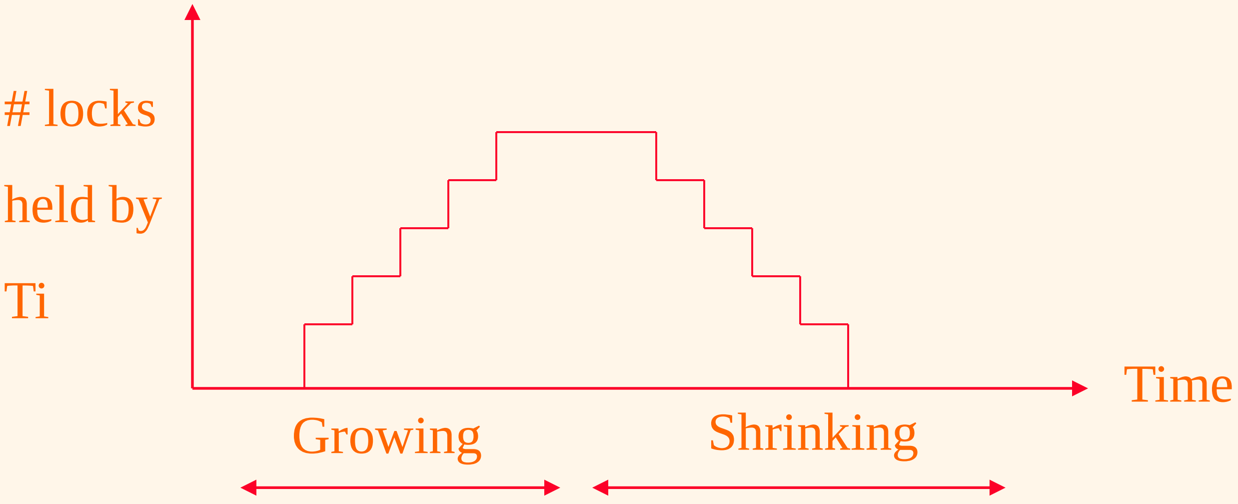
X lock granted, can go ahead

T_1
S(A)
R(A)
U(A)
X(A)
W(A)
U(A)

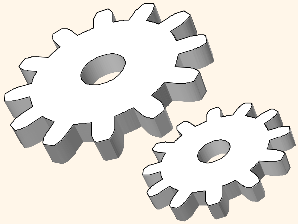
NOT 2PL

Properties of 2PL

- Every transaction has *two* phases
  - A **growing** phase -- acquires locks
    - Transaction can request locks
  - A **shrinking** phase -- releases locks
    - Transaction can only release locks but not request additional locks.

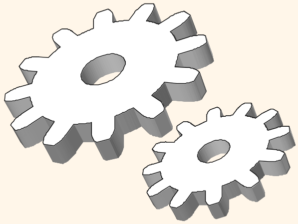


2PL Example



T_1	T_2
R(A)	**R**(A)
R(B)	**R**(B)
A = B + 10	B = A + 20
W(A)	**W**(B)

	T_1	T_2	
Growing	X(A)		
	S(B)		
	R(A)		
	R(B)		
	W(A)		
Shrinking	U(A)		
		S(A)	Growing
		R(A)	
	U(B)		
		X(B)	
		R(B)	
		W(B)	
		U(B)	Shrinking
		U(A)	

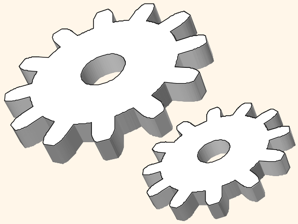


Question 1

- Determine if the below schedule follows the 2PL protocol.

T_1	T_2
R(A)	**R**(A)
R(B)	**R**(B)
B = B + A	
W(B)	

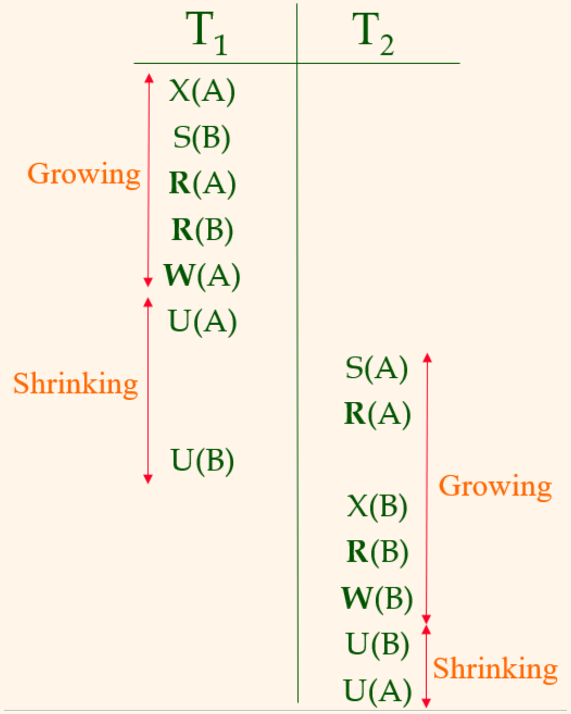
T_1	T_2
S(A)	
X(B)	
	S(A)
	R(A)
R(A)	
U(A)	
R(B)	
W(B)	
U(B)	
	S(B)
	R(B)
	U(A)
	U(B)



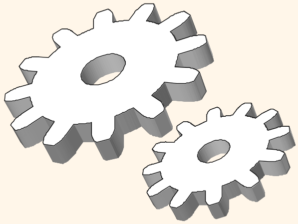
Question 2

- Can you find another schedule for the below transaction that fulfills the 2PL protocol? (You can omit the arithmetic operations in this schedule.)

T_1	T_2
R(A)	**R**(A)
R(B)	**R**(B)
A = B + 10	B = A + 20
W(A)	**W**(B)



	T_1	T_2	
	X(A)		
	S(B)		
Growing	**R**(A)		
	R(B)		
	W(A)		
	U(A)		
Shrinking		S(A)	
		R(A)	
	U(B)		
		X(B)	Growing
		R(B)	
		W(B)	
		U(B)	Shrinking
		U(A)	

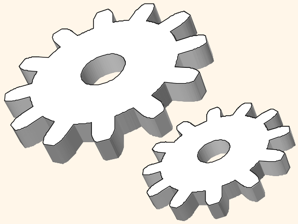


Weakness of 2PL: Deadlock

- May result in deadlock.
  - There exists a set of transactions such that every transaction in the set is waiting for another transaction in the set (to release the lock(s)).
  - None of the transactions in this set can proceed.

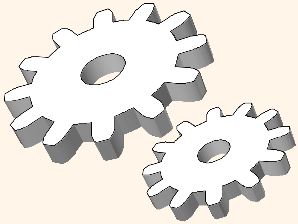
- Deadlock is a *serious* issue, which significantly degrades the performance in a database.

Example of Deadlock



T_3	T_4
X(A)	X(B)
Read(A)	Read(B)
Write(A)	Write(B)
X(B)	X(A)

- Neither T_3 nor T_4 can progress.
- A **deadlock**: either T_3 or T_4 must be rolled back so that its locks are released.

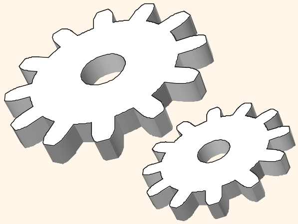


Example of Deadlock

T_1	T_2
R(A)	**R**(A)
R(B)	**R**(B)
A = B + 10	B = A + 20
W(A)	**W**(B)

	T_1	T_2	
	X(A)		
		X(B)	
Cannot grant this lock	S(B)		
		S(A)	Cannot grant this lock

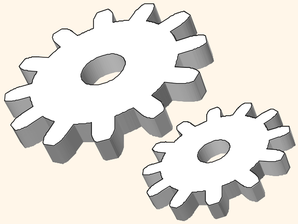
- Neither T_1 nor T_2 can progress.
- Either T_1 or T_2 must be rolled back so that its locks are released.



How to Handle Deadlocks?

- Deadlock detection + recovery
  - Detect the deadlock when the database system is running.
  - Recover from the deadlock.
- Deadlock prevention
  - Ensure the deadlock will never happen.

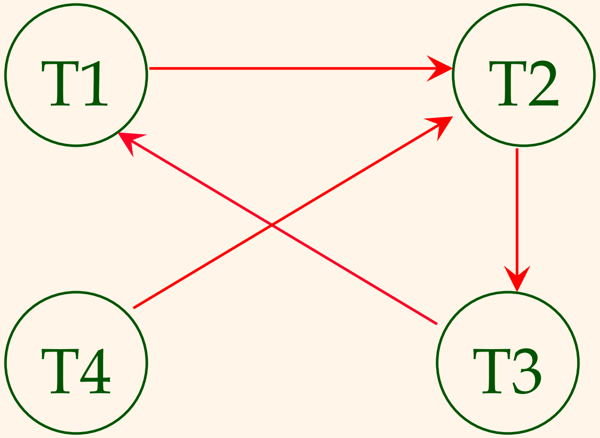
Deadlock Detection



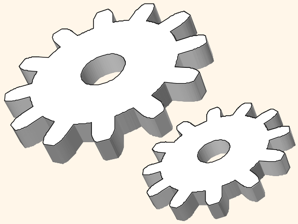
- Wait-for graph
  - Each vertex is a transaction.
  - An edge $T_1 \rightarrow T_2$ if T_1 is waiting for T_2 to release a lock.
  - The edge is removed when T_1 obtains the lock (T_2 releases that lock).
- The system has a deadlock *if and only if* there is a *cycle* in the graph.

Example of Deadlock Detection

T1:	S(A), **R**(A),		S(B)				
T2:		X(B),**W**(B)			X(C)		
T3:				S(C), **R**(C)			X(A)
T4:						X(B)	



Question 3

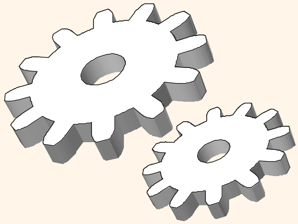


- Below are 4 transactions involving 4 data objects A, B, C, D in the database system. The r.h.s. table presents a schedule of these 4 transactions. Detect if executing this schedule will result in a deadlock.

T_1	S(A)	R(A)	X(B)	W(B)	U(A)	U(B)
T_2	S(C)	R(C)	X(A)	W(A)	U(C)	U(A)
T_3	S(B)	R(B)	X(C)	W(C)	U(B)	U(C)
T_4	S(D)	R(D)	X(A)	W(A)	U(D)	U(A)

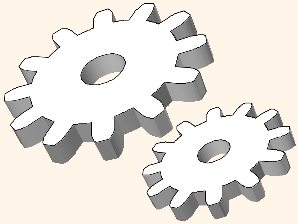
T_1	T_2	T_3	T_4
S(A) R(A)			
	S(C) R(C)		
		S(B) R(B)	
			S(D) R(D)
	X(A)		
		X(C)	
			X(A)
X(B)			

Deadlock Recovery



- When deadlock is detected, we need to *rollback* a transaction to *destroy the cycle* in the *wait-for graph*.
  - Other waiting transactions can proceed.
- Which one to roll back?
  - The one that incurs the minimum “cost”.
  - Different criteria:
    - Current execution time, age, etc.

Deadlock Prevention



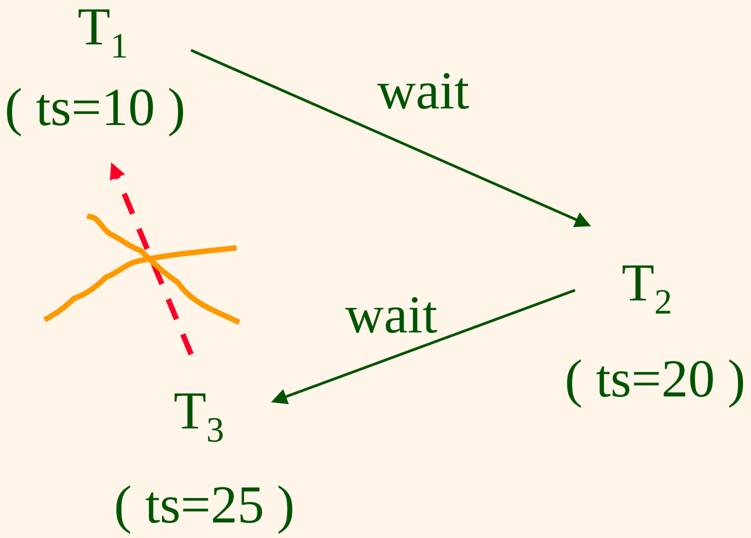
- **Deadlock prevention** ensures that the system will never deadlock.

- Some simple strategies:
  - **Resource ordering:** Impose an ordering on the data objects, and require a transaction to lock in that order (*not realistic in most cases*)
  - **Timeout:** a transaction waits for a lock only for *L* seconds. After that, it is rolled back. (*difficult to select a good L*)

- Next we will see more strategies.

Deadlock Prevention

- **A wait-die approach**
  - When each transaction starts, the database system assigns timestamp (ts) for it.
  - An older transaction is allowed to wait for a younger one to release lock.
  - A younger transaction never waits for an older one
    - Instead, it is rolled back.



T_1

(ts=10)

wait

T_2

(ts=20)

wait

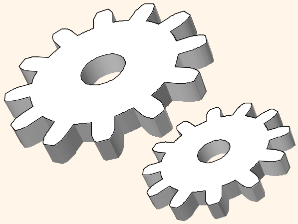
T_3

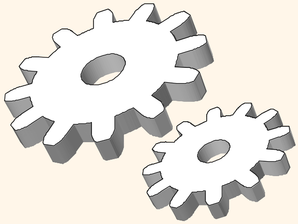
(ts=25)

An Example

A younger transaction never waits for an older one

T1 (ts=10)	T2 (ts=20)	T3(ts=25)
X(A)		
R(A)		
	S(B)	
	R(B)	
		X(C)
X(B)		
	S(C)	
		S(A)





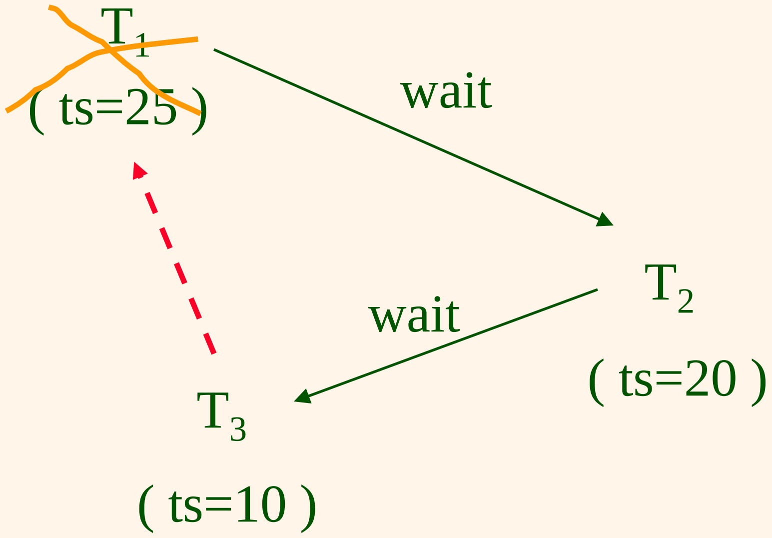
An Example

A younger transaction never waits for an older one

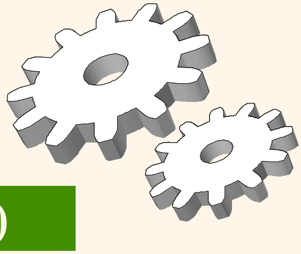
T1 (ts=10)	T2 (ts=20)	T3(ts=25)
X(A)		
R(A)		
	S(B)	
	R(B)	Rolled-back
X(B)		
	S(C)	
	R(C)	
	U(B)	
X(B)		
	U(C)	
U(A)		
U(B)		

Deadlock Prevention

- **A wound-wait approach**
  - When each transaction starts, the database system assigns timestamp for it.
  - An older transaction “wounds” (forces rollback) of a younger transaction instead of waiting for it.
  - A younger transaction waits for an older one.

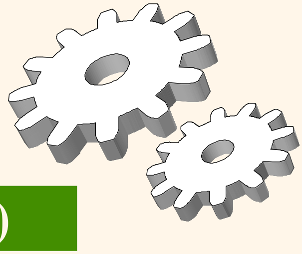


An Example



An older transaction never waits for a younger one

T1 (ts=25)	T2(ts=20)	T3(ts=10)
		X(C)
	S(B)	
	R(B)	
X(A)		
R(A)		
X(B)		
	S(C)	
		S(A)

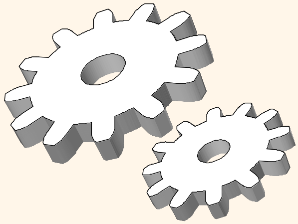


An Example

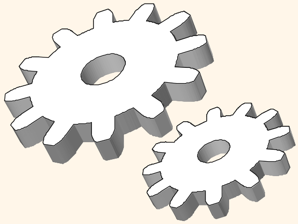
An older transaction never waits for a younger one

T1 (ts=25)	T2(ts=20)	T3(ts=10)
		X(C)
	S(B)	
	R(B)	
Rolled-back		
	S(C)	
		S(A)
		W(C)
		R(A)
		U(C)
	S(C)	
	R(C)	
	U(B)	
	U(C)	
		U(A)

Deadlock Prevention



- If a transaction rolls back and restarts
  - It will be given its original timestamp.
  - Every transaction will eventually become the *oldest* with *highest priority* (No starvation).

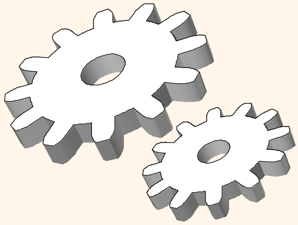


Question 4

- Describe what would happen if you use the following deadlock prevention approaches to execute the following schedule.
  - The wait-die approach
  - The wound-wait approach

Assume that the four transactions start in the order of T2, T1, T4, and T3.

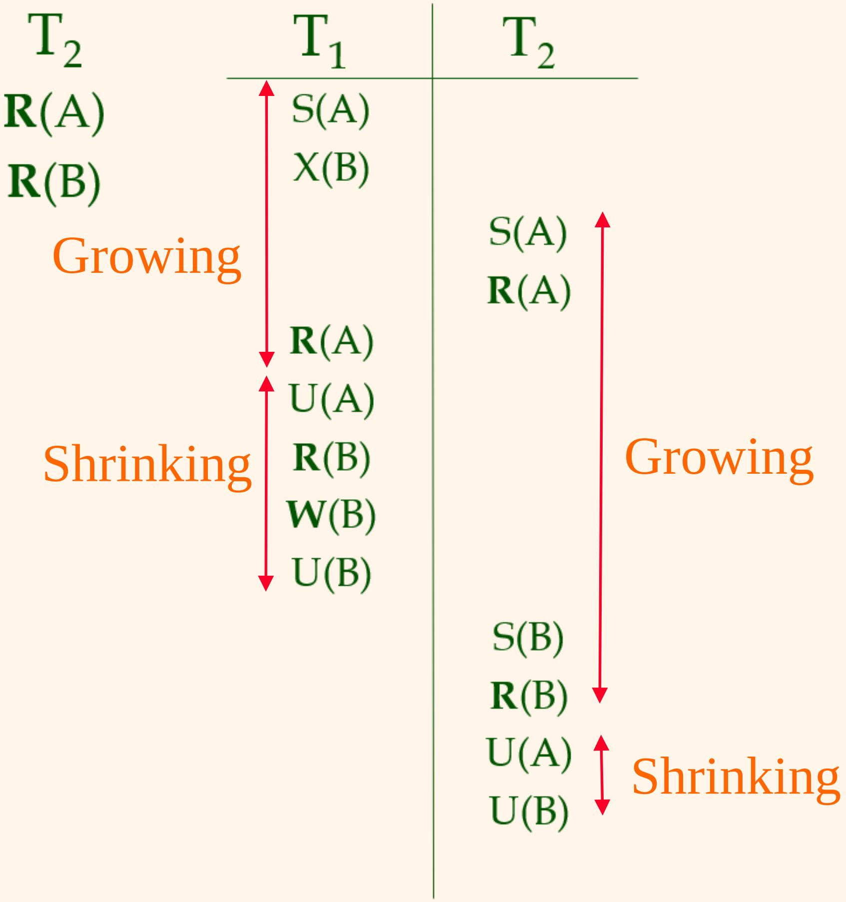
T1:	S(A), **R**(A),		S(B)				
T2:		X(B),**W**(B)			X(C)		
T3:				S(C), **R**(C)			X(A)
T4:						X(B)	



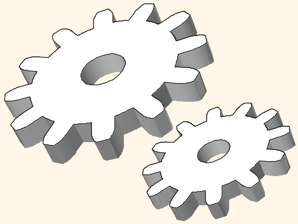
Solution to Question 1

- Yes

T_1	T_2
R(A)	**R**(A)
R(B)	**R**(B)
B = B + A	
W(B)	



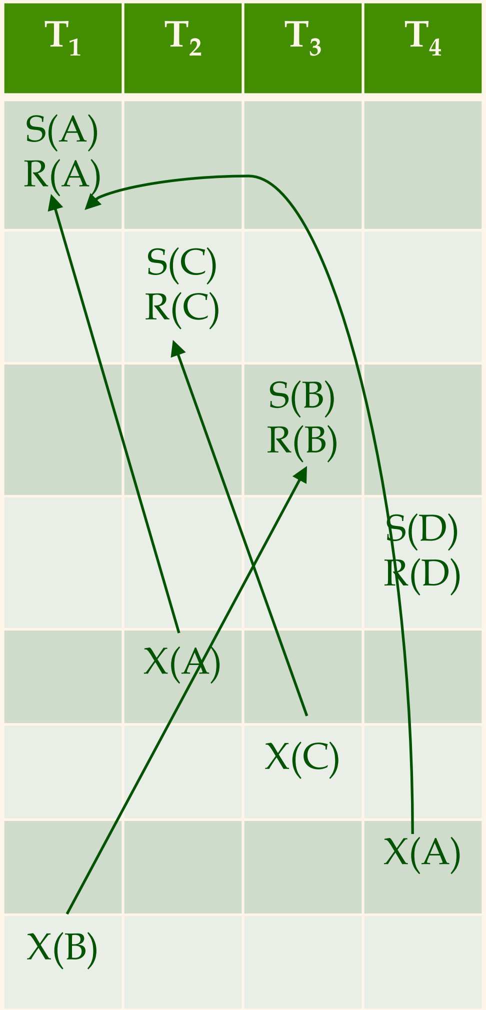
	T_1	T_2	
Growing	S(A)		
	X(B)		
		S(A)	Growing
		R(A)	
	R(A)		
Shrinking	U(A)		
	R(B)		
	W(B)		
	U(B)		
		S(B)	
		R(B)	
		U(A)	Shrinking
		U(B)	



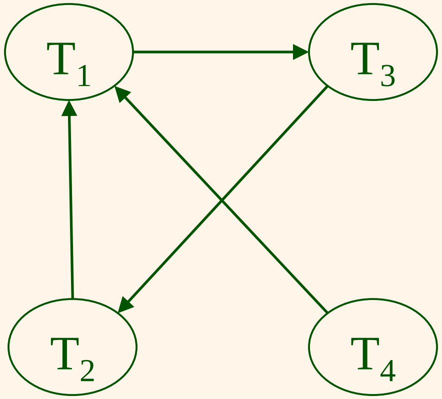
Solution to Question 2

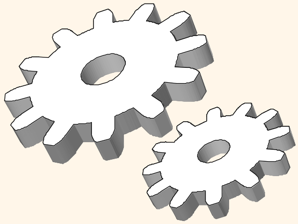
	T_1	T_2	
	X(A)		
	S(B)		
Growing	**R**(A)		
	R(B)		
	W(A)		
Shrinking	U(A)		
	U(B)		
		S(A)	
		R(A)	
		X(B)	Growing
		R(B)	
		W(B)	
		U(B)	Shrinking
		U(A)	

Solution to Question 3



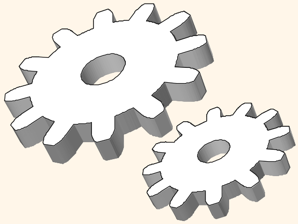
T_1	T_2	T_3	T_4
S(A) R(A)			
	S(C) R(C)		
		S(B) R(B)	
			S(D) R(D)
	X(A)		
		X(C)	
			X(A)
X(B)			





Solution to Question 4

- Wait-die approach
  - T2 starts earlier than T1. T1 is rolled back when it requests a shared-lock on B.
  - T2 starts earlier than T3. T2 can wait for T3 when it requests an exclusive-lock on C.
  - T2 starts earlier than T4. T4 is rolled back when it requests an exclusive-lock on B.
  - Since T1 is rolled back, T3: X(A) can be granted and T3 can proceed.
  - Both T2 and T3 can be executed, whereas T1 and T4 will be rolled back.



Solution to Question 4

- Wound-wait approach
  - T2 starts earlier than T1. T1 can wait for T2 when it requests a shared-lock on B.
  - T2 starts earlier than T3. When T2 requests an exclusive-lock on C, it will force T3 to roll back instead of waiting for it.
  - T2 starts earlier than T4. When T4 requests an exclusive-lock on B, T4 can wait for T2 to release its exclusive-lock on B.
  - T1, T2, and T4 can be executed, whereas T3 will be rolled back.