

**CS470: Introduction to Database Management Systems**

**Transaction Management  
Concurrency Control Mechanisms  
(Textbook Chapters 17 and 18)**

**V Kumar  
School of Computing and Engineering  
University of Missouri-Kansas City**

## Transaction Management

- Q. What is a transaction?
- A. A mechanism for applying the desired modifications/operations to the final database. A final database is the most up-to-date copy of the database.

### A user view of a transaction

**Transaction name:** Debit\_Credit.

**Transaction function:** To perform debit and credit operation on an account.

#### *Debit*

***Begin\_Transaction;***

***Get message (\* from terminal\*);***

***Extract Account\_no, Teller, Branch, Amount from message;***

***Find Account (Account\_no) in database;***

***If not found or Account\_bal < Amount or Amount < 0 then send negative message  
else***

***begin***

***Account\_bal := Account\_bal - Amount;***

***post history record on Account (Amount);***

***Cash\_drawer (Teller) := Amount;***

***Branch\_bal (Branch) := Branch\_bal (Branch) - Amount;***

***Put message ('New balance = ', Account\_bal);***

***end;***

***Commit;***

This is a small transaction. It reads very few database records and updates them. A large transaction usually reads a large number of data items and does significant amount of processing on each data item. A large transaction may run for a day or a week or could be a month. For example the transaction that produces a summary statement for each account in a bank may look like:

Summary transaction

```
SELECT *  
FROM Account, History  
WHERE Account.Account_no = History.Account_no and History_date Last_report  
GROUPED BY Account.Account_no  
ASCENDING BY Account.Account_address
```

The answer appears sorted by mailing address. This is a very long transaction and may take hours to complete if not managed efficiently.

### Database System View of a Transaction

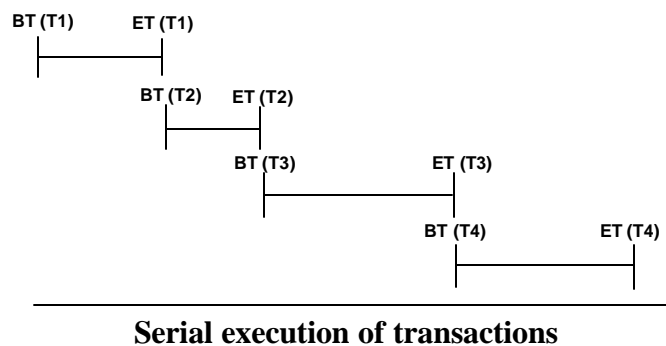
A database system interprets a transaction not as an application program but a logical sequence of low-level operations *read* and *write* (referred to as primitives). For example, the system will look at the debit\_credit transaction as follows:

Transaction Debit\_credit

$read(X)$       ( $X$  is the withdrawal amount)  
 $A := A - X$       ( $Debit X$  from the account  $A$ )  
 $write(A)$       ( $Write$  new account value to the database)  
 $read(Y)$       ( $Y$  is the credit amount)  
 $A := A + Y$       ( $Add Y$  to the existing account  $A$ )  
 $write(A)$       ( $Write$  new value of account  $A$  to the database)

In a database system many transactions are executed. Basically there are two ways of executing a set of transactions: (a) Serially or (b) Concurrently.

**Serial Execution:** In a serial execution transactions are executed strictly serially. Thus, Transaction  $T_i$  completes and writes its results to the database then only the next transaction  $T_j$  is scheduled for execution. This means at one time there is only one transaction is being executed in the system. Graphically,



In a serial execution, therefore,  $T_i$  begins only when  $T_{i-1}$  ends and so on. At any instant of time a data item is used by only one transaction, that is, a data item is not shared between two or more transactions. This also means that a transaction does not *interfere* the execution of any other transaction. Good things about serial execution

1. Correct execution, i.e., if the input is correct then output will be correct.
2. Fast execution since all the resources are available to the active.

The worst thing about serial execution is very inefficient resource utilization. The aim of database system is to utilize resources efficiently, generate less transaction waiting time and preserve database consistency. The only way to improve resource utilization is to execute transactions concurrently or simultaneously.

### Example of a serial execution

Suppose data items  $X = 10$ ,  $Y = 6$ , and  $N = 1$  and  $T_1$  and  $T_2$  are transactions.

<b><math>T_1</math></b>	<b><math>T_2</math></b>
$read(X)$	$read(X)$
$X := X + N$	$X := X + N$
$write(X)$	$write(X)$
$read(Y)$	
$Y := Y + N$	
$write(Y)$	

We execute this transaction serially as follows:



<b>Time</b>	<i>read (A)</i>	{A = 500}	<i>read (A)</i>	{A = 500}
	<i>A := A+100</i>	{A = 600}	<i>A:= A-50</i>	{A = 450}
	<i>write (A)</i>	{ A = 600}	<i>write (A)</i>	{ A = 450}

Final value of A = 450. The credit of T1 is missing (lost update) from the account.

**Dirty read:** Reading of a non-existent value of A by T2. If T1 updates A which is then read by T2, then if T1 aborts T2 will have read a value of A which never existed.

	<i>T1 (Credit)</i>		<i>T2 (Debit)</i>	
<b>Time</b>	<i>read (A)</i>	{A = 500}		
	<i>A := A+100</i>	{A = 600}		
	<i>write (A)</i>	{ A = 600}		
			<i>read (A)</i>	{A = 600}
			<i>A:= A-50</i>	{A = 550}
			<i>write (A)</i>	{ A = 550}

*T1 failed to complete*

T1 modified A = 600. T2 read A = 600. But T1 failed and its effect is removed from the database, so A is restored to its old value, i.e., A = 500. A = 600 is a nonexistent value but read (reading dirty data) by T2.

**Unrepeatable read:** If T2 reads A, which is then altered by T1 and T1 commits. When T2 re-reads A it will find different value of A in its second read.

	<i>T1 (Credit)</i>		<i>T2 (Debit)</i>	
<b>Time</b>	<i>read (A)</i>	{A = 500}		
	<i>A := A+100</i>	{A = 600}	<i>read (A)</i>	{A = 500}
	<i>write (A)</i>	{ A = 600}	<i>A:= A-50</i>	{A = 450}
			<i>read (A)</i>	{ A = 600}

In this execution T1 reads A = 500, T2 read A = 500. T1 modifies A to 600. When T2 re-reads A it gets A = 600. This should not be the case. T2 in the same execution should get only one value of A (500 or 600 and not both).

In serial execution these problems (dirty read, unrepeatable read, and lost update) would not arise since serial execution does not share data items. This means we can use the results of serial execution as a measure of correctness and concurrent execution for improving resource utilization. We need *serialization* of concurrent transaction.

**Serialization of concurrent transactions:** Process of managing the execution of a set of transactions in such a way that their concurrent execution produces the same end result as if they were run serially.

**Properties of Transactions:** Before we study serialization scheme we discuss the properties of transactions, which are essential to serialize concurrent execution of transactions.

**Atomicity:** This property has two states:

- Done or never started. Done - a transaction must complete successfully and its effect should be visible in the database.
- Never started - If a transaction fails during execution then all its modifications must be undone to bring back the database to the last consistent state, i.e., remove the effect of failed transaction.

**Consistency:** If the transaction code is correct then a transaction, at the end of its execution, must leave the database consistent.

**Isolation:** A transaction must execute without interference from other concurrent transactions and its intermediate modifications to data must not be visible to other transactions.

**Durability or permanency:** The effect of a completed transaction must persist in the database, i.e., its updates must be available to other transaction.

To eliminate **Lost update, Dirty read, and Unrepeatable read** problems **and to** implement atomicity, consistency, and isolation we need two additional operations (other than read and write), which are called **Lock** and **Unlock**, which are applied on a data item.

**Lock (X):** If a transaction T1 applies Lock on data item X, then X is locked and it is not available to any other transaction.

**Unlock (X):** T1 Unlocks X. X is available to other transactions.

### Types of a Lock

**Shared lock:** A Read operation does not change the value of a data item. Hence a data item can be read by two different transactions simultaneously under share lock mode. So only to read a data item T1 will do: *Share lock (X), then Read (X), and finally Unlock (X).*

**Exclusive lock:** A write operation changes the value of the data item. Hence two write operations from two different transactions or a write from T1 and a read from T2 are not allowed. A data item can be modified only under Exclusive lock. To modify a data item T1 will do: *Exclusive lock (X), then Write (X) and finally Unlock (X).*

When these locks are applied, then a transaction must behave in a special way. This special behavior of a transaction is referred to as *well-formed*.

**Well-formed:** A transaction is well-formed if it does not lock a locked data item and it does not try to unlock an unlocked data item.

**Examples:** T1 and T2 are two transactions. They are executed under locking as follows. T1 locks A in exclusive mode. When T2 wants to lock A, it finds it locked by T1 so T2 waits for Unlock on A by T1. When A is released then T2 locks A and begins execution.

Suppose a lock on a data item is applied, the data item is processed and it is unlocked immediately after reading/writing is completed as follows. Initial values of A = 10 and B = 20.

<p>T1</p> <p>Lock (A)</p> <p>read (A)            {A = 10}</p> <p>A := A + 100</p> <p>write (A)            {A = 110}</p> <p>Unlock (A)</p> <p>Lock (B)</p> <p>read (B)            {B = 100}</p> <p>B := B + 10</p> <p>write (B)            {B = 110}</p> <p>Unlock (B)</p>	<p>T2</p> <p>Lock (B)</p> <p>read (B)            {B = 20}</p> <p>B := B * 5</p> <p>write (B)            {B = 100}</p> <p>Unlock (B)</p> <p>Lock (A)</p> <p>Read (A)            {A = 110}</p> <p>A := A + 20</p> <p>Write (A)            {A = 130}</p> <p>Unlock (A)</p>
--	---

The final value of A = 130 and B = 110. This is not correct because a serial execution of T1 and then T2 will produce A = 130 and B = 150. This means the above method of locking and unlocking is not correct. The correct way of locking must follow *two-phase* scheme.

### Two-Phase Scheme

A transaction must not lock any data item once it has unlocked some data item. This scheme has two phases:

**Growing phase:** A transaction applies locks on desired data items.

**Shrinking phase:** A transaction unlocks all data items it locked in the growing phase.

Serialization of concurrent transactions requires that all transactions must be *well-formed* and *two-phase*. The mechanism that serializes concurrent transactions is called *Concurrency Control Mechanisms* (CCMs).

### Concurrency Control Mechanisms

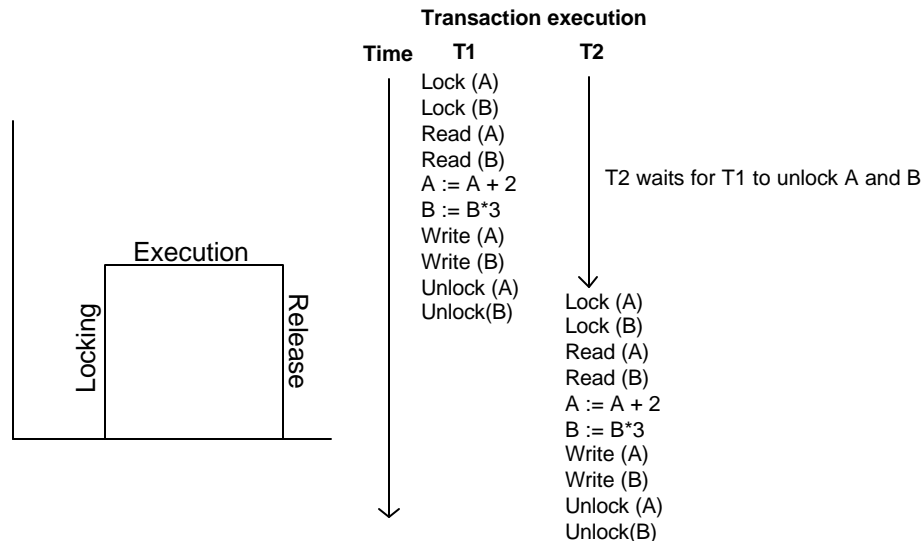
A CCM assume that the transaction code is free from programming errors and guarantee that all transactions will be well-formed and would follow two-phase locking policy. We identify three phases in transaction execution: (a) Growing (locking), (b) Execution (data modification) and (c) Shrinking (releasing locks). Under two phase strategy, lock and unlock operations can be applied in four different ways (a) Simultaneous Locking and Simultaneous Release, (b) Incremental Locking and Simultaneous Release, (c) Simultaneous Locking and Incremental Release, and (d) Incremental Locking and Incremental Release. We will discuss each of these mechanisms in detail.

We introduce the concept of schedule. A schedule is a history of the execution of a set of transactions. In a schedule read and write operations of concurrent transactions are listed in the order they are requested.

### Example of a schedule

T1R(A) T1R(B) T1R(C) T1W(A) T1W(B) T1W(C) T2R(A) T2W(A)

**Simultaneous locking and Simultaneous release (Static locking):** All concurrent transactions go through *Growing* phase (apply locks on desired data items), *Execution* phase and finally *Shrinking* (release/unlocking) phase. Graphically this CCM can be illustrated as follows:



### CCM: Simultaneous Locking and Simultaneous Release

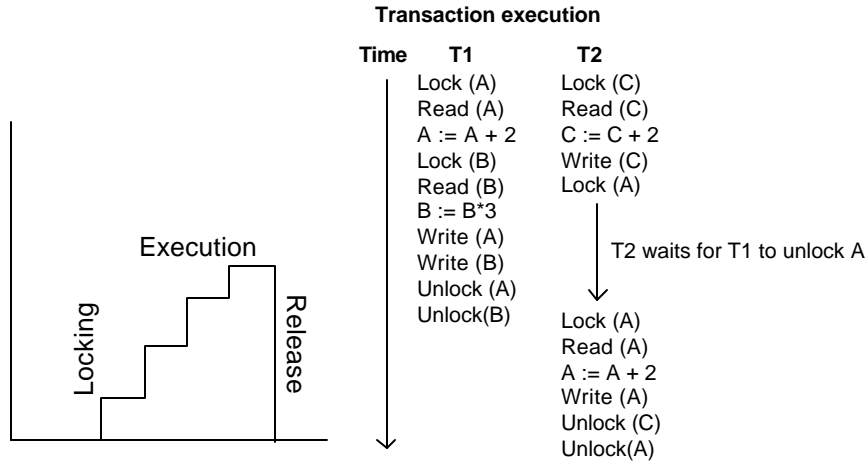
**Schedule:** T1R(A) T1R(B) T1W(A) T1W(B) T2R(A) T2R(B) T2W(A) T2W(B)

**Implementation:** A preprocessor must identify all the desired items referenced by the transaction. Locking phase, Execution phase, and Release phase are mutually exclusive.

**Advantage:** Simple implementation. May be good for batch transactions.

**Disadvantage:** Redundant locking. Not possible for interactive transactions since the next data requirement may depend on user response.

**Incremental locking Simultaneous release:** This algorithm is widely known as General Waiting. In this mechanism a transaction during execution asks for an item, locks it if it is free, and modifies (execution phase) the data item. At the end of modification the transaction asks for the next data item, locks it if it is free and modifies it. At the end of execution phase the transaction unlocks simultaneously all the data items it locked. If a requested data item is not free then the transaction waits (blocked) for the item to be come free. So the data items are locked by a transaction as execution proceeds, thus the execution and locking phases are mixed together. The release phase is independent. Graphically:



### Incremental Locking and Simultaneous Release

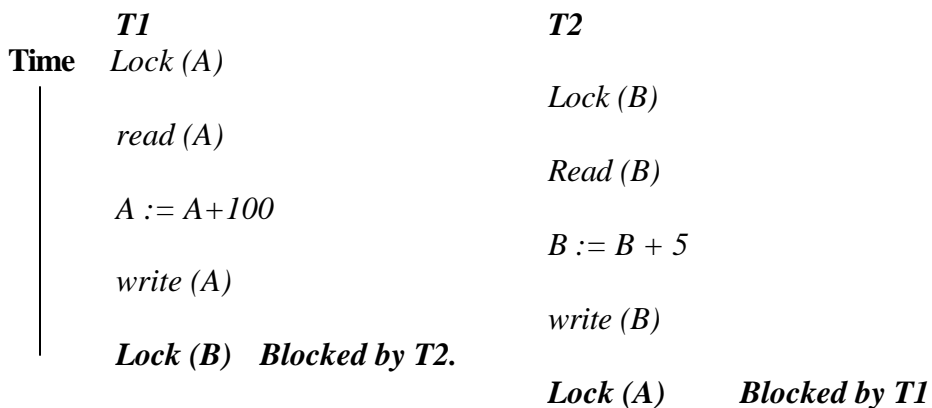
**Schedule:** T1R(A) T2R(C) T2W(C) T1R(B) T1W(A) T1W(B) T2R(A) T2W(A)

This is serializable since it is equivalent to a serial schedule. This means this CCM produces serializable schedule hence it is a correct algorithm.

**Advantages:** No redundant locks. Transaction may wait less for a data item.

**Disadvantages:** Deadlock may happen. Some entities may remain locked even after they have been modified. For example, item A remains locked even its modification is over.

**Deadlock:** The following example shows the occurrence of deadlock.



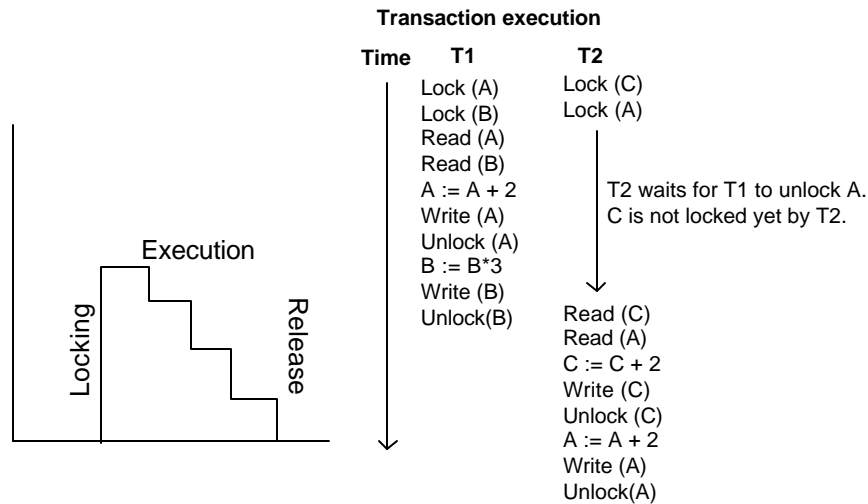
Neither T1 nor T2 can proceed further. This means no transaction can complete, thus, this will not produce a serializable schedule.

**Schedule:** If T1R(A) T1W(A) T2R(B) T2W(B) T1R(B) blocked T2R(A) blocked.

This is a cyclic schedule so not serializable. One of the transactions, therefore, is rolled-back to achieve serialization. T1 and T2 cannot proceed since each is waiting for other to release data items. In this situation a forced release is required where one of the transactions (T1 or T2) is selected for roll-back. If T1 is selected then all its operations on data items (in this case A) are undone. When the roll-back is complete T1 can be restarted from the beginning.

**Simultaneous locking and Incremental release:** To reduce the locking duration from data items release phase can be made incremental. In this algorithm, locks are applied simultaneously

but as soon as modification of an item is over, the item is released. Execution and release phase are combined and can be illustrated as follows:



### Simultaneous Locking and Incremental Release

**Schedule:** T1R(A) T1R(B) T1W(A) T1W(B) T2R(C) T2R(A) T2W(C) T2W(A)

**Advantages:** Reduces transaction waiting time by incremental release. No deadlock.

**Disadvantages:** It suffers with cascade roll-back (to undo the operations of one transaction many other transactions may have to be undone). Redundant locking.

Example:

T1	T2	T3	T4
Lock (A)	Lock (B)	Lock (C)	Lock (C)    waits
Lock (E)	Lock (A)    waits	Lock (B)    waits	waits
read (A)	waits	waits	waits
read (E)	waits	waits	waits
A := A+E	waits	waits	waits
write (A)	waits	waits	waits
Unlock (A)	read (B)	waits	waits
E := E*2	B := B*2	waits	waits
write (E)	write (B)	waits	waits
Unlock (E)	Unlock (B)	read (C)	waits
FAIL	read (A)	C := C*2	waits
	A := A*2	write (C)	waits
	write (A)	Unlock (C)	read (C)
	Unlock (A)	read (B)	C := C*2
		B := B+2	write (C)
		write (B)	Unlock (C)
		Unlock (B)	

Schedule

T1R(A) T1R(E) T1W(A) T2R(B) T1W(E) T2W(B) T3R(C) T2R(A) T3W(C) T2W(A) T3W(C)  
T4R(C) T3R(B) T4W(C) T3W(B).

Execution:

T2 gets input from T1; T2 depends on T1.

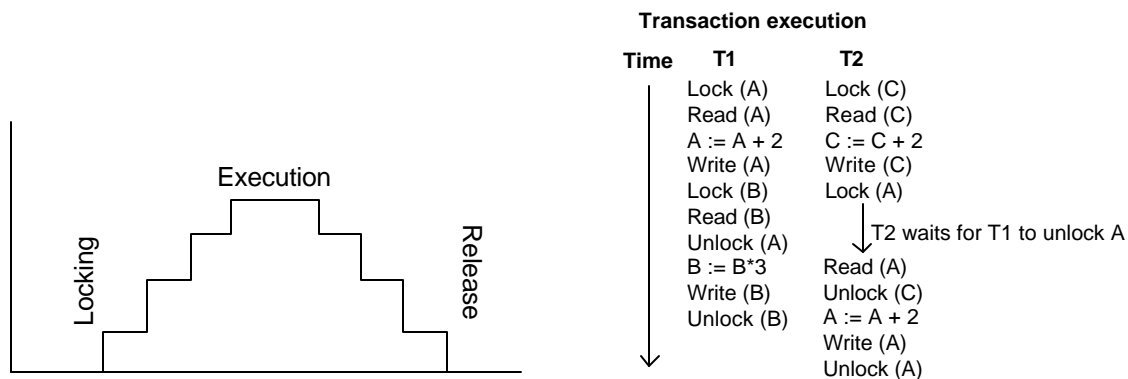
T3 gets input from T2; T3 depends on T2.

T4 gets input from T3; T4 depends on T3.

**Cascade roll-back:** If T1 fails T1 will be rolled-back. If T1 is rolled-back T2 must be rolled-back. If T2 is rolled-back, T3 must be rolled-back. and if T3 is rolled-back T4 must be rolled-back.

**Solution:** T1 must commit first then T2 then T3 and finally T4.

**Incremental locking Incremental release:** Further minimizes transaction waiting time. Locking + Execution phase, and Execution + Release phase. Locks are acquired incrementally and released incrementally. Graphically:



### Incremental Locking and Incremental Release

**Advantage:** Reduced transaction waiting time so higher number of concurrent transactions in the system.

**Disadvantages:** Allows deadlock and cascade roll-back to happen.

**Analysis:** Incremental locking and simultaneous release (General waiting) is the most commonly used CCM. It allows deadlock to happen, therefore, deadlock detection and resolution is required. There are many ways for detecting deadlock and many ways for selecting a transaction to be rolled-back to resolve a deadlock. However, numerous performance studies (experimental and analytical) show that deadlocks are not frequent, so deadlock detection and resolution are not expensive.

We have studied several CCMs based on two-phase locking policy. Locking dynamically defines the execution order of transactions. This means that during execution of concurrent transactions, the order of execution is decided by locking. If the execution order is predefined then also serialization can be achieved. This technique is known as Timestamping.

### CCM based on timestamp

**Timestamp:** An increasing integer number

**Transaction timestamp (ts):** Associated with each transaction. The associated timestamp determines a transaction's age, i.e., when it did enter in the system. A transaction's timestamp is unique, i.e., no two transactions' timestamp can be the same.

**Data item timestamp:** Timestamp associated with each data item. When a transaction modifies a data item it stamps the data item with its timestamp, indicating that the data item was modified by this transaction.

**Example**

- T1: 1 is the value of timestamp of transaction T1.
- T2: 2 is the timestamp of transaction T2.
- E1: Entity E1 was last modified by T1.
- T1 is older than T2, i.e., T1 arrived in the system before T2.

We will study two CCMs based on timestamp.

**Simple Timestamp technique**

**Accessing a data item  $D_x$  by a transaction  $T_y$**

```

If  $x$  (data item  $D$ 's  $ts$ )  $<$   $y$  (transaction  $ts$ ) then
begin
    access and modify the data item  $D$ ;
    overwrite  $x$  with  $y$  (transaction's  $ts$ ), i.e.,  $x := y$ 
end
else roll-back  $T_y$ ;

```

**Example**

Transaction	Needs data item		
T1	A	B	C
T2	B	C	D
T3	A	B	F

Assume timestamps of all data items = 0.

Execution (Apply above algorithm)

```

T1 begins   Asks for A 0 < 1 OK A's ts = 1 (A1)
T2 begins   Asks for B 0 < 2 OK B's ts = 2 (B2)
T3 begins   Asks for A 1 < 3 OK A's ts = 3 (A3)
T1          Asks for B 2 1 Not OK T1 must roll-back.

```

This means a younger transaction (T2) has already modified B, which cannot happen in a serial execution. This operation will produce an illegal schedule (non-serializable schedule), therefore, T1 must be rolled-back. But rolling-back T1 will undo A and A's  $ts = 0$ . T3 accessed A after T1, so T3 must also be rolled-back. T1 and T3 will then start again with higher timestamps. Remember, the new timestamps for T1 and T3 must be larger than the most recent timestamp.

```

T2          Asks for C 0 < 2 OK C's ts = 2 (C2)
T2          Asks for D 0 < 2 OK D's ts = 2 (D2) T2 commits.

```

Serialization is achieved by rolling-back transactions. There is no blocking and waiting for data items.

**Problem:** This CCM does is unable to differentiate between a read lock and a write lock. Result: If T1 reads data item A after T2 did, T1 must be rolled-back since the timestamp of A will be 2 (updated by T2's read).

**Solution:** Each entity is associated with two timestamps: one read timestamp (rt) and one write timestamp (wt).

**Algorithm:** A read operation is managed by the read timestamp and a write by the write timestamp.

### **Read operation**

```
Get wt (write timestamp of the entity).
If wt < ts then
begin
    overwrite rt by the larger of the ts and wt;
    read data item
end
else roll-back the transaction;
```

### **Write operation**

```
Get rt of the data item
If rt < ts then
begin
    overwrite wt by ts;
    modify data item
end
else roll-back the transaction;
```

### **Two consecutive writes**

```
Compare wt and ts
If wt < ts then
begin
    modify the data item;
    overwrite wt with ts
end
else roll-back the transaction;
```

### **Two-phase vs. Timestamping**

As mentioned before, under two-phase policy the execution order of concurrent transactions are determined dynamically. During execution it is not possible to know which one of the  $n$  transactions will get the desired data item. Under timestamp technique, locking data item is not necessary and there is no two-phase (growing/shrinking) process. This is because the order of transaction execution is predefined by assigning timestamps to incoming transactions. A conflict under Timestamping is always resolved by rolling-back one of the conflicting transactions. In two-phase conflicting one of the conflicting transactions is blocked. Since timestamp uses transaction roll-back there is no deadlock possible. In two-phase policy, deadlocks may occur and mechanisms required for their detection and resolution. Two-phase CCMs are easier to implement since they do not require timestamps. Timestamping is difficult since generation and ever increasing value of a timestamp is not easy to handle. After some time the value of a timestamp may be too large for the system to store. Significant amount of storage space is required for storing timestamps and updating them is time consuming. In two-phase CCMs these do not exist making their implementation easier.