

Databases



Transaction Management

First Lecture

Levente Erős – eros@db.bme.hu

Ruba AlMahasneh – mahasnehr@gsuite.tmit.bme.hu



Agenda

1. Basic terms
 - a. Transactions
 - b. Items
 - c. Operations
2. Schedule
 - a. Serializability of Schedules
3. Concurrency Control
 - a. Concurrency Possible Problems [Cases to avoid with Schedules-Problems with schedules]:
 - b. Scheduler
 - c. Concurrency Control Techniques
4. Deadlocks and Livelocks- Concurrent Processing
 - a. Deadlock management techniques
 - b. The Wait-for Graph
 - c. Livelocks
5. Transaction models
 - a. Simple Transaction Model
 - b. R/W model
6. Precedence graph
7. Two-phase locking
8. Granularity and Performance
9. States of a transaction

Transactions

- **Transaction:** is a single execution of a single program. Composed of read/write operations. This program may be a simple query or multiple queries/statements expressed in one of the query languages.
- **Ques:** Can several independent executions of the same program run in progress simultaneously ? give an example?
- A transaction reads and/or writes data *to and from* the database, into private workspace, where all computations are performed.

Example: Money Transfer

If I want to withdraw some money from the ATM machine

- Does withdrawing money contain a single step? Or multiple steps?
- Executing these series of steps must meet some requirements.
- **Atomic**; all steps carried out or none.
- **Consistent**; only fully completed Trs. will affect the DB.
- **Isolated**; Transactions are unaware of other transactions performing operations on the DB. The output of each transaction is not affected by any other Trs.
- **Durable**; the effects of Trs are preserved in the DB.

Data Items

- What is a data item?
- **Items** : units of data to which access is controlled (size may vary, how?)

Give examples?

- The database must be partitioned into **items**. Why?
- The size of items used by system is often called its **granularity**. "**fine-grained**" system uses small items and "**coarse-grained**" one uses large items.

Schedule

Schedule: the order in which the steps of the transactions (read, write and so on) are executed.

Scheduler: is a portion of the database system that arbitrates between conflicting requests. It controls access rights of transactions.

Schedules

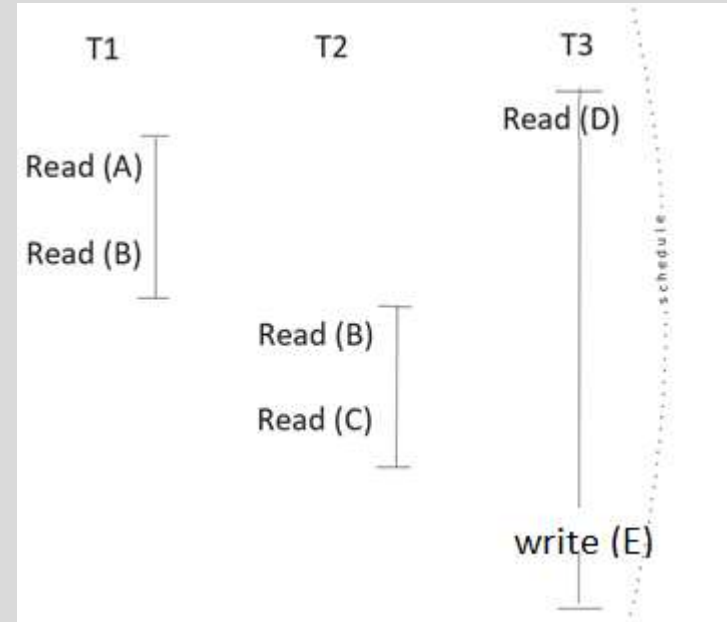
- **Serial** : one transaction at a time - no overlapping transactions
- **Non-Serial**: overlapping transactions (most common in practice)
- **Non-Serial can be Serializable**
- A schedule said to be serializable **IIF** all its effects are equal to a serial schedule = **serial equivalent exists**

Serializability of Schedules

Ques:

Spot the serial and non serial schedule(s) parts.
Do serialized **schedules** exist in practice? Why?

Since Serial schedules are very unlikely in many practical cases → Serial equivalent guarantees the same order as that obtained by running the same transactions *serially* in some order.



Serial Schedules

VS

Serializable Schedules

No concurrency is allowed.

Thus, all the transactions necessarily execute serially one after the other.

Serial schedules lead to less resource utilization and CPU throughput.

Serial Schedules are less efficient as compared to serializable schedules.

(due to above reason)

Concurrency is allowed.

Thus, multiple transactions can execute concurrently.

Serializable schedules improve both resource utilization and CPU throughput.

Serializable Schedules are always better than serial schedules.

(due to above reason)

Serializability of Schedules

T_1	T_2	T_1	T_2	T_1	T_2
READ A		READ A		READ A	
A:=A-10		A:=A-10	READ B	A:=A-10	
WRITE A			B:=B-20	WRITE A	READ B
READ B		WRITE A	WRITE B	READ B	B:=B-20
B:=B+10		READ B		B:=B+10	WRITE B
WRITE B	READ B	B:=B+10	READ C	WRITE B	READ C
	B:=B-20	WRITE B	C:=C+20		C:=C+20
	WRITE B		WRITE C		WRITE C
	READ C				
	C:=C+20				
	WRITE C				
(a)		(b)		(c)	

Discussion

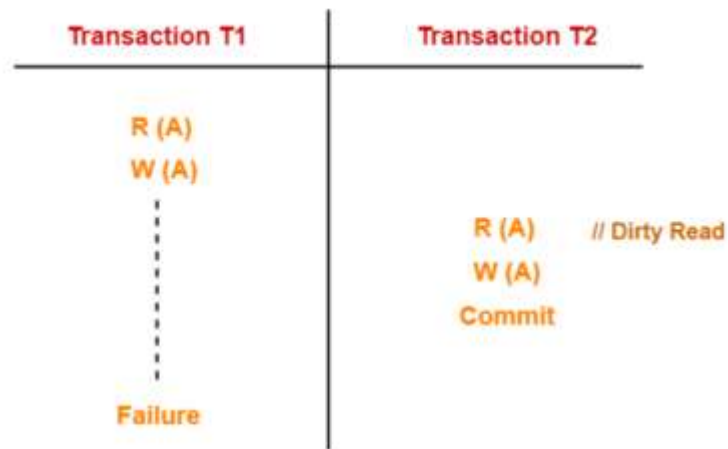
- It is possible to prevent the schedule of Figure section (c) from occurring by having all transactions lock before reading it.
- In Figure section (a) we see *serial* schedule, and in Figure section (b) is *serializable*, but not serial, schedule. Figure section (c) shows *non-serializable* schedule.
- Recall that we have *defined schedule to be serializable if its effect is equivalent to that of serial schedule*. However, it is not possible to test whether two schedules have the same effect for all initial values of the items, if arbitrary steps on the items are allowed, and there are an infinity of possible initial values. We assume that values cannot be the same unless they are produced by **exactly** the same *sequence* of steps.

Concurrency Control

- It is the management of two simultaneous transactions to execute without conflicts between each other (such as in multiuser systems).
- The simultaneous execution of transactions over shared databases can create several data integrity and consistency problems. How?
- What advantages of concurrency control you can think of?

Concurrency Possible Problems [Cases to avoid with schedules-Problems with schedules]:

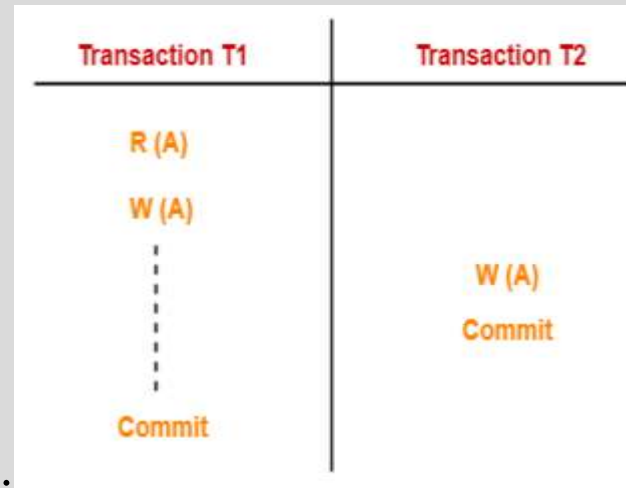
- Uncommitted Dependency or **dirty** read problem
- Inconsistent retrievals



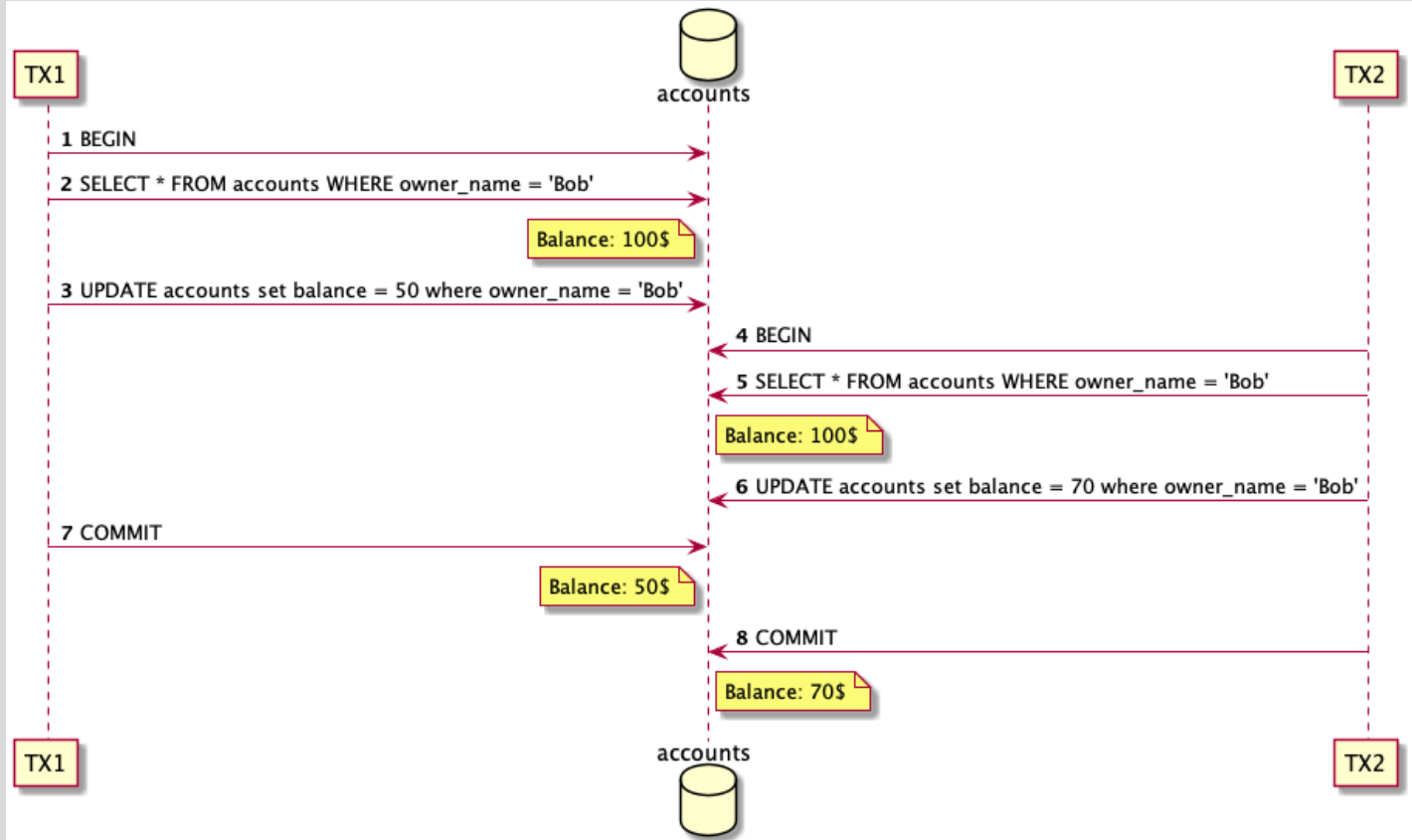
1. T1 reads the value of A.
2. T1 updates the value of A in the buffer.
3. T2 reads the value of A from the buffer.
4. T2 writes the updated the value of A.
5. T2 commits.
6. T1 fails in later stages and rolls back.

Lost Updates

- T1 reads the value of A (= 10 say).
- T2 updates the value to A (= 15 say) in the buffer.
- T2 does blind write A = 25 (write without read) in the buffer.
- T2 commits.
- When T1 commits, it writes A = 25 in the database.
- T1 over written value of A in the database.
- Thus, update from T2 gets lost.



Lost Updates



Unrepeatable Read

- This problem occurs when a transaction gets to read unrepeated
- T1 reads the value of X (= 10 say).
- T2 reads the value of X (= 10).
- T1 updates the value of X (from 10 to 15 say) in the buffer.
- T2 again reads the value of X (but = 15).
- T2 gets to read a different value of X in its second reading.
- T2 wonders how the value of X got changed because according to it, it is running in isolation.

Transaction T1	Transaction T2
R (X)	
	R (X)
W (X)	
	R (X) // Unrepeated Read

Phantom Read

- T1 reads X.
- T2 reads X.
- T1 deletes X.
- T2 tries reading X but does not find it.
- T2 finds that there does not exist any variable X when it tries reading X again.
- T2 wonders who deleted the variable X because according to it, it is running in isolation

Transaction T1	Transaction T2
R (X)	
Delete (X)	R (X)
	Read (X)

Ques: How can we avoid all those problems ?

Conclusion –possible solution

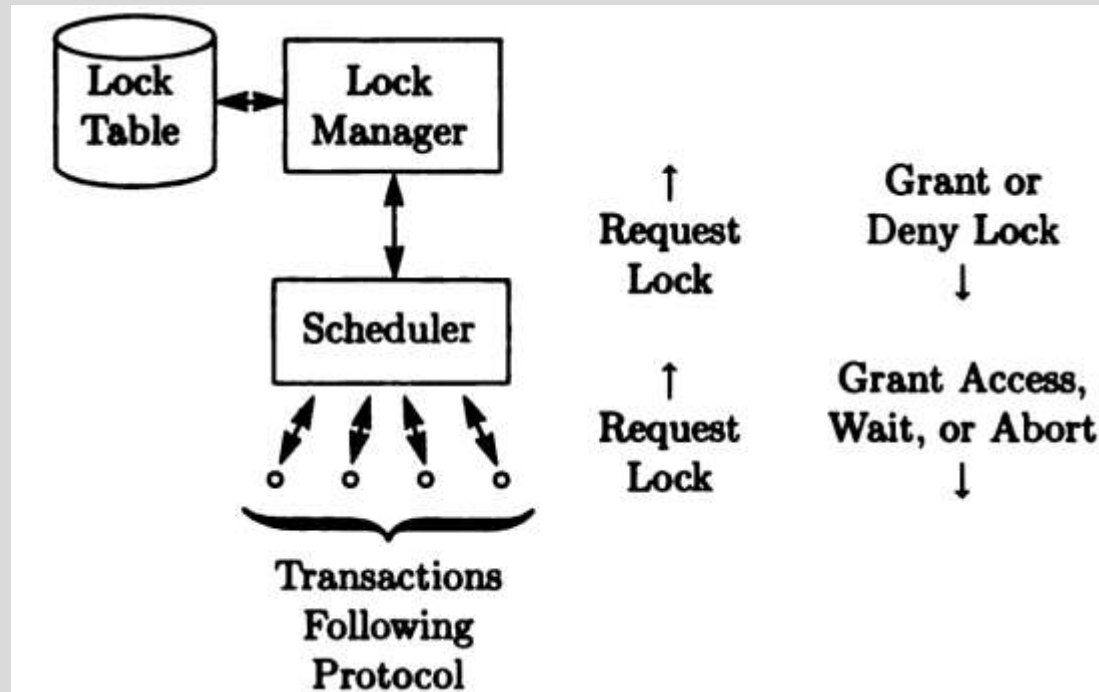
- Having a serial Schedule is the solution; where all the steps of each transaction occur consecutively.
- ***Are serial schedules practical solution? Why ? What are other possible efficient solutions?***

Concurrency Control Techniques – Scheduler

Lock: is access privilege to an item for a transaction, which the *lock manager* can grant or withhold from transaction. It stores the current locks in lock table, which consists of records ($\langle \text{data item} \rangle$, $\langle \text{lock type} \rangle$, $\langle \text{transaction} \rangle$)

- **Ques:** what is the meaning of record (I, L, T) ? \rightarrow transaction **T** has lock of type **L** on item **I**
- The lock manager finds locks on given items, insert lock records, and delete lock records, this will allow efficient management of locks.
- **Ques:** what is the key for lock records? Why ?

Concurrency Control -Relationship of the Lock Manager, Scheduler, and Protocol



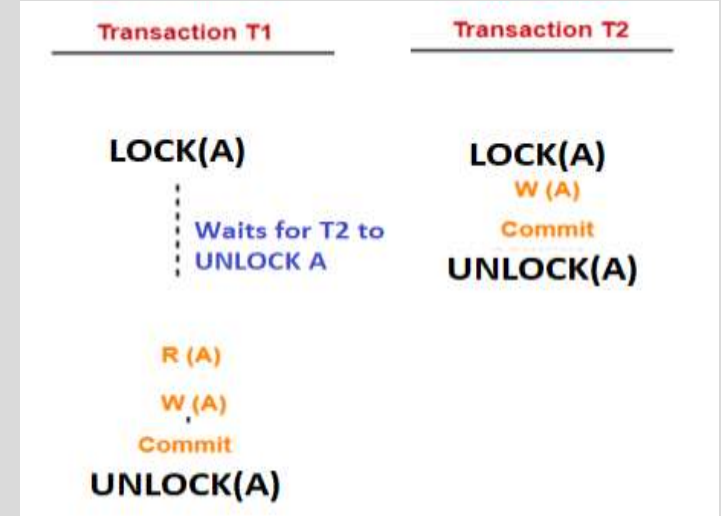
Concurrency Control Techniques

Locking : the transaction request to lock **item(s)** to perform some steps, after completion, the transaction releases the lock.

If an item was occupied when trying to lock it, the transaction will wait.

All locks must be released eventually.

• How a **lock** solve the lost update problem here?



Deadlock- Concurrent Processing

Suppose we have two transactions T1 and T2 whose significant actions, as far as concurrent processing is concerned are:

T1: LOCK A; LOCK B; UNLOCK A; UNLOCK B;

T2: LOCK B; LOCK A; UNLOCK B; UNLOCK A;

Discuss : Suppose **T1** and **T2** begin execution at about the same time. **T1** requests and is granted lock on **A**, and **T2** requests and is granted lock on **B**. Then **T1** requests lock on **B**, and is forced to wait because **T2** has lock on that item. Similarly, **T2** requests lock on and must wait for **T1** to unlock **A**.

Deadlock

Neither transaction can proceed; each is waiting for the other to unlock needed item, so both **T1** and **T2** wait forever.

- All activities come to a halt state and remain at a standstill.
- It will remain in a standstill until the DBMS detects the deadlock and **aborts** one of the transactions.

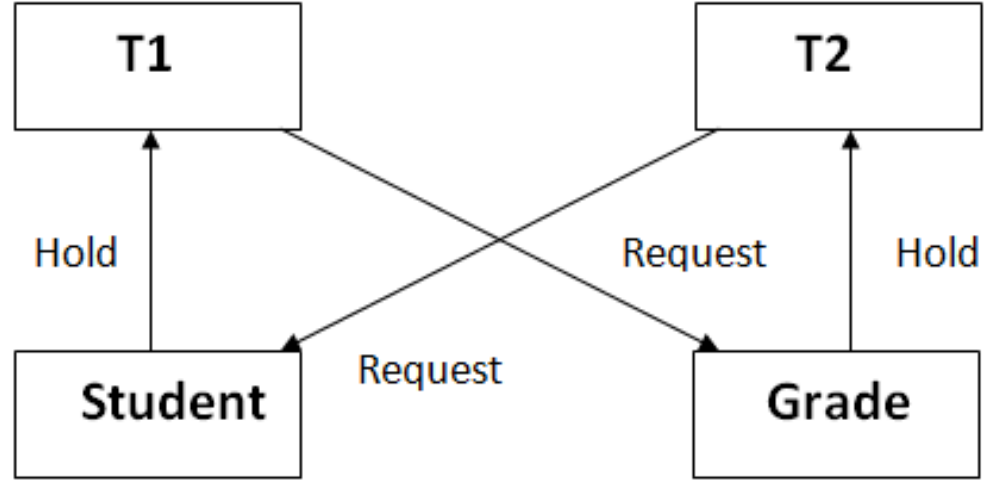


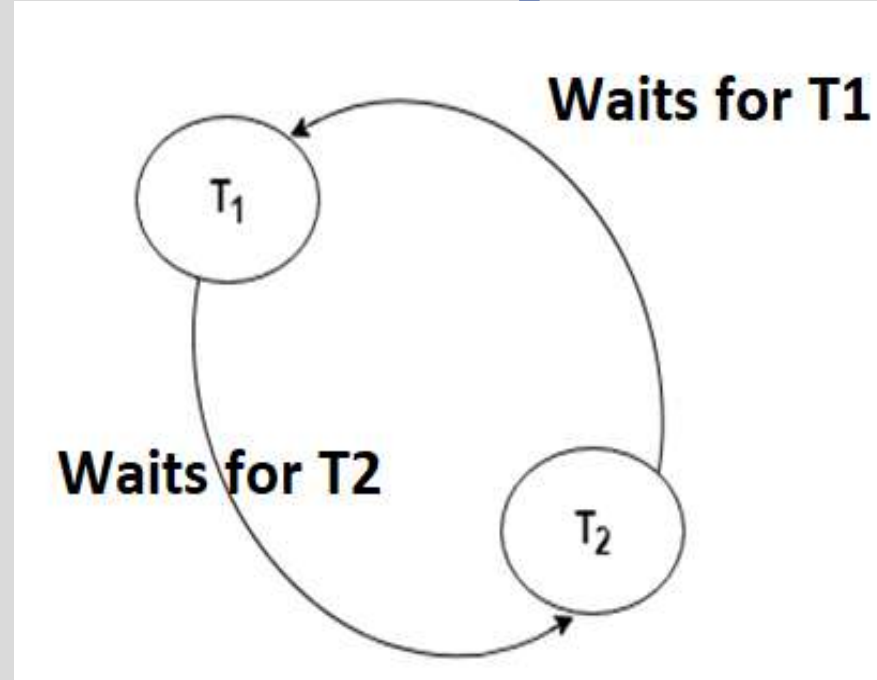
Figure: Deadlock in DBMS

Deadlock Avoidance/Detection- Wait for Graph

Transaction T1	Transaction T2
LOCK (A)	
	LOCK (B)
LOCK (B)	
	LOCK (A)

Deadlock Avoidance/Detection- Wait for Graph

Deadlock avoidance mechanism is used to detect any deadlock situation in advance. A method like "*wait for graph*" is used for detecting the deadlock situation but this method is suitable only for the smaller database.



The scheduler decides how to resolve this conflict , to release the deadlocks (by aborting a transaction for example)

Deadlock- Possible Solutions

- each transaction *requests all its locks at once*, and let the lock manager grant them all, if possible, or grant none and make the process wait, if one or more are held by another transaction.
- Assign an *arbitrary linear ordering to the items*, and require all transactions to request locks in this order.

Deadlock

Deadlock Prevention

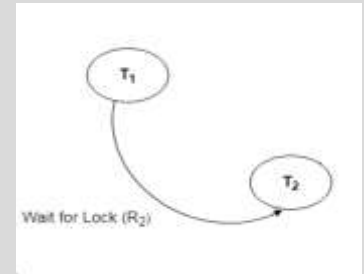
- Deadlock prevention method is suitable for a large database. If the resources are allocated in such a way that deadlock never occurs, then the deadlock can be prevented.
- The Database management system (DBMS) analyzes the steps of the transaction whether they can create a deadlock situation or not. If they do, then the DBMS never allowed that transaction to be executed.

Deadlock Ignorance

- Another approach to handling deadlocks is to do nothing to prevent them. By periodically examine the lock requests and see if there is deadlock.
- Why this might be beneficial as a solution?

Live lock- Concurrent Execution

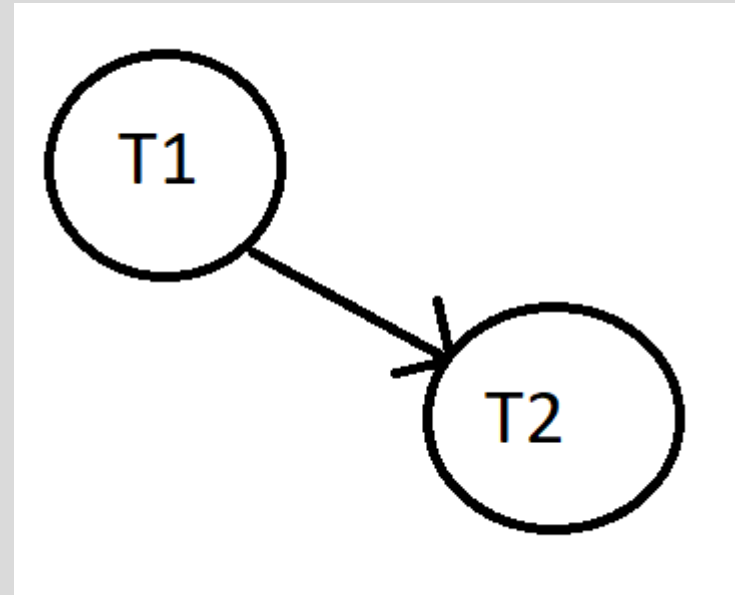
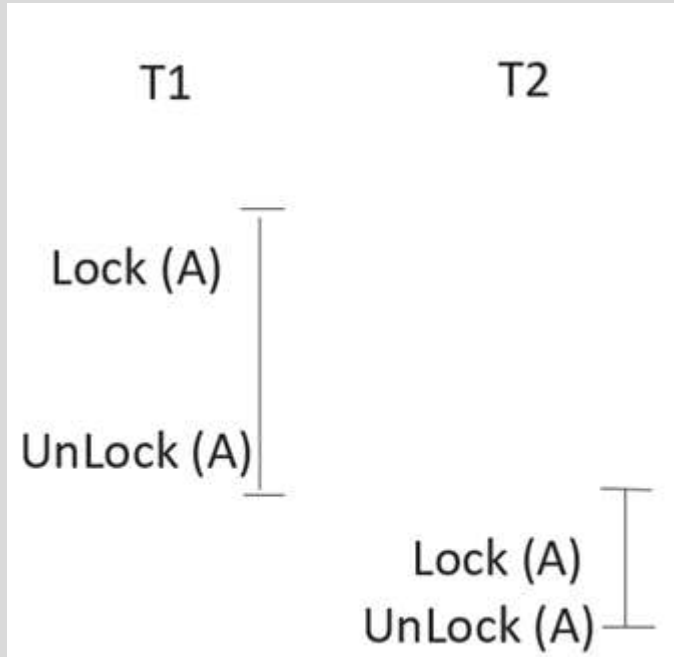
- The system that grants and enforces locks on items cannot behave unpredictably.
- **Scenario:** when **T1** released its lock on **A**, the lock was granted to **T2**. What if while **T2** was waiting, transaction **T3** also requested lock on **A**, and **T3** was granted the lock before **T2**? Then while **T3** had the lock on **A**, **T4** requested lock on **A**, which was granted after **T3** unlocked **A**, and so on. It is possible that **T2** could wait forever, since other transactions had lock on **A**, even though there are an unlimited number of times at which **T2** might have been given chance to lock **A**.
- Suggest Possible solutions for the live lock?



Transaction Models

- **Simple Transaction Model:** one type of lock , two transactions can not lock the same item(s)
- **Precedence graph:**
 - nodes represent the transactions.
 - Edges between $T1 \rightarrow T2$ IIF there is a lock that T2 is granted and the last transaction who released that lock was T1 . There is an item that T1 unlocked and the first transaction to lock right afterwards (for the same item) was T2.
- IIF Precedence is DAG (directed acyclic graph) \rightarrow the scheduling is serializable

Precedence Graph



Discussion

- How many programs are trying to modify record *A*. *what are the possible real-life problems in this structure*
- *Ques: What are the possible solution to prevent having two programs modifying the same record at the same time? Use the lock strategy we discussed earlier in the example below.*

A in database	5	5	5	5	6	6
T₁:	READ A		A:=A+1			WRITE A
T₂:		READ A		A:=A+1	WRITE A	
A in T₁'s workspace	5	5	6	6	6	6
A in T₂'s workspace		5	5	6	6	

How Locks Control Concurrency When Transactions Execute in Parallel

- Let us consider two transactions **T1** and **T2**. Each Trs. accesses an item **A**, which we assume has an integer value, and adds one to **A**. The two Trs. are executions of the program **P** defined by:
- P: READ A; A:=A+1; WRITE A;** [the value of **A** exists in the database. **P** reads **A** into its workspace, adds one to the value in the workspace, and writes the result into the database]

A in database	5	5	5	5	6	6
T₁:	READ A		A:=A+1			WRITE A
T₂:		READ A		A:=A+1	WRITE A	
A in T₁'s workspace	5	5	6	6	6	6
A in T₂'s workspace		5	5	6	6	

Possible Solution

- The most common solution to the problem represented is to provide lock on A . Before reading A , transaction $T1$ must lock A , which prevents another transaction from accessing until $T1$ is finished with A .
- The need for $T1$ to set lock on A prevents $T2$ from accessing A . $T2$ must wait until the other transaction unlocks A , which it should do only after finishing with A .

Remarks

- **Unlock command** is executed by the transaction holding the lock.
- **Lock Manager**: locks record for the transaction when it requests it for a specific item.
- Each transaction will **unlock** any item it locks, eventually.

THE TWO-PHASE LOCKING PROTOCOL

- We want to avoid having a schedule that is not serializable , we should follow some rules → two phase locking
- *Two-phase locking:*
 - requires that in any transaction, *all locks precede all unlocks*
 - each transaction releases a lock for the first time (the order of releasing is not important) before it receives the last lock.
- (*the point in between those two phases is called synchronization point → when the transaction has received all its requested locks before releasing the first lock*).

THE TWO-PHASE LOCKING PROTOCOL

Transactions obeying this protocol are said to be two-phase; the first phase is the **locking** phase, and the second is the **unlocking** phase.

The schedule that uses 2PL is serializable schedule

T1

Lock (A)

Lock(B)

→→ synchronization point →→

unlock(B)

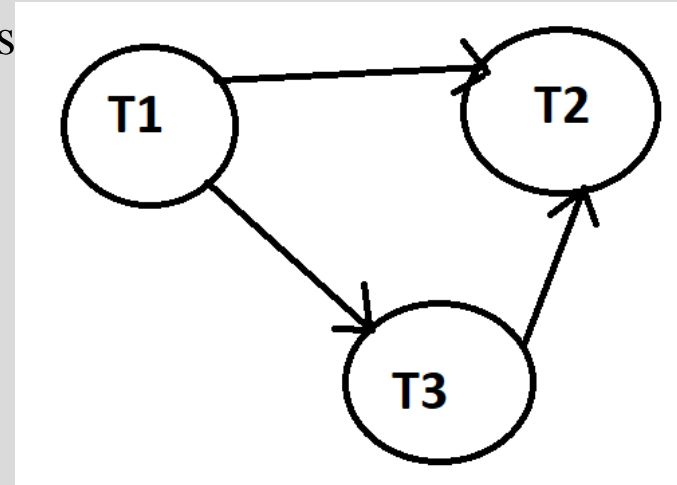
unlock(A)

Note: When the transaction unlocks an item in the second phase it can not lock ANY other item again

Serial equivalent from the precedent graph

- **Topological order** : T2 (it does not have a leaving edge) . T3, T1 == serial equivalent schedule of this graph → **T1, T3, T2**
- In case of 2PL (since there might be many serial equivalents) *ONE* (since there can be many) serial equivalent is going to be the order of the synchronization points

[ie. According to the order of synchronization points



Which transaction is *two-phase protocol* and which is not ?

LOCK A
LOCK B
UNLOCK A $f_1(A, B)$
UNLOCK B $f_2(A, B)$

T_1

LOCK B
LOCK C
UNLOCK B $f_3(B, C)$
LOCK A
UNLOCK C $f_4(A, B, C)$
UNLOCK A $f_5(A, B, C)$

T_2

Control Concurrency when Transactions Execute in Parallel

Types of Locks

- *Shared* Lock [Transaction can only **read** the data item values]
- *Exclusive* Lock [Used for both **read** and **write** data item values]

	RLock	WLock
RLock	✓	✗
WLock	✗	✗

Exclusive Lock [read/write lock Model]

Lock Based Protocol- Lock Manager

1. Shared lock:

- It is also known as a *Read-only lock*. The data item can only be read by the transaction.
- It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on that data item.

2. Exclusive lock:

- The data item can be both *read and written* by the transaction.
 - Multiple transactions do *not* modify the same data simultaneously.
- How can we apply this concept to flight tickets booking process?

How Does items Granularity Affect Performance with Locks?

- Choosing *large* granularity cuts down on the system *overhead* needed to maintain locks, since we need less space to store the locks, and we save time because fewer actions regarding locks need to be taken.
- *Small* granularity allows many transactions to operate in *parallel*, since transactions are then less likely to want locks on the same items.

Brain Teaser

```
UPDATE Customers  
SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'  
WHERE CustomerID = 1;
```

```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate  
FROM Orders INNER JOIN Customers ON  
Orders.CustomerID=Customers.CustomerID;
```

- How would you choose the **items** in those two cases and why?

Hint: Consider best performance

Granularity

- ***Rule of thumb***: proper choice for the size of an item is such that the average transaction accesses few items.
- ***Assumption for simplicity***: we shall assume that when part of an item is modified, the whole item is modified

Discuss

- if the typical transaction (in relational system) reads or modifies one tuple, which it finds via an index, what would be the proper item?
- If the typical transaction takes join of two or more relations, and thereby requires access to all the tuples of these relations, which item would you choose?

Conclusion: To Achieve More Concurrency While Guaranteeing Correctness. What must be done?

every schedule of every collection of transactions is allowed if its effect happens to be equivalent to some serial schedule and forbidden otherwise.

Is Abort Desirable or Not ? Why!

- Forcing many transactions to wait for long periods may cause too many locks (waiting transactions might already have some locks).
- deadlock causes many transactions to delay so long that the effect becomes noticeable for end users. (Ex: the user standing at an ATM machine).
- when deadlocks are inevitable, the scheduler has *no choice* but to **abort** at least one of the transactions involved in the deadlock.

Possible States of a Transaction Explained

1. Failed state

- If any of the checks made by the *database recovery system* fails, then the transaction is said to be in the failed state.
- Example: total mark calculation, can you explain how this might apply?

2. Aborted

- If any of the checks fail and the transaction has reached a failed state then the database recovery system will revert to its previous consistent state [it will **abort** or **roll back** the transaction].
- If a transaction fails in the *middle*, then *all* the executed operations are rolled back.

Possible States of a Transaction

- After aborting the transaction, the database recovery module will select one of the two operations:
 1. Re-start the transaction
 2. Kill the transaction

3. Active state

- The active state is the first state of every transaction (the transaction is being executed).
- For example: Insertion or deletion or updating a record is done in the active state **BUT** all the records are still not *saved* (committed) to the database.

Possible States of a Transaction

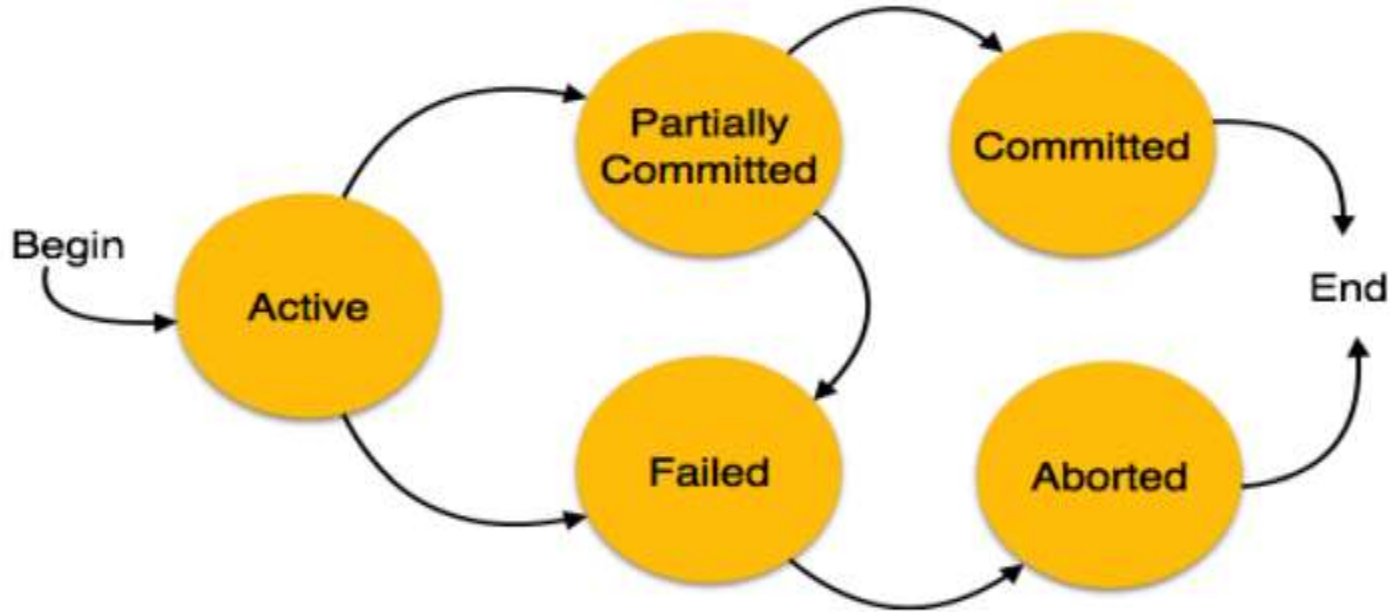
4. Partially Committed

- a transaction executes its final step, but the data is still not saved to the database.
- Ex : total mark calculation, a final display of the total marks step is executed in this state.

5. Committed

- if the transaction executes all its steps successfully. In this state, all the effects are now *permanently* saved in the database.

Possible States of a Transaction Explained



Databases



Transaction Management

Second Lecture

Levente Erős – eros@db.bme.hu

Ruba AlMahasneh – mahasnehr@gsuite.tmit.bme.hu



Agenda

1. Read/Write Precedence Graph
2. Locks on Hierarchal structure –Tree Protocol
 - a. Tree Protocol
 - b. Warning Protocol
3. Scheduling with time stamps Protocol
4. When to abort a transaction?
5. Time Stamps Transaction Handling

R/W Lock Model

R/W Lock Model

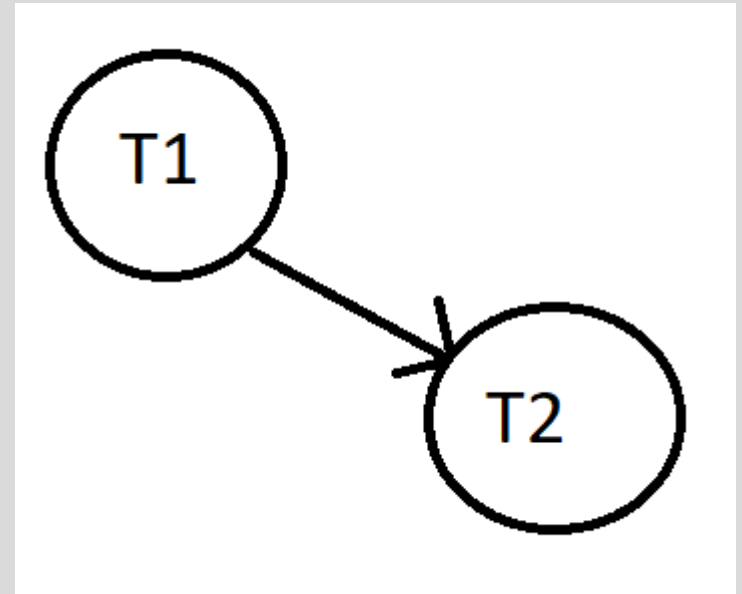
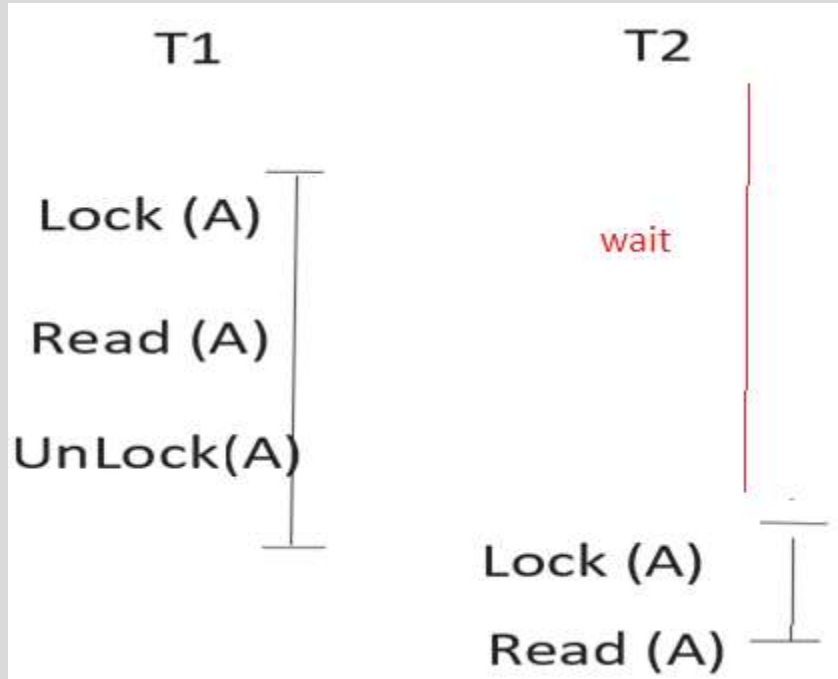
- Lock → Read Lock (RLock)
→ Write Lock (WLock)

Goal: Less restrictive lock management
and higher performance

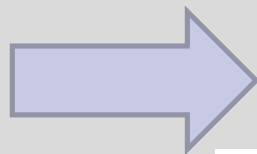
	RLock	WLock
RLock	✓	✗
WLock	✗	✗

Compatibility Matrix

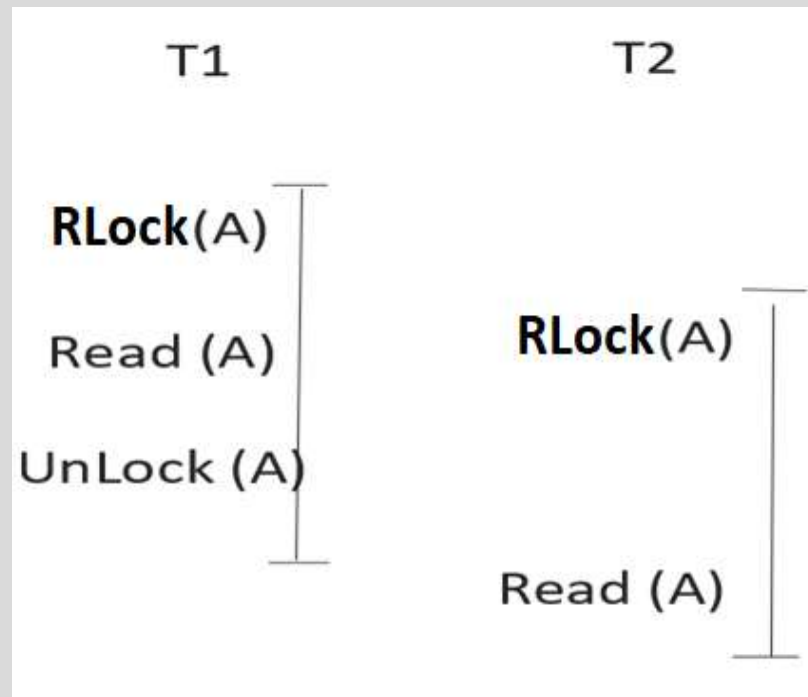
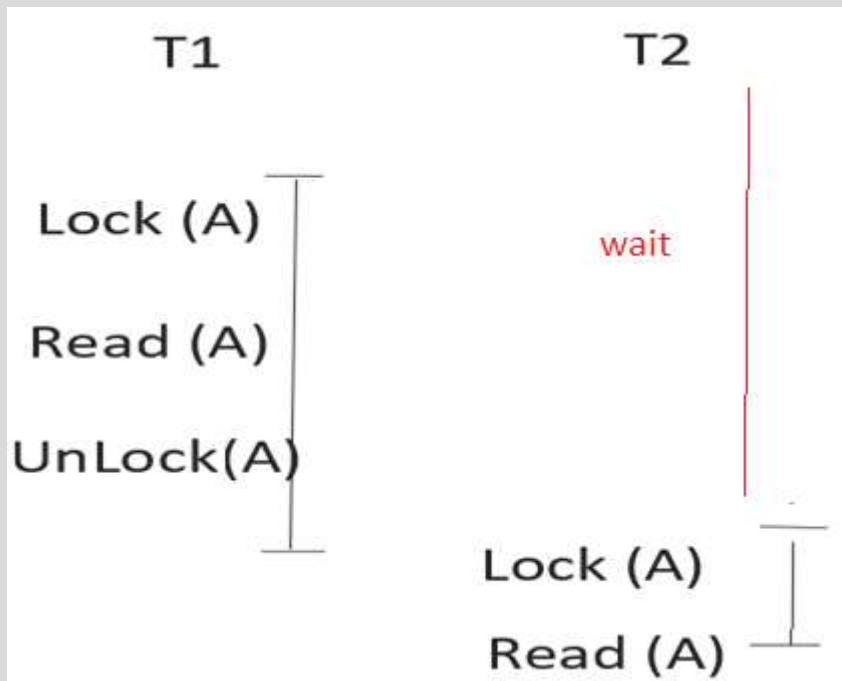
Simple Lock Model



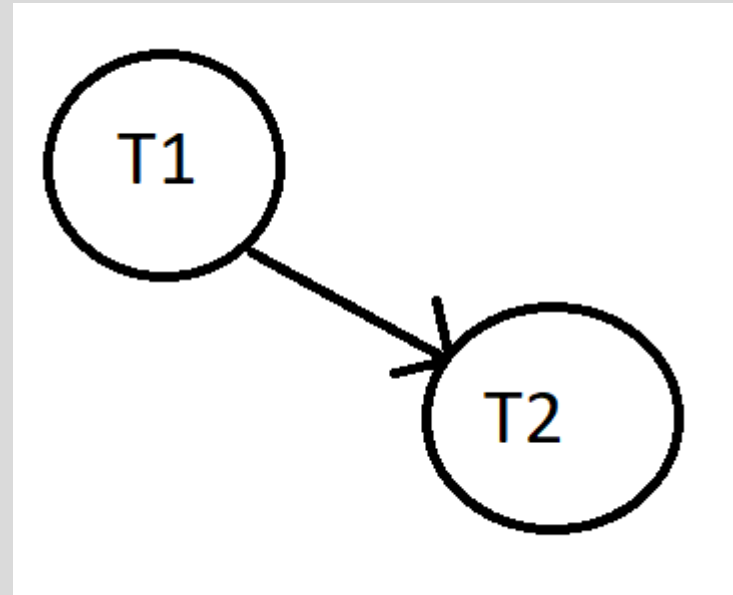
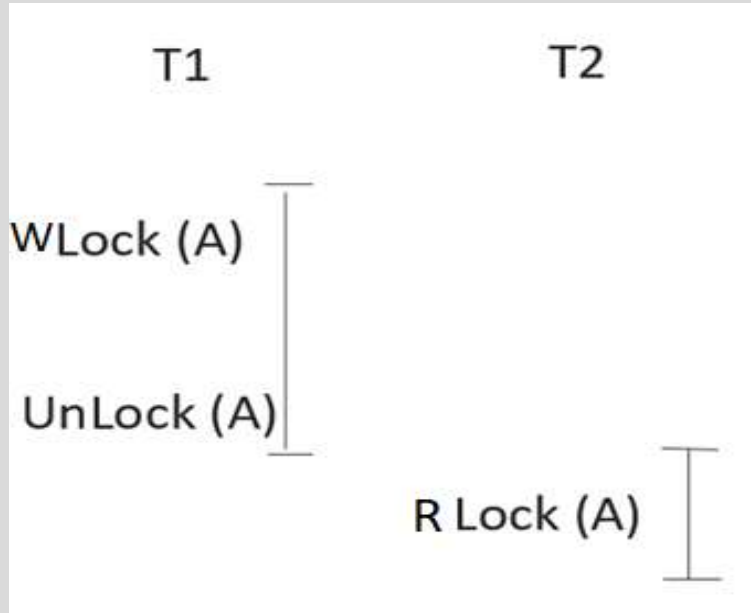
Simple Lock Model



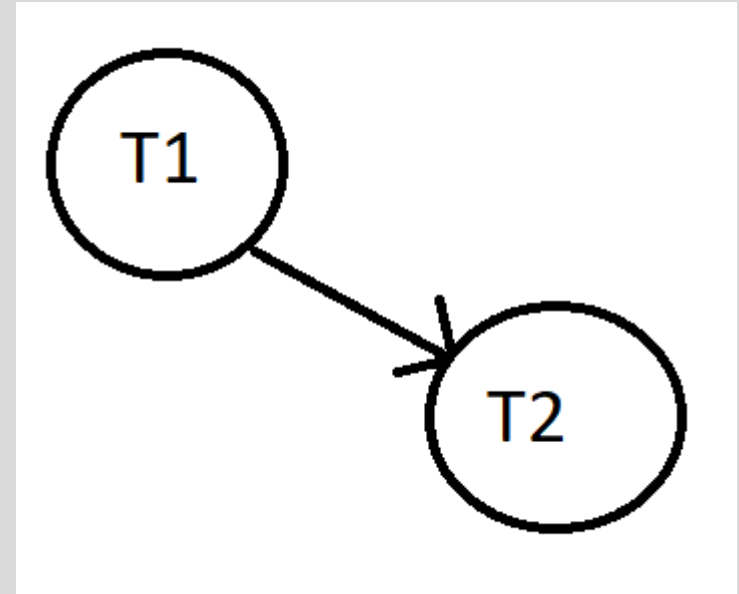
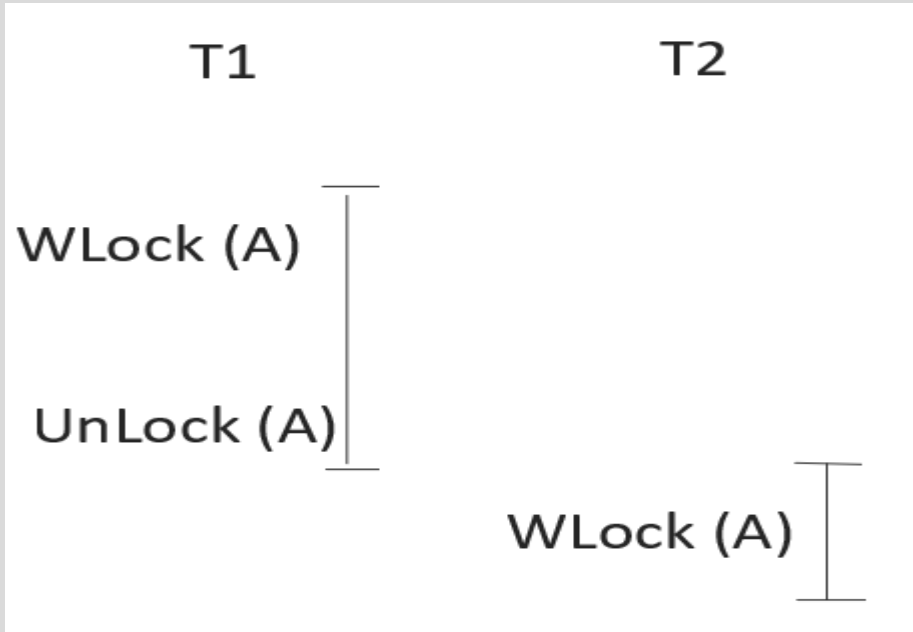
R/W Lock Model



Read/Write Precedence Graph

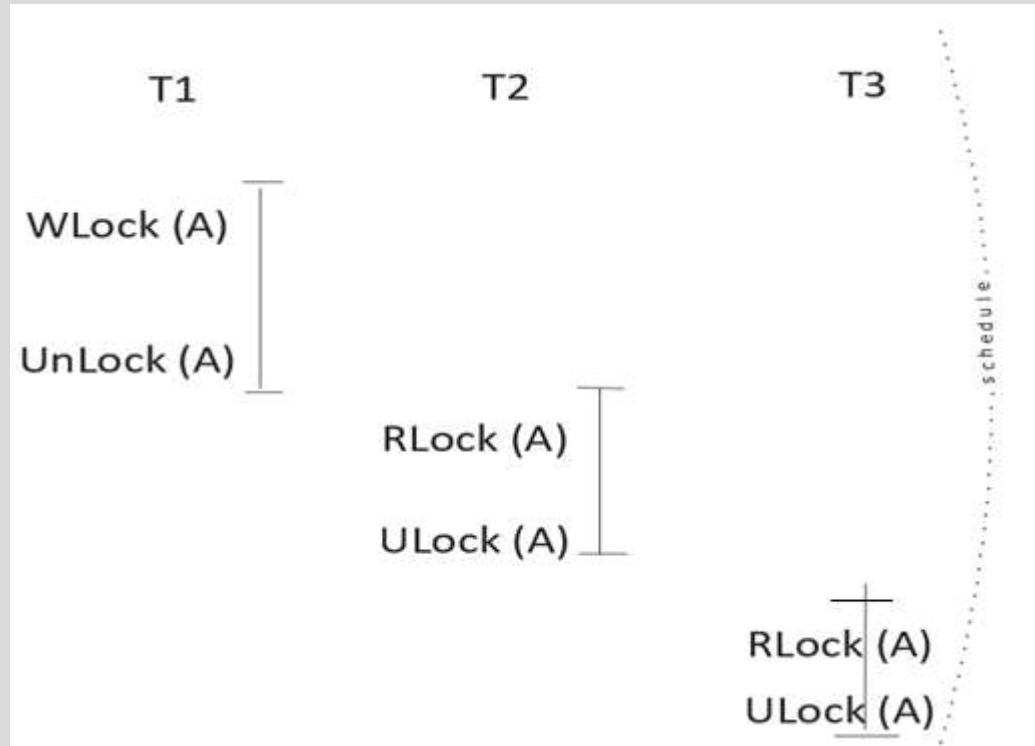


Read/Write Precedence Graph

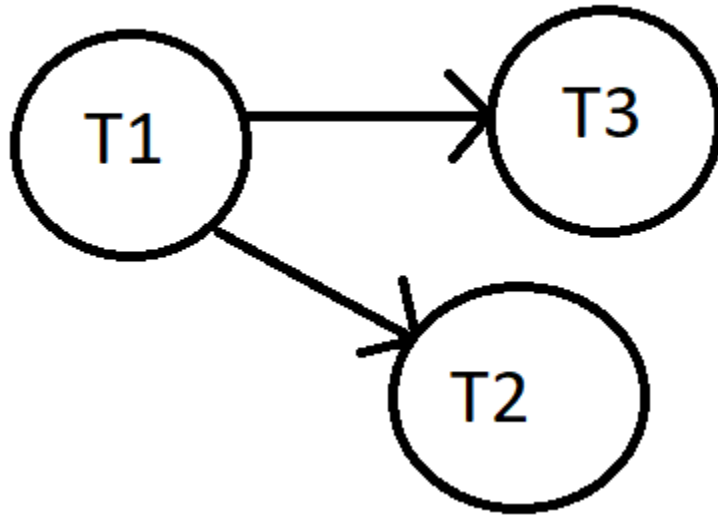


Ques: What is the precedence graph if we have two read locks requested ?

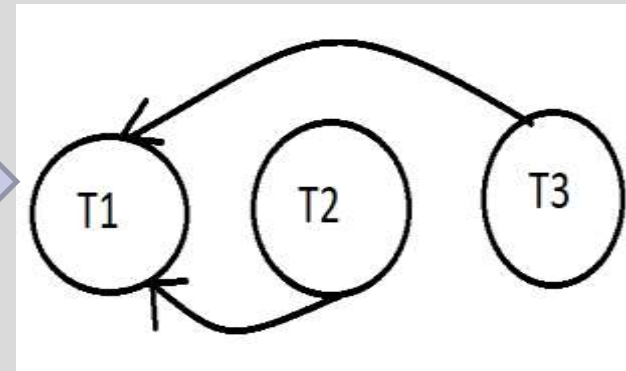
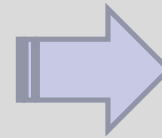
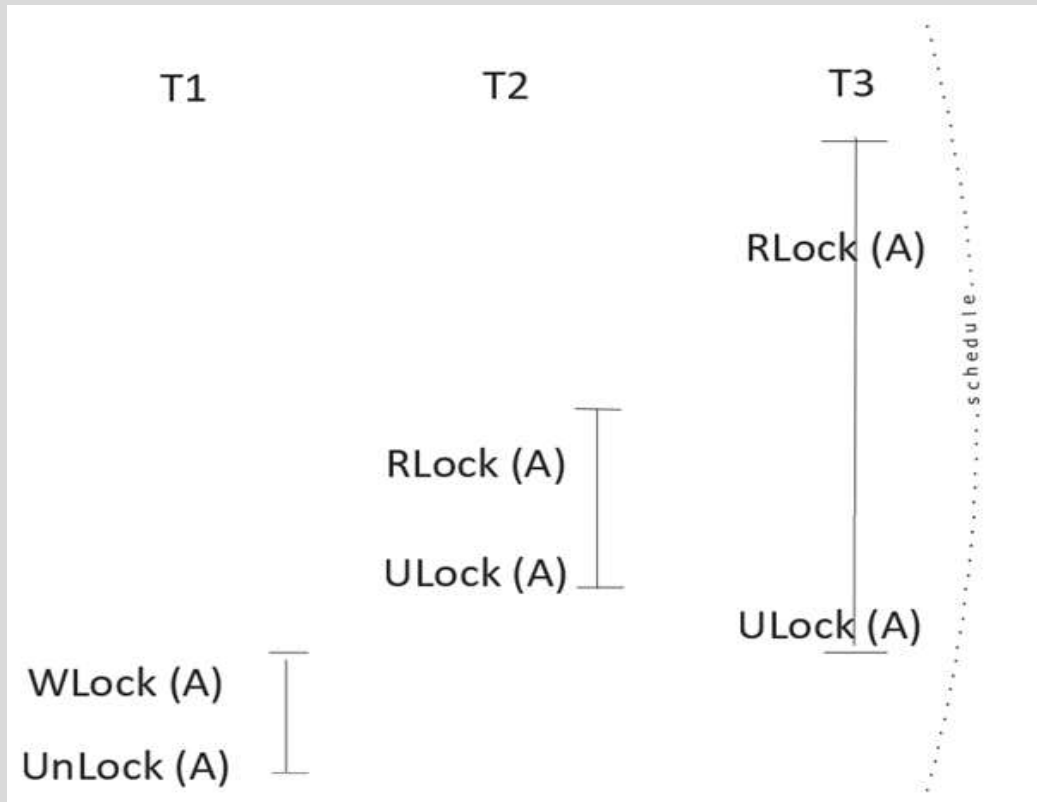
Read/Write Precedence Graph




Precedence Graph-Serial Equivalent



Read/Write Precedence Graph



Precedence Graph's Rules

- If there is no directed cycle graph  then our schedule is serializable and it has serial equivalent schedule.
- Read locks order is irrelevant since it does not affect the data itself and no changes committed to the data
- No edges between Read locks why?
- **Remark:** **Unlock** is source of the arrow , **W/R** process is the destination of the arrow (edge) in the graph.

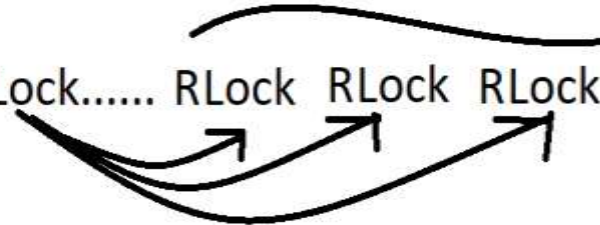
Precedence Graph's Rules

WLock, ULock.....WLock



RLock Series

WLock, ULock..... RLock RLock RLock



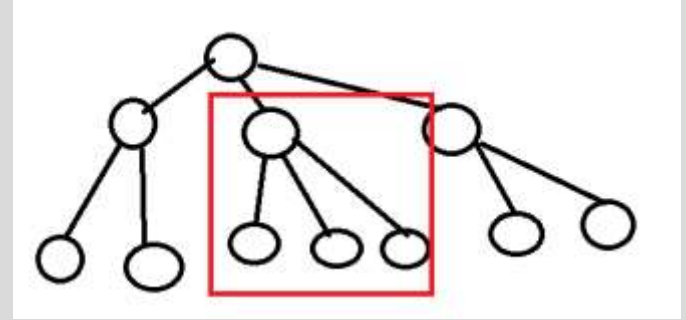
RLock Series

RLock, ULock RLock, ULock WLock



Locks on Hierarchical structure –Tree Protocol

- **Goal** : Locking some parts of the tree
- **Protocols** :
 - **Tree Protocol** : uses LOCK on a single node.
 - Serializable
 - No deadlocks
 - **Warning Protocol** : Uses WARN or LOCK on a single node

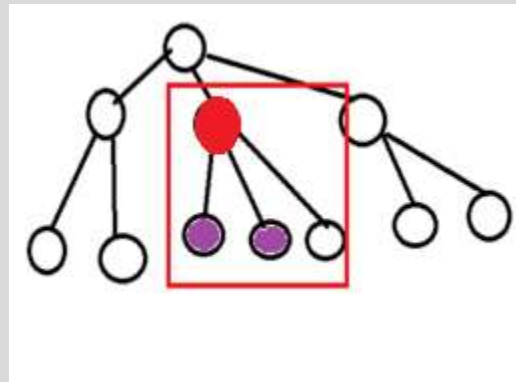


Locks on Hierarchical structure –Tree Protocol

- Rules we must follow :
 1. **First Lock** can be put on any single node
 2. Further Locks must be children of a **Locked** node
 3. One node can not be **relocked** (each node can be only locked once)

Remarks :

1. You are only allowed to access the Locked node [access privilege]
2. In Tree protocol, no *dead locks* occur
3. Tree protocol produces Serializable Schedule

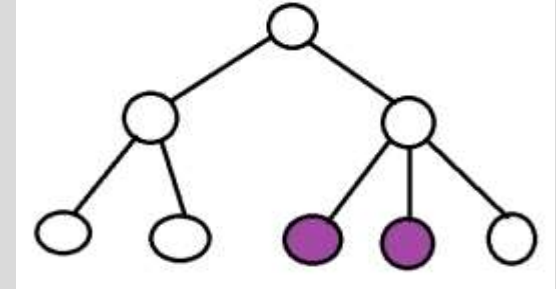






Locks on Hierarchical structure –Warning Protocol

- Uses Warns and Locks
- Locking a node is access to the **whole** sub Tree (R,W)

Rules:

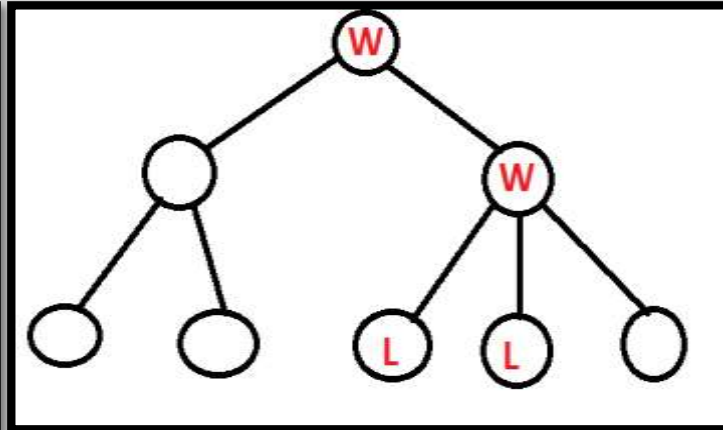
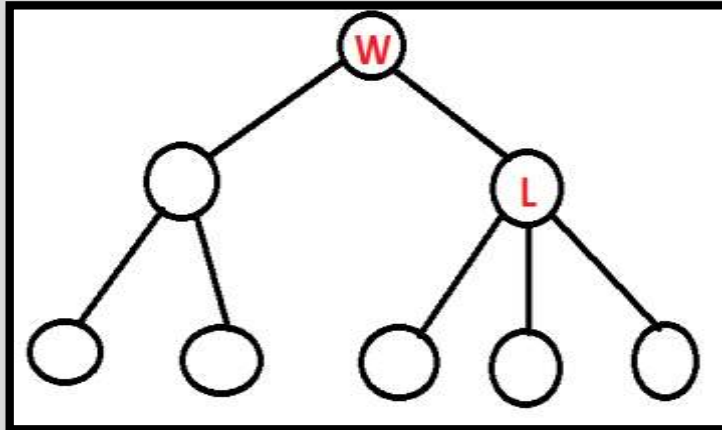
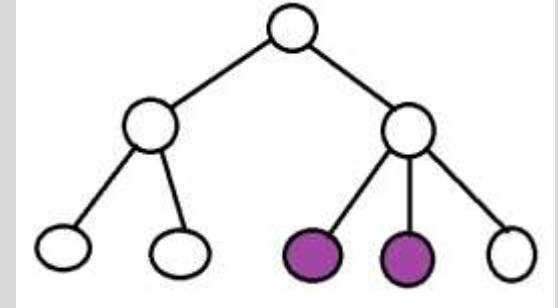
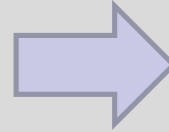
1. Lock or Warn the **root** node
2. You can only **Lock** or **Warn** a node if there is a **Warn** on the parent
3. *2 Phase locking must be used*, meaning after the first *unlock* you can not place any warns or locks anywhere at the tree [crucial for schedule serializability]



	Lock	Warn
Lock		
Warn		

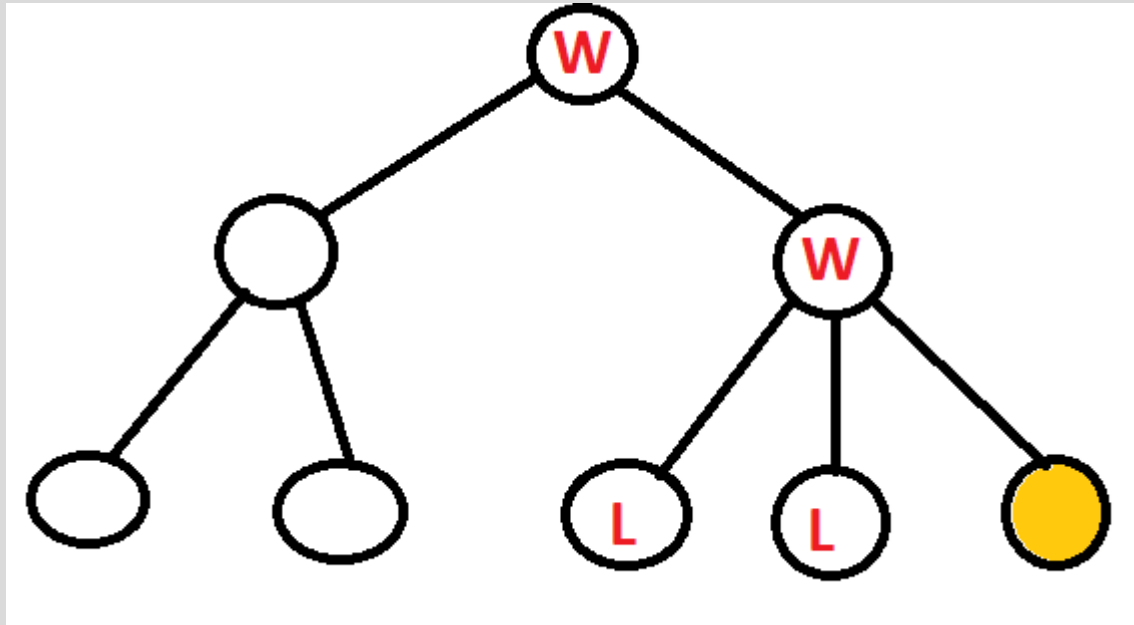
Locks on Hierarchical structure –Warning Protocol

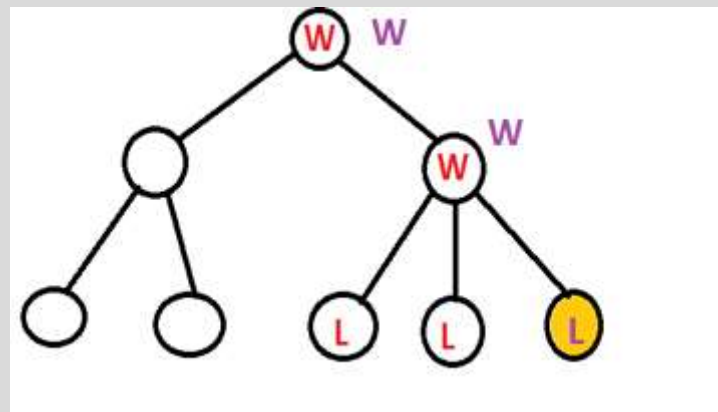
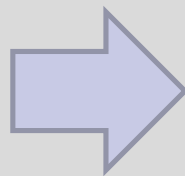
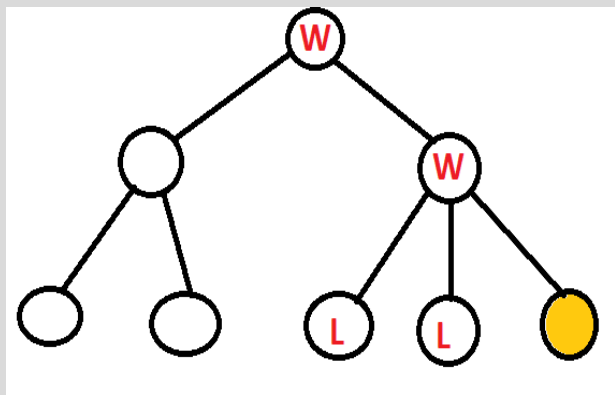
Quest: How can I perform Warn/Lock structure on the below Tree
If I want to access the colored children ?



Locks on Hierarchical structure –Warning Protocol

Can you think on how *another* transaction can lock the colored node?





Scheduling with timestamps Protocol

- The transaction don't overlap and influence each others final results , they don't collide too much.
- **Timestamping**: for each transaction a unique identifier (time stamp) created by DBMS that indicates relative starting time of a transaction $t(T_i)=i$
- Allocate time stamp for each transaction , it grows as the transaction proceed (timestamps are *proportional to the start time*)
- Timestamps (Read/Write time stamps) are allocated for each **data item**.
 - **Read timestamp** for each data item [*Timestamp of the transaction that read the data item most recently*] = $R(A)$
 - **Write timestamp** for each data item [*Timestamp of the transaction that Wrote the data item most recently*] = $W(A)$

Scheduling with time stamps Protocol

- **Ultimate Goal**

- ❑ Create a *predefine* serial equivalent with W, R operations with all (unique) time stamps of all transactions, so if there was any transaction requested a read or write operation that **is against this serial equivalent**
→ **will be aborted.**

Example :

serial equivalent of transactions T1,T2,T3..... → $t(T1) < t(T2) < t(T3)$

[growing]

This serial equivalent will be enforced otherwise anything that does not conform will be **aborted**

When to abort a transaction ?

- If any of the checks fail and the transaction has reached a failed state then the database *recovery system* will revert to its previous consistent state.
- If the transaction fails in the *middle* of the transaction, then *all* the executed transactions are rolled back to its consistent state.
- *After aborting* the transaction, the database recovery module will select one of the two operations:
 1. Re-start the transaction
 2. Kill the transaction

Time Stamps Transaction Handling

Action : Transaction T reads or Writes Data Item A. WE REPRESENT $t(T)$

A	T tries reads	T tries writes
$R(A) \leq t(T)$ $W(A) \leq t(T)$	OK $R(A) := t(T)$	OK $W(T) := t(T)$
$R(A) > t(T)$ $W(A) \leq t(T)$	OK $R(A) := R(A)$ REMAINS	ABORT, because it does not conform with the serialized schedule TS
$R(A) > t(T)$ $W(A) > t(T)$	ABORT VALUE DOES NOT EXIST	ABORT, because it does not conform with the serialized schedule TS
$R(A) \leq t(T)$ $W(A) > t(T)$ $R(A) \leq t(T) < W(A)$	ABORT VALUE DOES NOT EXIST	Either abort OR write, but the value will not updates (it will be rewritten), <i>not abort and not write...</i> Thomas writing rule

Transaction Management



Third Lecture

Levente Erős – eros@db.bme.hu

Ruba AlMahasneh – mahasnehr@gsuite.tmit.bme.hu

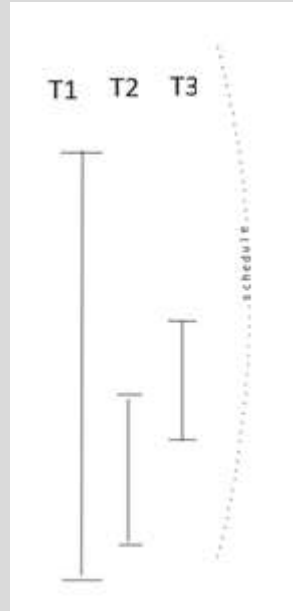


Agenda

- **Scheduling Transactions With Time Stamps**
- **Version Handling/Management Time Stamp Scheduling**
- **Handling Transactions Errors**
 - **Possible causes and solutions**
 - **Time stamps and strictness**
 - **Handling OS and Media Errors –Journaling**
- **Redo Recovery Protocol Procedure**

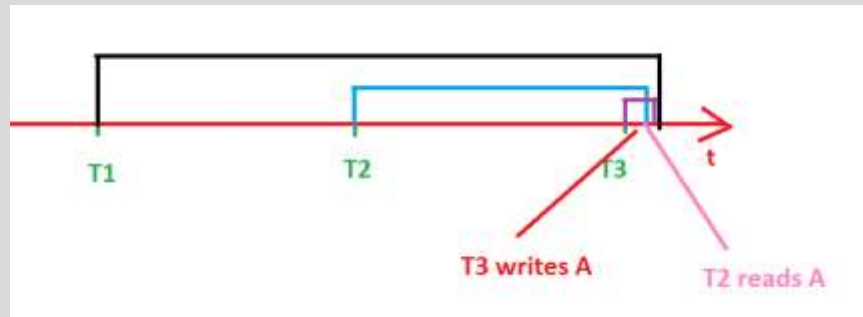
Scheduling Transactions With Timestamps

- **GOAL:** to build the serial equivalent schedule using time stamps for the transactions [we are not using the lock concept]
- Every transaction has a unique time stamp. We are forcing serial equivalent schedule where Transactions are executed in near zero amount of time .
- There will be overlapping among transactions, thus if they do not follow the serial equivalent they must be aborted.



Version Handling/Management Timestamp Scheduling

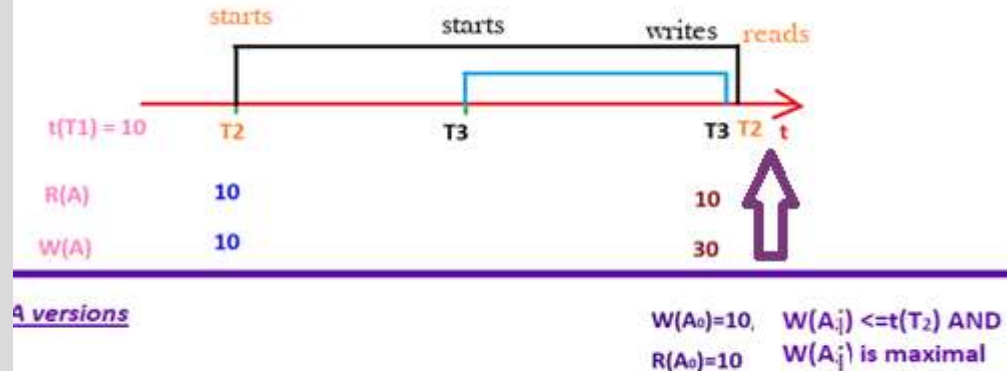
- When a data item is written, the original one is not overwritten but a new version of it is created.
- The time stamp is not changed but a new time stamp is created for this transaction.
- **Ques:** What shall we do at T2 where it wants to read A?



Version Handling/Management Time Stamp Scheduling

- Different versions serve as logs and history tracking record for the data items and the operations that are done on them and when.
- How does Timestamp scheduling avoid the two aborts we explained earlier?

Ques: When we want to perform the read operation of $T2 \Rightarrow R(A)$ what shall we do to guarantee we are getting the latest value of item A?



Handling Transactions Errors

- **Transaction Error:** is whatever cause a transaction not to terminate is considered a transaction error
- **Possible Causes:**
 1. Deadlock Resolution → Abort
 2. Ensuring serializability of the schedule → Abort
 3. Arithmetic/program error
 4. OS Error
 5. Media Failure

Handling Transactions Errors

- **Consistent state:** the state of the DB only contains successful (fully executed) transactions.
- The history of DB made from series of consistent states (one after another).
- How can we make sure the DB changes from one consistence place to another ?
- *We eliminate the possibility of the first three transaction errors to ensure consistency of the DB states*
- Find the commit points. What is a commit point?

Handling Transactions Errors-Strict 2PL

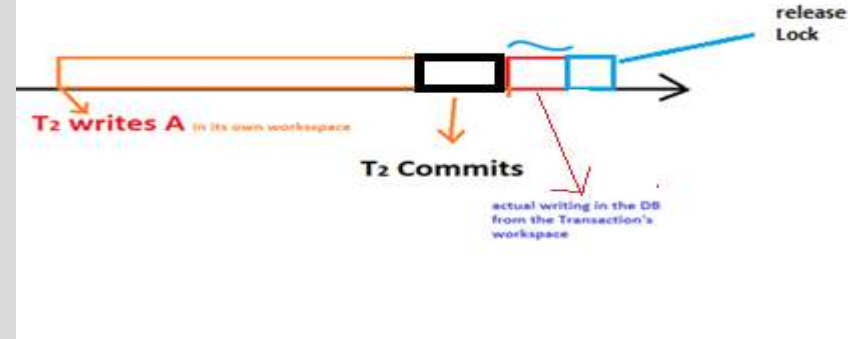
- Each transaction has work place, it performs all the actions there then it reaches the commit point when those operations are moved from the work space to the actual public DB space [then the first 3 errors can not affect the DB, why ? *transaction can not fail anymore*]
- How does that help in keeping consistent DB state?
- **Conclusion:** everything is dirty data before the commit point .

Strict 2PL

- **Strict two phase locking :**

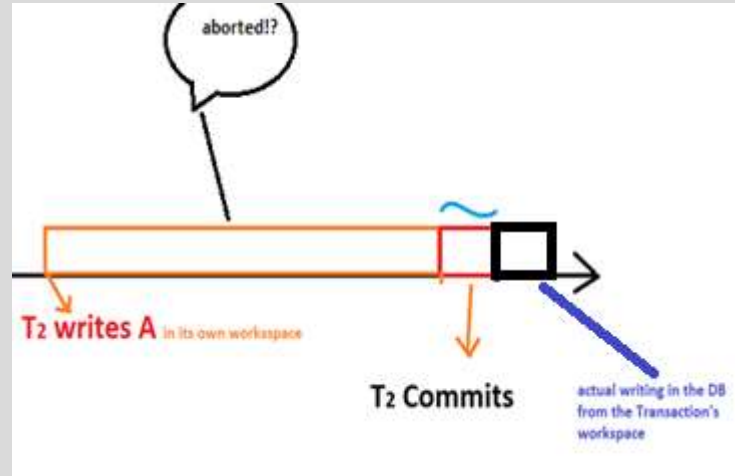
A transaction follows strict 2PL **IIF**:

- 2PL
 - It only writes in the DB after the transaction has reached its *commit point*
- **Cascading abort:** A situation in which the abort of one transaction forces the abort of another transaction to prevent the second transaction from reading invalid (uncommitted) data.
 - No Dirty Data → no Cascading aborts
 - All those safety measures will affect the performance of the operations (no free lunch concept)



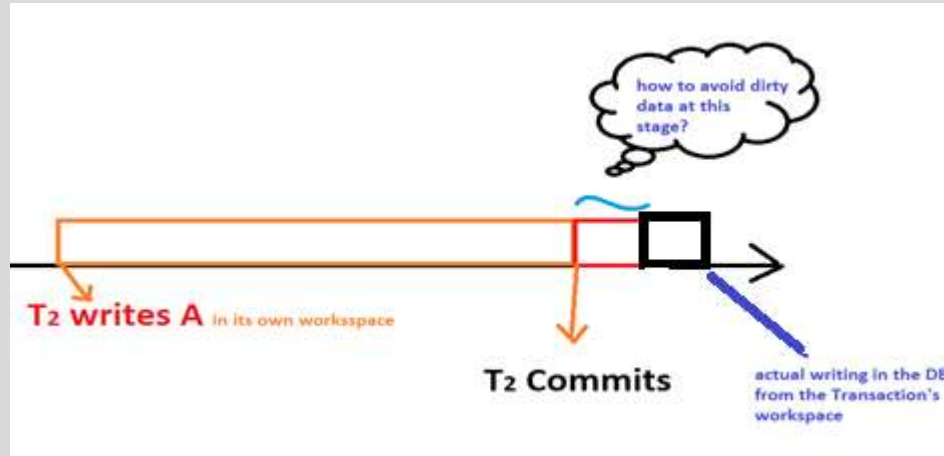
Time stamps and strictness

- **Problem** : time stamps scheduling is optimal in case there are not overlapping transactions, and it uses aborts when needed.



Time stamps and strictness

- The time stamps are placed on different points (commit point, actual writing on DB) which can be problematic

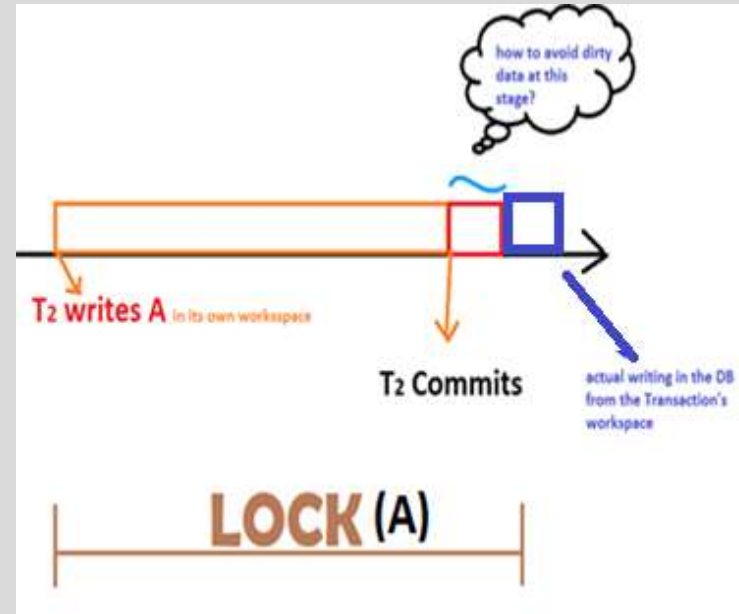


Time stamps and strictness

- **Solutions**

1- We introduce **lock** with the **time stamp** scheduling between the beginning of the transaction and the actual writing in the DB from the local space

2- We accept that there will be dirty data if we do not want to compromise the desired performance.



How does that affect the processing performance?

Aggressive Vs Conservative Protocols

1. Conservative Protocols :

- It is also known as *Static 2-PL*.
- This protocol requires the transaction to lock **all** the items it access before the transaction begins execution by pre-declaring its read-set and write-set.
- If any of the pre-declared items needed cannot be locked, the transaction does not lock any of the items, instead, it waits until all the items are available for locking.
- *Conservative 2PL is deadlock-free.*

Aggressive Vs Conservative Protocols

2. Aggressive Protocols (puts performance before avoiding aborts) :

- Requests a lock on an item immediately before reading or writing the item. If an item is to be written after reading, the read-lock is taken first and upgraded to a write-lock when needed.
- Can this lead to deadlocks ?
- Which has better performance Aggressive Vs Conservative ?

Aggressive Protocols -

Example: Timestamp scheduling, we allow the transactions to run, which doesn't even lock and transactions don't wait just perform operations and if a given operation is not allowed, abort.

Difference between Conservative and Strict/Aggressive 2-PL :

Conservative 2-PL

Strict 2-PL

In Conservative 2-PL, A transaction has to acquire locks on all the data items it requires before the transaction begins its execution.	In Strict 2-PL, A transaction can acquire locks on data items whenever it requires (only in growing phase) during its execution.
It does not have growing phase.	It has growing phase.
It has shrinking phase.	It has partial shrinking phase.
It ensures that the schedule generated would be Serializable and Deadlock-Free.	It ensures that the schedule generated would be Serializable, Recoverable and Cascadeless.
It does not ensure Recoverable and Cascadeless schedule.	It does not ensure Deadlock-Free schedule.

Difference between Conservative and Strict 2-PL :

Conservative 2-PL

Strict 2-PL

It does not ensure Strict Schedule.	It ensures that the schedule generated would be Strict.
It is less popular as compared to Strict 2-PL.	It is the most popular variation of 2-PL.
It is not used in practice.	It is the most popular variation of 2-PL
In Conservative 2-PL, a transaction can read a value of uncommitted transaction.	In Strict 2-PL, a transaction only reads value of committed transaction.

Handling OS and Media Errors –Journaling

- Journaling guarantees that all changes are on disk before a transaction is marked committed as long as O/S and hardware caching are disabled.
- **Journaling (Log) for any transaction:**
 - (T, Begin)
 - (T, Commit)
 - (T, Abort)
 - Write operation \rightarrow (T, A's old value + A's New value)

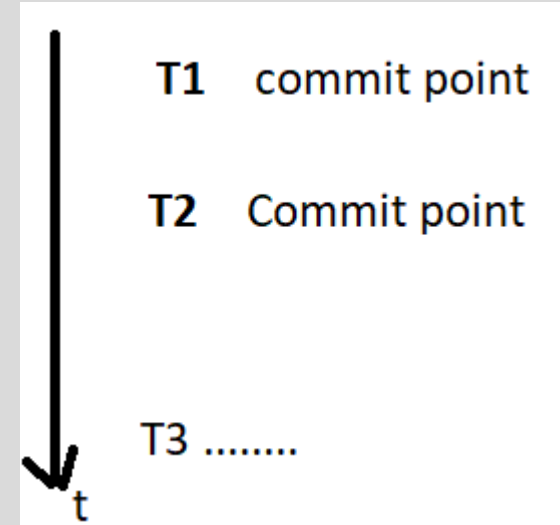
Handling OS and Media Errors –Journaling

- A Journal is written to mass storage before DB is written, except for *abort* (which is first performed then the logging of the event happens)
- **Redo Protocol is used:** we will only store the *new value* of the transaction that has been written.
- Redo Protocol is strict 2PL protocol [steps]:
 - (T, Begin)
 - (T, A, New value)
 - (T, Commit)
 - Move Journal → Mass storage in DB
 - If any Modification in RAM → Mass storage [HDD]
 - Releasing Locks

Journaling Performance ?

What if restoring a consistent state required going very far back in the logs?

To save changed pages in the database cache to the hard disk, we set up **journaling checkpoints** to occur automatically. After the checkpoint has been reached, the data in the journal file is no longer needed, so the file can be reused.



Why We Need Journaling ?

- If the OS can not run for some reason (crash) we must set (restore) the DB to a consistent state (the committed transactions). Which will be the latest /most recent one logged in the journal.

Why we choose the most recent version ?

Check point : artificial consistent state, where you can start a new transaction == Latest consistent state

Redo Recovery Protocol Procedure

- **Redo Recovery Steps:**
 - Release locks
 - Finding committed transactions from the last *consistent state* = ***check point***
 - Re-play [rewriting to the DB] committed transactions based on the journal