



Mansoura University
Faculty of Computers and Information
Department of Information System



[IS313P] Database System II

Grade: 3 rd. IS & IT

Dr. Amira Rezk



Concurrency Control Techniques



AGENDA



INTRODUCTION

- Why Concurrency Control Is Needed
 - The Lost Update Problem
 - The Temporary Update (or Dirty Read) Problem
 - The Incorrect Summary Problem
- Purpose of Concurrency Control
 - To enforce Isolation (through mutual exclusion) among conflicting transactions.
 - To preserve database consistency through consistency preserving execution of transactions.
 - To resolve read-write and write-write conflicts.



INTRODUCTION

- Concurrency Control Techniques
- Pessimistic Concurrency Control (LOCK)
 - Assumes that conflicts will happen
 - Detect conflicts as soon as they occur and
 - Resolve them using locking.
- Optimistic Concurrency Control
 - Assumes that conflicts between transactions are rare.
 - Does not require locking
 - Transaction executed without restrictions
 - Check for conflicts just before commit



PESSIMISTIC CONCURRENCY CONTROL LOCKING

- A lock is a variable associated with a data item in the database. Generally there is a lock for each data item in the database.
- A lock describes the status of the data item with respect to possible operations that can be applied to that item. It is used for synchronising the access by concurrent transactions to the database items.
 - A transaction locks an object before using it
 - When an object is locked by another transaction, the requesting transaction must wait



TYPES OF LOCKS

- A binary lock can have two states or values: locked and unlocked (1- 0)

lock_item(X):

```
B:  if LOCK(X) = 0          (* item is unlocked *)
    then LOCK(X) ← 1      (* lock the item *)
    else
      begin
        wait (until LOCK(X) = 0
              and the lock manager wakes up the transaction);
        go to B
      end;
```

unlock_item(X):

```
LOCK(X) ← 0;          (* unlock the item *)
if any transactions are waiting
  then wakeup one of the waiting transactions;
```



TYPES OF LOCKS ... A BINARY LOCK

- lock table, a hash file on the item name.
 - Items not in the lock table are considered to be unlocked.
 - In its simplest form, each lock can be a record with three fields: <Data_item_name, LOCK, Locking_transaction> plus a queue for transactions that are waiting to access the item.
- The DBMS has a lock manager subsystem to keep track of and control access to locks.
- T is said to hold the lock on item X.
 - At most one transaction can hold the lock on a particular item.
 - Thus no two transactions can access the same item concurrently.



TYPES OF LOCKS ... A BINARY LOCK

- If the simple binary locking scheme is used, every transaction must obey the following rules:
- A transaction T must issue the operation $\text{lock_item}(X)$ before any $\text{read_item}(X)$ or $\text{write_item}(X)$ operations are performed in T .
- A transaction T must issue the operation $\text{unlock_item}(X)$ after all $\text{read_item}(X)$ and $\text{write_item}(X)$ operations are completed in T .
- A transaction T will not issue a $\text{lock_item}(X)$ operation if it already holds the lock on item X .
- A transaction T will not issue an $\text{unlock_item}(X)$ operation unless it already holds the lock on item X .



TYPES OF LOCKS ... SHARED/EXCLUSIVE (OR READ/WRITE) LOCKS

- There are three locking operations: `read_lock(X)`, `write_lock(X)`, and `unlock(X)`.
- A lock associated with an item `X`, `LOCK(X)`, now has three possible states: read-locked, write-locked, or unlocked.
- A read-locked item is also called share-locked because other transactions are allowed to read the item,
- A write-locked item is called exclusive-locked because a single transaction exclusively holds the lock on the item.



TYPES OF LOCKS ...

SHARED/EXCLUSIVE (OR READ/WRITE) LOCKS

- the system must enforce the following rules:
- A transaction T must issue the operation $\text{read_lock}(X)$ or $\text{write_lock}(X)$ before any $\text{read_item}(X)$ operation is performed in T .
- A transaction T must issue the operation $\text{write_lock}(X)$ before any $\text{write_item}(X)$ operation is performed in T .
- A transaction T must issue the operation $\text{unlock}(X)$ after all $\text{read_item}(X)$ and $\text{write_item}(X)$ operations are completed in T .
- A transaction T will not issue a $\text{read_lock}(X)$ operation if it already holds a read (shared) lock or a write (exclusive) lock on item X .
- A transaction T will not issue a $\text{write_lock}(X)$ operation if it already holds a read (shared) lock or write (exclusive) lock on item X .
- A transaction T will not issue an $\text{unlock}(X)$ operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X .



TYPES OF LOCKS ...

CONVERSION OF LOCKS.

- lock conversion; that is, a transaction that already holds a lock on item X is allowed under certain conditions to convert the lock from one locked state to another.
- it is possible for a transaction T to issue a `read_lock(X)` and then later to upgrade the lock by issuing a `write_lock(X)` operation.
 - If T is the only transaction holding a read lock on X at the time it issues the `write_lock(X)` operation, the lock can be upgraded; otherwise,
 - the transaction must wait.
- It is also possible for a transaction T to issue a `write_lock(X)` and then later to downgrade the lock by issuing a `read_lock(X)` operation.
- When upgrading and downgrading of locks is used, the lock table must include transaction identifiers in the record structure for each lock (in the `locking_transaction(s)` field) to store the information on which transactions hold locks on the item.



- Initial values: $X=20, Y=30$
- Result serial schedule T1 followed by T2: $X=50, Y=80$
- Result of serial schedule T2 followed by T1: $X=70, Y=50$

T_1	T_2
<pre>read_lock(Y); read_item(Y); unlock(Y); write_lock(X); read_item(X); X := X + Y; write_item(X); unlock(X);</pre>	<pre>read_lock(X); read_item(X); unlock(X); write_lock(Y); read_item(Y); Y := X + Y; write_item(Y); unlock(Y);</pre>



Time

T_1	T_2
<pre>read_lock(Y); read_item(Y); unlock(Y);</pre>	<pre>read_lock(X); read_item(X); unlock(X); write_lock(Y); read_item(Y); Y := X + Y; write_item(Y); unlock(Y);</pre>
<pre>write_lock(X); read_item(X); X := X + Y; write_item(X); unlock(X);</pre>	

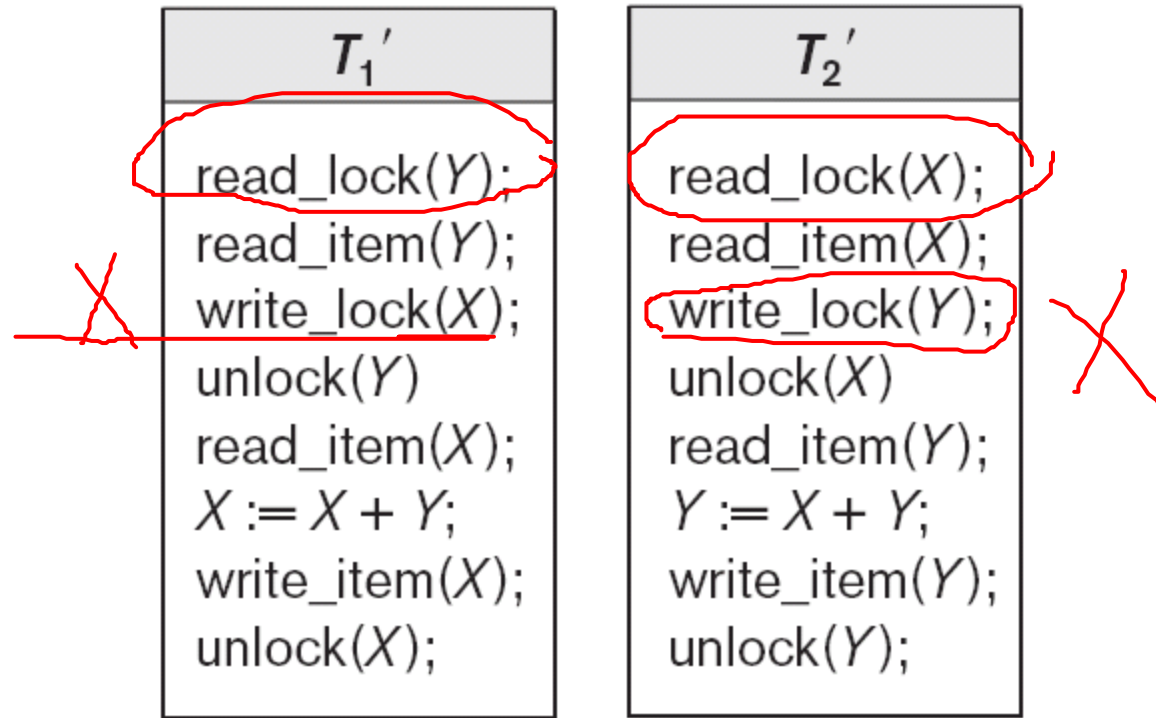
Result of schedule S :
 $X=50, Y=50$
(nonserializable)



GUARANTEEING SERIALIZABILITY BY TWO-PHASE LOCKING

- Two Phases:
 - (a) Locking (Growing / Expanding)
 - (b) Unlocking (Shrinking).
- Locking (Growing) Phase:
 - A transaction applies locks (read or write) on desired data items one at a time.
- Unlocking (Shrinking) Phase:
 - A transaction unlocks its locked data items one at a time.
- Requirement:
 - For a transaction these two phases must be mutually exclusively, that is, during locking phase unlocking phase must not start and during unlocking phase locking phase must not begin.





Transactions T_1 and T_2 , which are the same as T_1 and T_2 , but follow the two-phase locking protocol.



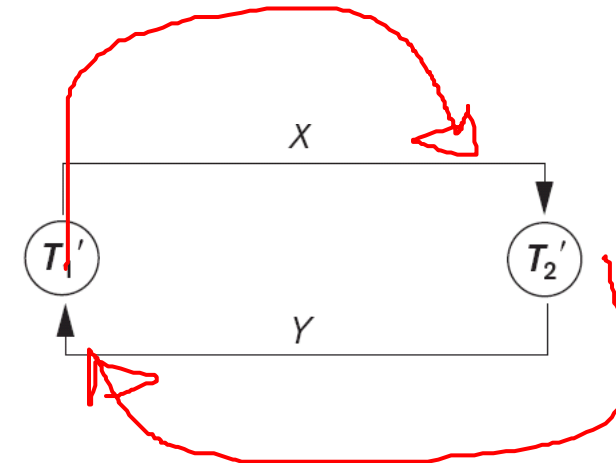
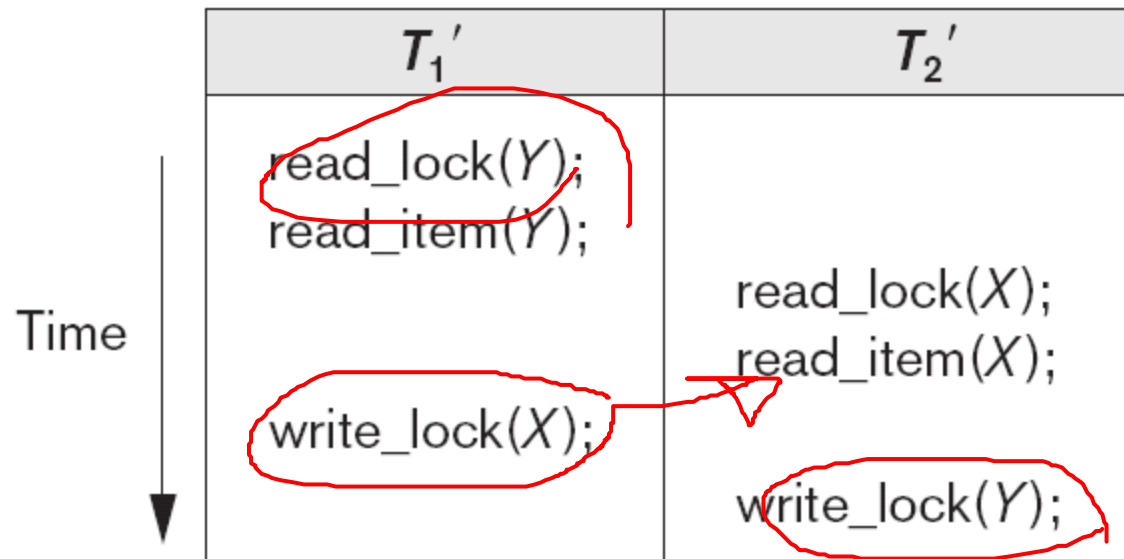
TWO-PHASE LOCKING

- Two-phase policy generates two locking algorithms
 - (a) Basic
 - (b) Conservative
- Basic:
 - Transaction locks data items incrementally. This may cause deadlock which is dealt with.
- Strict:
 - A stricter version of Basic algorithm where unlocking is performed after a transaction terminates (commits or aborts and rolled-back). This is the most commonly used two-phase locking algorithm.
- Conservative:
 - Prevents deadlock by locking all desired data items before transaction begins execution.



DEALING WITH DEADLOCK AND STARVATION

- Deadlock occurs when each transaction T in a set of two or more transactions is waiting for some item that is locked by some other transaction T in the set.



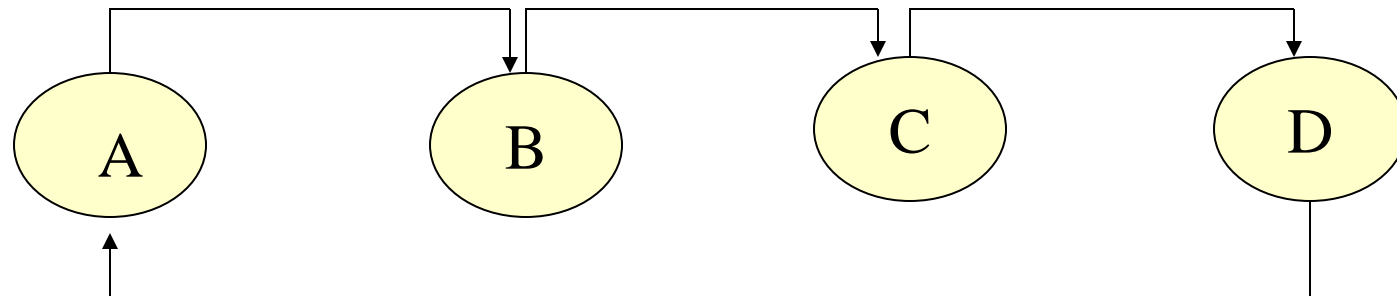
DEALING WITH DEADLOCK AND STARVATION

- Deadlock prevention
 - A transaction locks all data items it refers to before it begins execution.
 - This way of locking prevents deadlock since a transaction never waits for a data item.
 - The conservative two-phase locking uses this approach.



DEALING WITH DEADLOCK AND STARVATION

- Deadlock detection and resolution
 - In this approach, deadlocks are allowed to happen. The scheduler maintains a wait-for-graph for detecting cycle. If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled-back.
 - A wait-for-graph is created using the lock table. As soon as a transaction is locked, it is added to the graph. When a chain like: T_i waits for T_j waits for T_k waits for T_i or T_j occurs, then this creates a cycle.



DEALING WITH DEADLOCK AND STARVATION

- Deadlock avoidance
 - There are many variations of two-phase locking algorithm.
 - Some avoid deadlock by not letting the cycle to complete.
 - That is as soon as the algorithm discovers that blocking a transaction is likely to create a cycle, it rolls back the transaction.
 - Wound-Wait and Wait-Die algorithms use timestamps to avoid deadlocks by rolling-back victim.



DEALING WITH DEADLOCK AND STARVATION

- Wait-die
 - If $TS(T_i) < TS(T_j)$, then (T_i older than T_j)
 - T_i is allowed to wait
 - Otherwise (T_i younger than T_j)
 - Abort T_i (T_i dies) and restart it later with the same timestamp
- Wound-wait
 - If $TS(T_i) < TS(T_j)$, then (T_i older than T_j)
 - Abort T_j (T_i wounds T_j) and restart it later with the same timestamp
 - Otherwise (T_i younger than T_j)
 - T_i is allowed to wait

old / young
wait / die



DEALING WITH DEADLOCK AND STARVATION

■ Starvation

- Starvation occurs when a particular transaction consistently waits or restarted and never gets a chance to proceed further.
- In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled-back.
- This limitation is inherent in all priority-based scheduling mechanisms.
- One solution for starvation is to have a fair waiting scheme, such as using a first-come-first-served queue
- In Wound-Wait scheme a younger transaction may always be wounded (aborted) by a long running older transaction which may create starvation.



CONCURRENCY CONTROL BASED ON TIMESTAMP ORDERING

- Timestamp is a unique identifier created by the DBMS to identify a transaction.
- timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the transaction start time.
- The timestamp of transaction T is $TS(T)$.
- Time Stamps use:
 - 1. Counter that is incremented each time its value is assigned to a transaction (1,2,3...)
 - 2. Use system clock and ensure that no two timestamp values are generated during the same tick of the clock.



THE TIMESTAMP ORDERING ALGORITHM

- 1. $\text{read_TS}(X)$. The read timestamp of item X is the largest timestamp among all the timestamps of transactions that have successfully read item X —that is,
 - $\text{read_TS}(X) = \text{TS}(T)$, where T is the youngest transaction that has read X successfully.
- 2. $\text{write_TS}(X)$. The write timestamp of item X is the largest of all the timestamps of transactions that have successfully written item X —that is,
 - $\text{write_TS}(X) = \text{TS}(T)$, where T is the youngest transaction that has written X successfully.

$T_1(X)$

$T_2(X)$


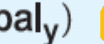




13 41



BASIC TIMESTAMP ORDERING (TO).

- 1. Transaction T issues a $\text{write_item}(X)$ operation:
 - If $\text{read_TS}(X) > \text{TS}(T)$ or if $\text{write_TS}(X) > \text{TS}(T)$, then an younger transaction has already read the data item so abort and roll-back T and reject the operation.
 - If the condition in part (a) does not exist, then execute $\text{write_item}(X)$ of T and set $\text{write_TS}(X)$ to $\text{TS}(T)$.
- 2. Transaction T issues a $\text{read_item}(X)$ operation:
 - If $\text{write_TS}(X) > \text{TS}(T)$, then an younger transaction has already written to the data item so abort and roll-back T and reject the operation.
 - If $\text{write_TS}(X) \leq \text{TS}(T)$, then execute $\text{read_item}(X)$ of T and set $\text{read_TS}(X)$ to the larger of $\text{TS}(T)$ and the current $\text{read_TS}(X)$.



Time	Op	T ₁₉	T ₂₀	T ₂₁
t ₁	 begin_transaction			
t ₂	read(bal_x)	read(bal_x)		
t ₃	bal_x = bal_x + 10	bal_x = bal_x + 10		
t ₄	write(bal_x)	write(bal_x)	begin_transaction	
t ₅	read(bal_y)		read(bal_y)	
t ₆	bal_y = bal_y + 20		bal_y = bal_y + 20	begin_transaction
t ₇	read(bal_y)			read(bal_y) 
t ₈	write(bal_y)		write(bal_y) [†] 	
t ₉	bal_y = bal_y + 30			bal_y = bal_y + 30
t ₁₀	write(bal_y)			write(bal_y) 
t ₁₁	bal_z = 100			bal_z = 100
t ₁₂	write(bal_z)			write(bal_z) 
t ₁₃	bal_z = 50	bal_z = 50		commit
t ₁₄	write(bal_z)	write(bal_z) [‡] 	begin_transaction	
t ₁₅	read(bal_y)	commit	read(bal_y)	
t ₁₆	bal_y = bal_y + 20		bal_y = bal_y + 20	
t ₁₇	write(bal_y)		write(bal_y)	
t ₁₈			commit	



STRICT TIMESTAMP ORDERING

- 1. Transaction T issues a $\text{write_item}(X)$ operation:
 - If $\text{TS}(T) > \text{read_TS}(X)$, then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).
- 2. Transaction T issues a $\text{read_item}(X)$ operation:
 - If $\text{TS}(T) > \text{write_TS}(X)$, then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).



Questions?

