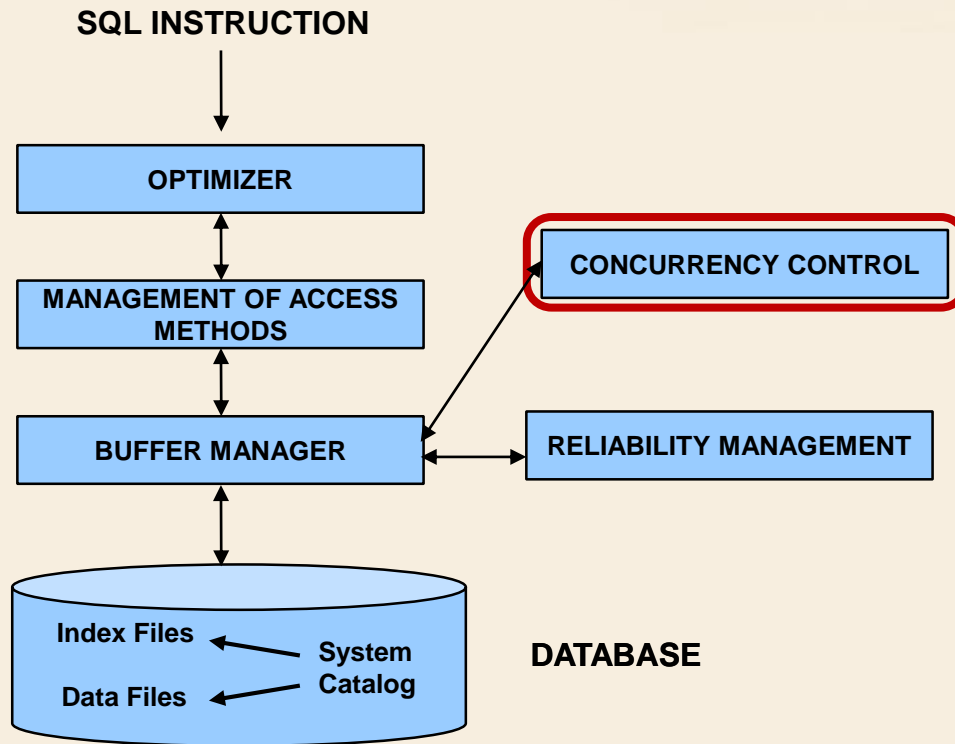




# Database Management Systems

## Concurrency Control

# DBMS Architecture



# Concurrency control

- The workload of operational DBMSs is measured in tps, i.e., transactions per second
  - $\approx 10\text{-}10^3$  for banking applications and flight reservations
- Concurrency control provides *concurrent access* to data
  - It increases DBMS efficiency by
    - maximizing the number of transactions per second (throughput)
    - minimizing response time

# Elementary I/O operations

- Elementary operations are
  - Read of a single data object  $x$ 
    - $r(x)$
  - Write of a single data object  $x$ 
    - $w(x)$
- They may require reading from disk or writing to disk an entire page

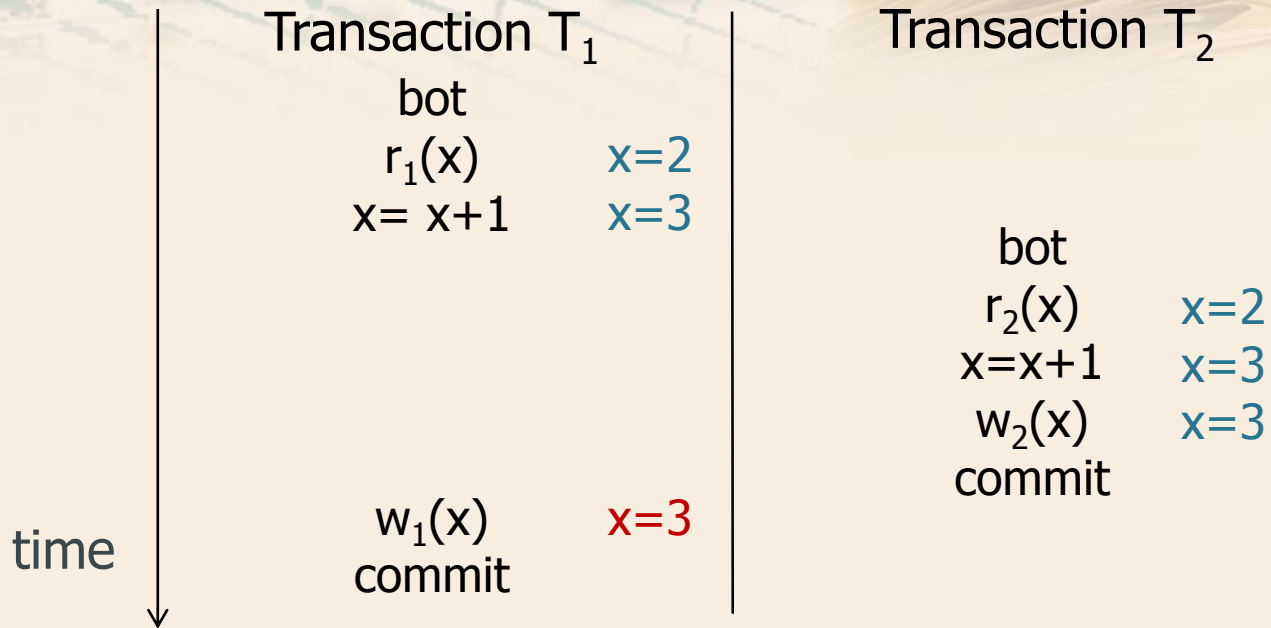
## ➤ The *scheduler*

- is a block of the concurrency control manager
- is in charge of deciding if and when read/write requests can be satisfied

## ➤ The absence of a scheduler may cause correctness problems

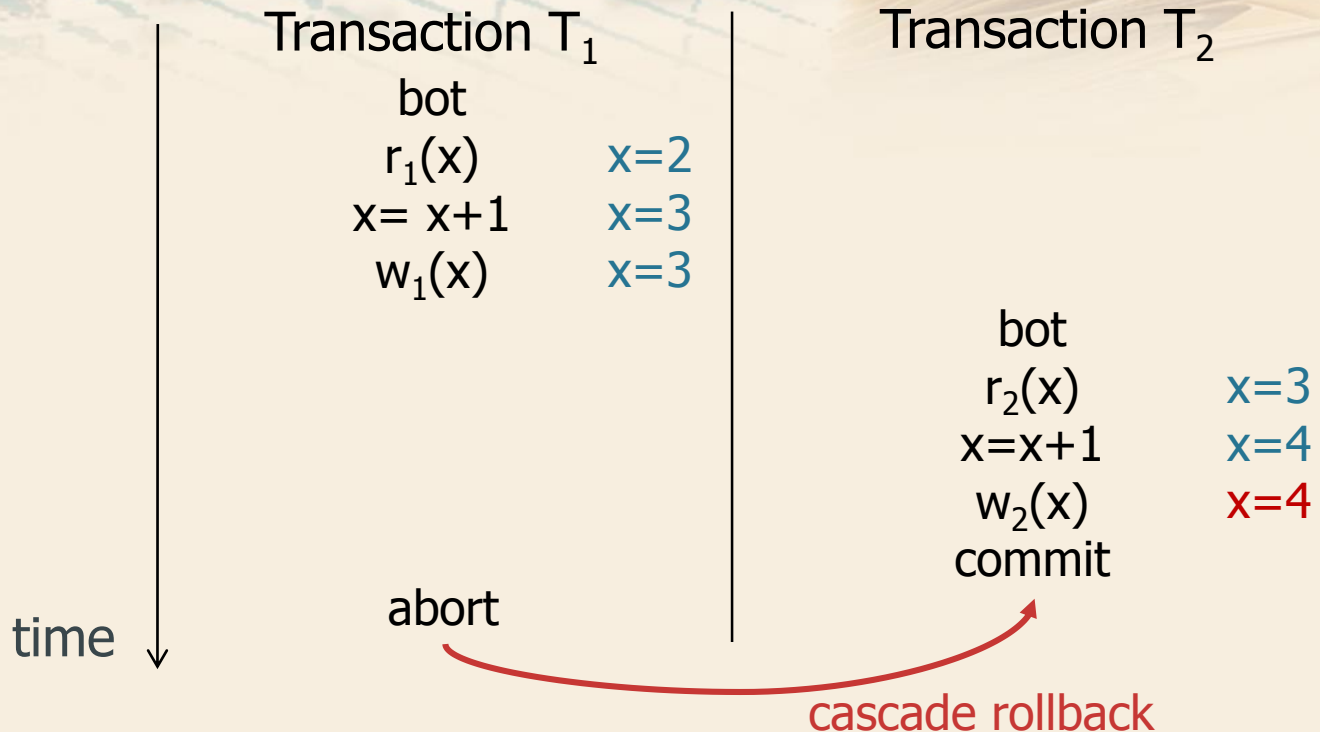
- also called anomalies

# Lost update



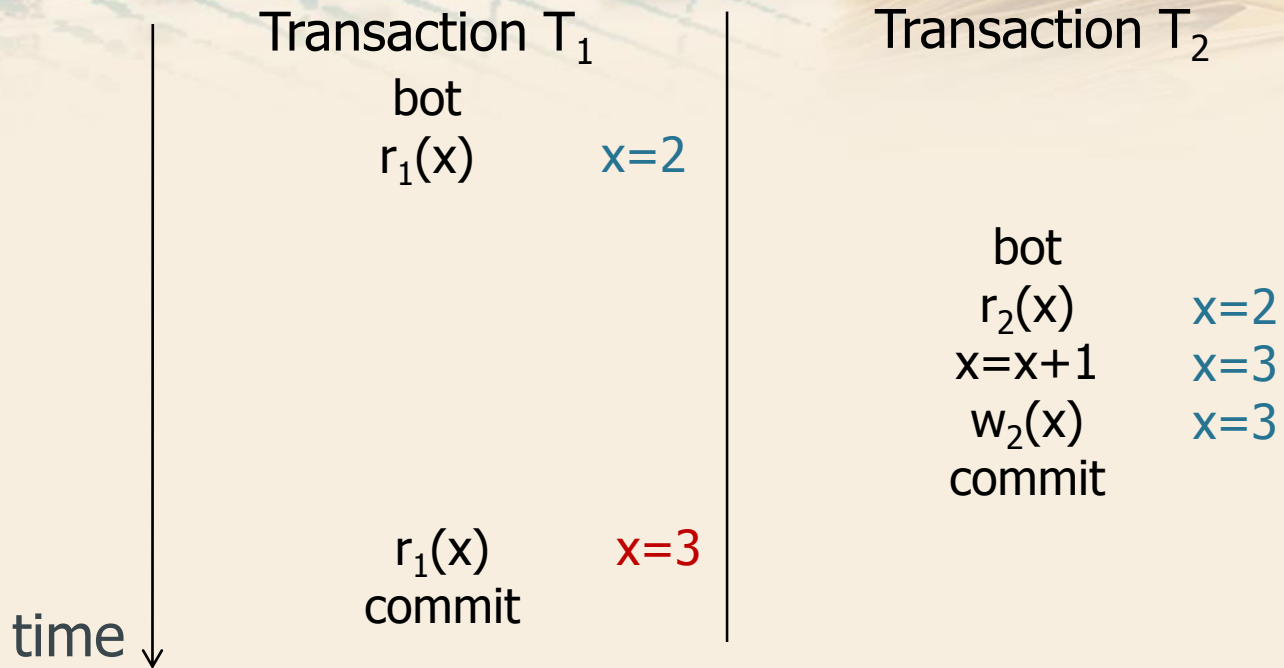
- The *correct* value is  $x=4$
- The effect of transaction  $T_2$  is *lost* because both transactions read the same initial value

## Dirty read



➤ Transaction  $T_2$  reads the value of  $X$  in an intermediate state which *never* becomes stable (permanent)

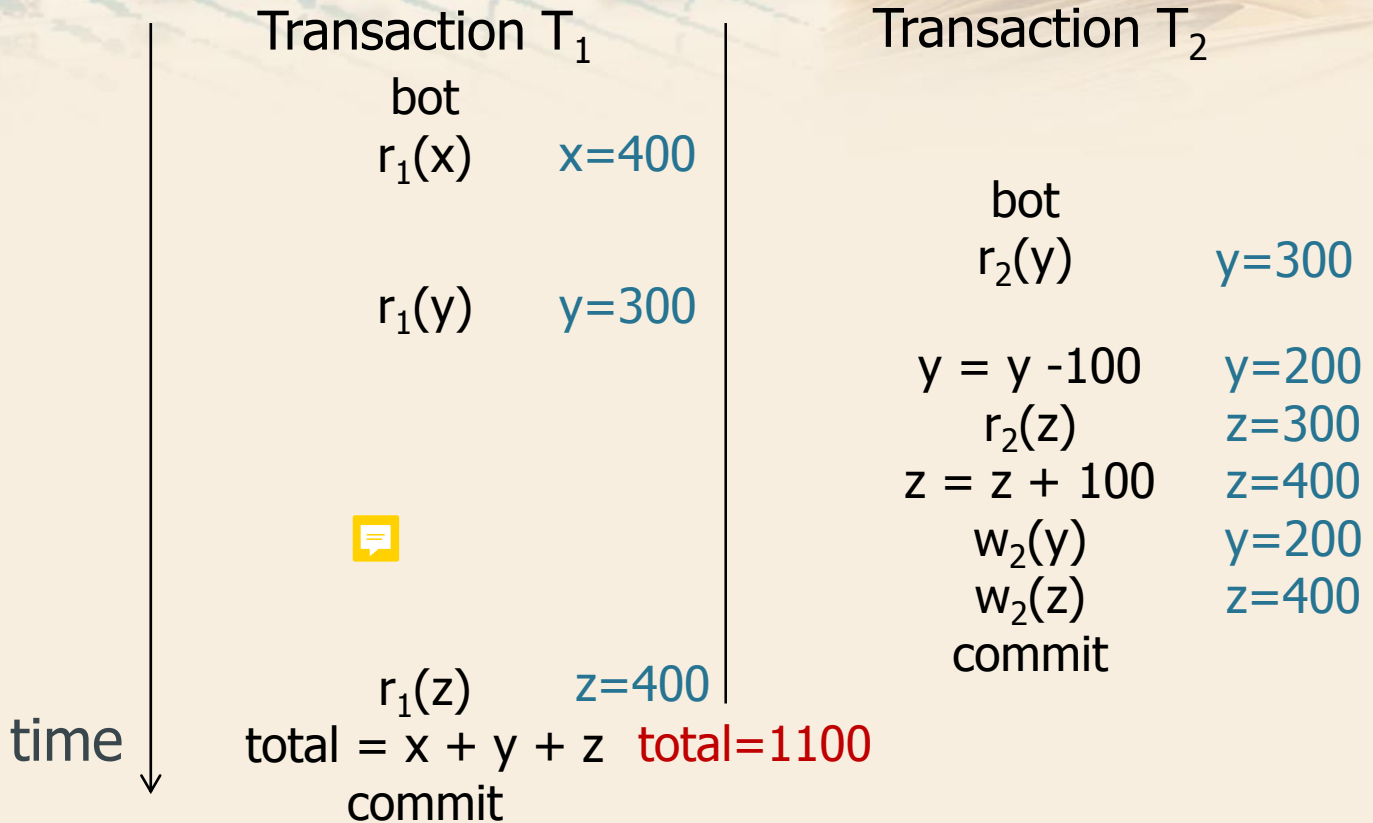
# Inconsistent read



- ⇒ Transaction  $T_1$  reads  $x$  twice
- $x$  has a different value each time



## Ghost update (a)

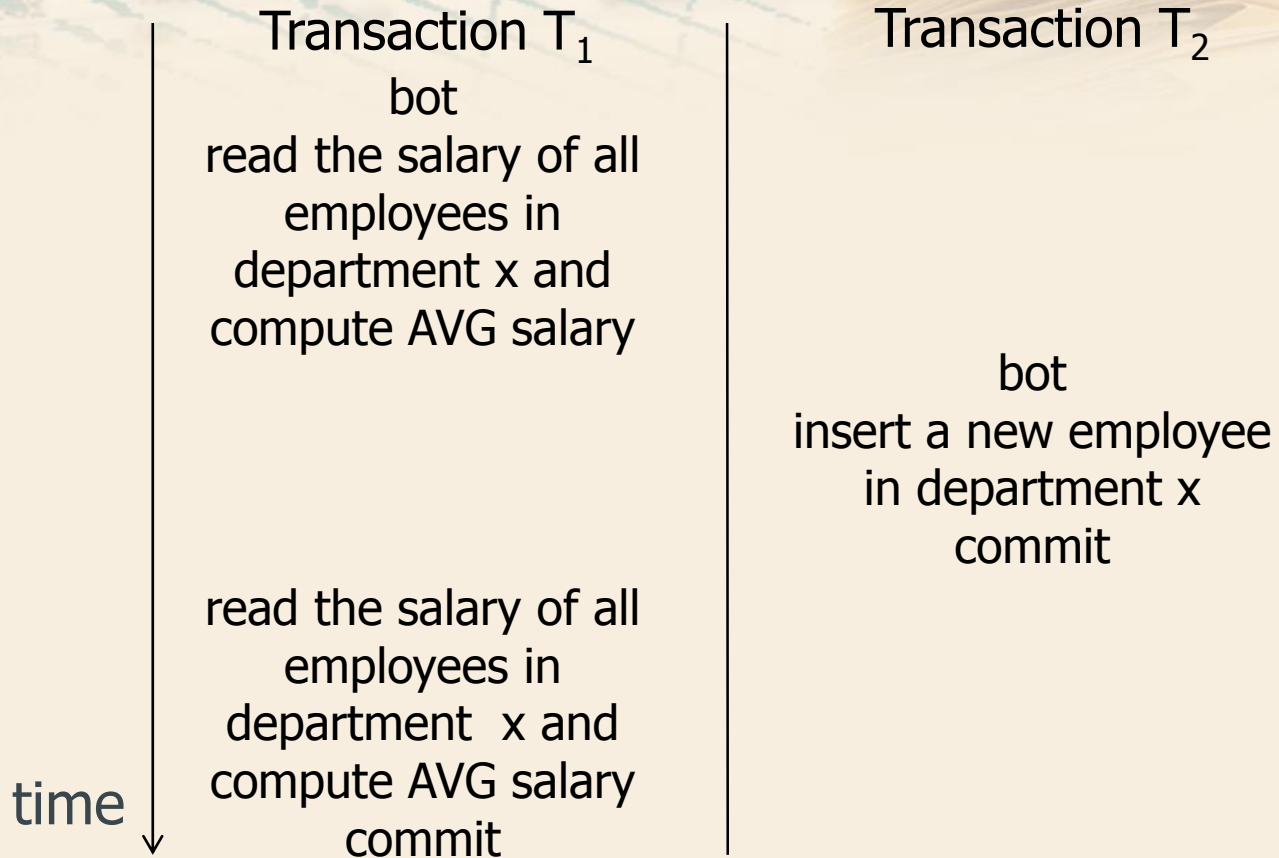


➤ The *correct* value is total =  $400+200+400=1000$

## Ghost update (a)

⤵ Transaction  $T_1$  only *partially* observes the effect of transaction  $T_2$

## Ghost update (b)



## Ghost update (b)

- The insert operation is the ghost update
- Problem
  - The data is *not yet* in the database before the insert



# Database Management Systems

## Theory of Concurrency Control

- The *transaction* is a sequence of read and write operations characterized by the same TID (Transaction Identifier)

$$r_1(x) \ r_1(y) \ w_1(x) \ w_1(y)$$

- The *schedule* is a sequence of read/write operations presented by concurrent transactions

$$r_1(z)r_2(z)w_1(y)w_2(z)$$

- Operations in the schedule appear in the arrival order of requests

- Concurrency control accepts or rejects schedules to avoid anomalies
- The scheduler has to accept or reject operation execution *without knowing the outcome* of the transactions
  - abort/commit

## Commit projection

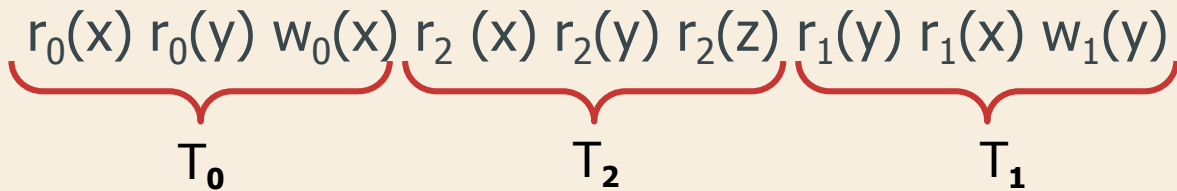
- Commit projection is a simplifying hypothesis  
*The schedule only contains transactions performing commit*
- The dirty read anomaly is not addressed
- This hypothesis will be removed later



# Serial schedule

➤ In a *serial schedule*, the actions of each transaction appear in sequence, without interleaved actions belonging to different transactions

➤ Example



# Serializable schedule

- An arbitrary schedule  $S_i$  (commit projection) is correct when it yields the same result as an arbitrary serial schedule  $S_j$  of the same transactions
- $S_i$  is *serializable*
  - $S_i$  is equivalent to an arbitrary serial schedule of the same transactions

# Equivalence between schedules

- Different *equivalence classes* between two schedules
  - View equivalence
  - Conflict equivalence
  - 2 phase locking
  - Timestamp equivalence
- Each equivalence class
  - detects a set of acceptable schedules
  - is characterized by a different complexity in detecting equivalence

# View equivalence

## ➤ Definitions

- reads-from

- $r_i(x)$  reads-from  $w_j(x)$  when
  - $w_j(x)$  precedes  $r_i(x)$  and  $i \neq j$
  - there is no other  $w_k(x)$  between them

- final write

- $w_i(x)$  is a final write if it is the last write of  $x$  appearing in the schedule

## ➤ Two schedules are *view equivalent* if they have

- the same reads-from set
- the same final write set


# View serializable schedule

➤ A schedule is *view serializable* if it is view equivalent to an arbitrary serial schedule of the same transactions


- VSR: schedules which are view serializable

➤ Example

$S_1 = w_0(x) \ r_2(x) \ r_1(x) \ \underline{w_2(x)} \ \underline{w_2(z)}$

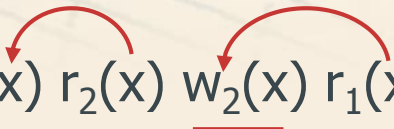


$S_2 = w_0(x) \ r_1(x) \ r_2(x) \ \underline{w_2(x)} \ \underline{w_2(z)}$

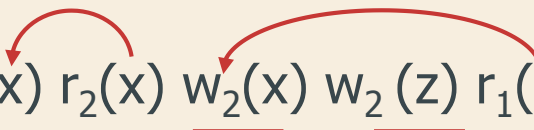


➤  $S_1$  is view serializable because it is view equivalent to  $S_2$

## View equivalence

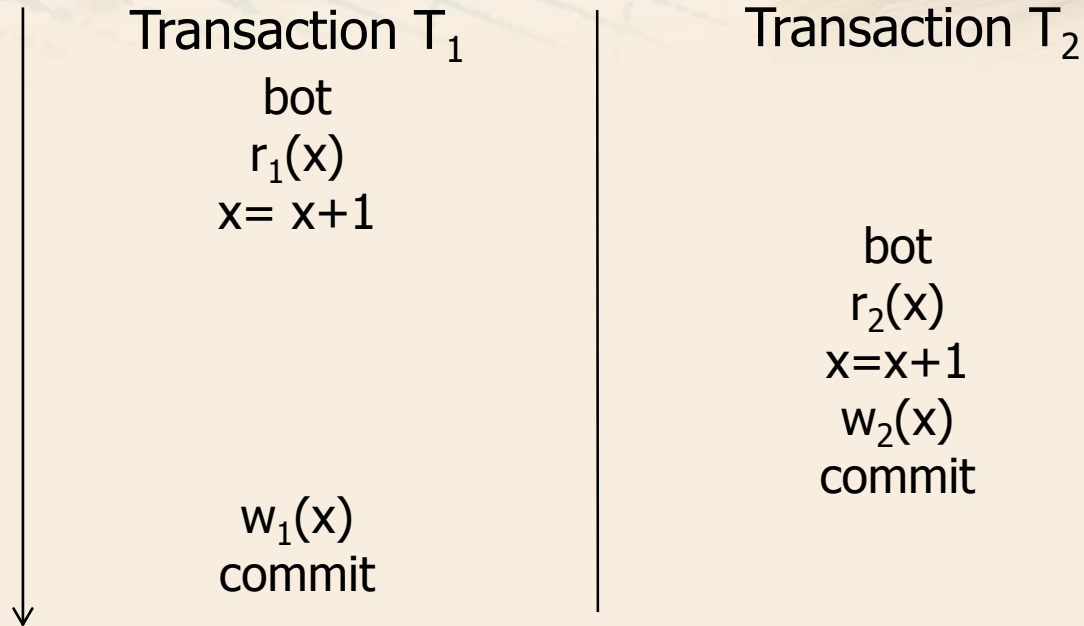
$$S_3 = w_0(x) \ r_2(x) \ \underline{w_2(x)} \ r_1(x) \ \underline{w_2(z)}$$


- $S_3$  is not view equivalent to  $S_2$
- the reads-from sets are different

$$S_4 = w_0(x) \ r_2(x) \ \underline{w_2(x)} \ \underline{w_2(z)} \ r_1(x)$$


- $S_3$  is view serializable because it is view equivalent to  $S_4$

# Lost update anomaly



➤ Corresponding schedule


$$S = r_1(x) \ r_2(x) \ w_2(x) \ w_1(x)$$


## Lost update anomaly

$$S = r_1(x) \ r_2(x) \ w_2(x) \ \underline{w_1(x)}$$

➤ Is this schedule serializable?

➤ Only two possible serial schedules

$$S_1 = r_1(x) \ w_1(x) \ r_2(x) \ \underline{w_2(x)}$$


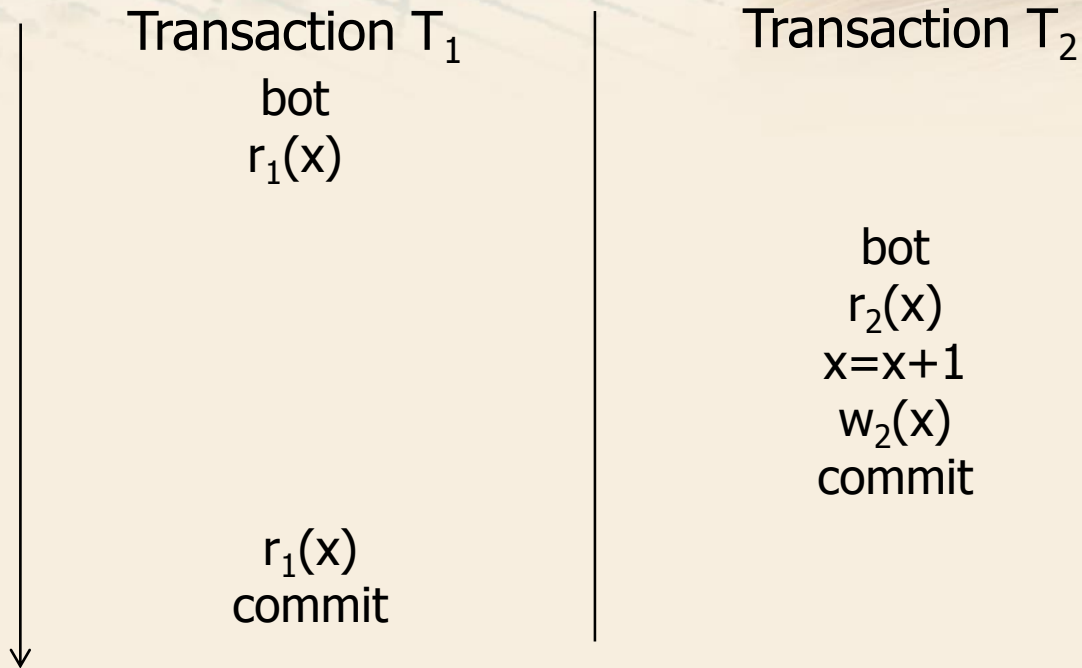
$$S_2 = r_2(x) \ w_2(x) \ r_1(x) \ \underline{w_1(x)}$$


➤ S is not view equivalent to any serial schedule

- not serializable
- should be rejected




# Inconsistent read anomaly



➤ Corresponding schedule

$$S = r_1(x) \ r_2(x) \ w_2(x) \ r_1(x)$$


# Inconsistent read anomaly

$$S = r_1(x) \ r_2(x) \ \underline{w_2(x)} \ r_1(x)$$


➤ Is this schedule serializable?

➤ Only two possible serial schedules

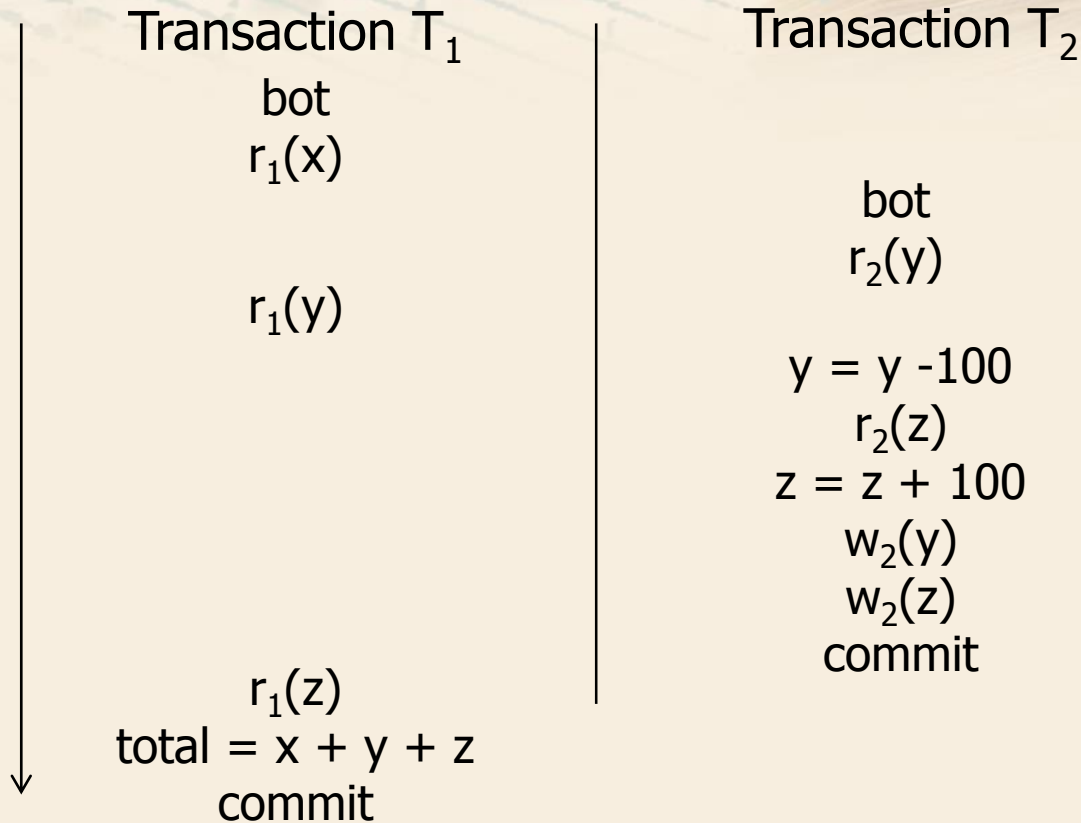
$$S_1 = r_1(x) \ r_1(x) \ r_2(x) \ \underline{w_2(x)}$$

$$S_2 = r_2(x) \ \underline{w_2(x)} \ r_1(x) \ r_1(x)$$


➤ S is not view equivalent to any serial schedule


- not serializable
- should be rejected

# Ghost Update (a)



$$S = r_1(x) \ r_2(y) \ r_1(y) \ r_2(z) \ w_2(y) \ w_2(z) \ r_1(z)$$


## Ghost Update (a)

$$S = r_1(x) \ r_2(y) \ r_1(y) \ r_2(z) \ \underline{w_2(y)} \ \underline{w_2(z)} \ r_1(z)$$


➤ Is this schedule serializable?

➤ Only two possible serial schedules

$$S_1 = r_1(x) \ r_1(y) \ r_1(z) \ r_2(y) \ r_2(z) \ \underline{w_2(y)} \ \underline{w_2(z)}$$

$$S_2 = r_2(y) \ r_2(z) \ \underline{w_2(y)} \ \underline{w_2(z)} \ r_1(x) \ r_1(y) \ r_1(z)$$


➤ S is not view equivalent to any serial schedule

## Checking view serializability

- Detecting view equivalence to a *given* schedule has linear complexity
- Detecting view equivalence to an *arbitrary* serial schedule is NP complete
  - not feasible in real systems
- Less accurate but faster techniques should be considered

# Conflict equivalence

## ➤ Conflicting actions

- Action  $A_i$  is in conflict with action  $A_j$  ( $i \neq j$ ) if both actions operate on the same object and at least one of them is a write
  - Read-Write conflicts (RW or WR)
  - Write-Write conflicts (WW)

## ➤ Two schedules are *conflict equivalent* if

- they have the same conflict set
- each *conflict pair* is in the same order in both schedules

# Conflict serializable schedule

➤ A schedule is *conflict serializable* if it is equivalent to an arbitrary serial schedule of the same transactions

- CSR: schedules which are conflict serializable

➤ Example

$$S = w_0(x) \ r_1(x) \ w_0(z) \ r_1(z) \ r_2(x) \ r_3(z) \ w_3(z) \ w_1(x)$$
$$S_s = w_0(x) \ w_0(z) \ r_2(x) \ r_1(x) \ r_1(z) \ w_1(x) \ r_3(z) \ w_3(z)$$

# Conflict serializable schedule

## ➤ Example

$S = w_0(x) \ r_1(x) \ w_0(z) \ r_1(z) \ r_2(x) \ r_3(z) \ w_3(z) \ w_1(x)$

$S_s = w_0(x) \ w_0(z) \ r_2(x) \ r_1(x) \ r_1(z) \ w_1(x) \ r_3(z) \ w_3(z)$

➤ Schedule S is conflict serializable





# Detecting conflict serializability

- To detect conflict serializability it is possible to exploit the *conflict graph*
- Conflict graph
  - a node for each transaction
  - an edge  $T_i \rightarrow T_j$  if
    - there exists at least a conflict between an action  $A_i$  in  $T_i$  and  $A_j$  in  $T_j$
    - $A_i$  precedes  $A_j$
- If the conflict graph is acyclic the schedule is CSR
- Checking graph cyclicity is linear in the size of the graph

## Example of conflict graph

$S = w_0(x) r_1(x) w_0(z) r_1(z) r_2(x) r_3(z) w_3(z) w_1(x)$

The diagram shows a sequence of literals:  $w_0(x)$ ,  $r_1(x)$ ,  $w_0(z)$ ,  $r_1(z)$ ,  $r_2(x)$ ,  $r_3(z)$ ,  $w_3(z)$ , and  $w_1(x)$ . Red lines connect conflicting pairs of literals, indicating dependencies or conflicts. The connections are as follows:  $w_0(x)$  is connected to  $w_0(z)$  and  $w_1(x)$ ;  $r_1(x)$  is connected to  $r_1(z)$ ;  $w_0(z)$  is connected to  $r_1(z)$  and  $w_3(z)$ ;  $r_1(z)$  is connected to  $r_2(x)$  and  $w_3(z)$ ;  $r_2(x)$  is connected to  $w_1(x)$ ;  $r_3(z)$  is connected to  $w_3(z)$ ; and  $w_3(z)$  is connected to  $w_1(x)$ .

$T_0$

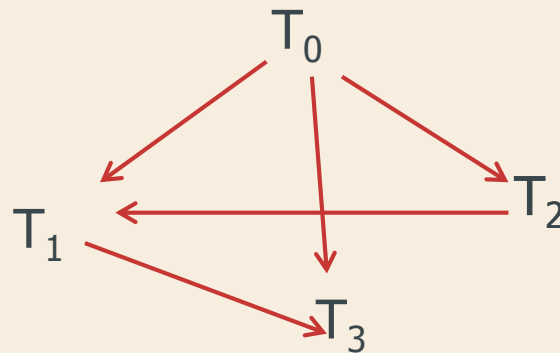
$T_1$

$T_2$

$T_3$

## Example of conflict graph

$S = w_0(x) \ r_1(x) \ w_0(z) \ r_1(z) \ r_2(x) \ r_3(z) \ w_3(z) \ w_1(x)$



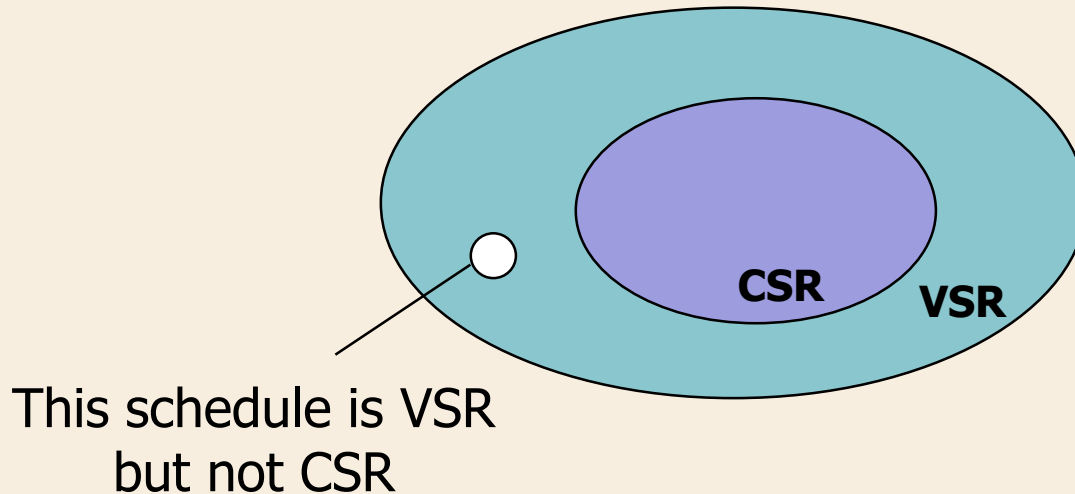
➤  $S$  is CSR (no cycles)

# Detecting conflict serializability

- Real system settings
  - 100 tps (transactions per second)
  - each transaction accesses  $\approx 10$  pages
  - each transaction lasts  $\approx 5s$
- The conflict graph is characterized by 500 nodes
  - $100 \text{ tps} * 5 \text{ seconds}$
- Accesses to be checked for conflicts
  - $500 \text{ nodes} * 10 \text{ page accessed} \approx 5000 \text{ accesses}$
- At each access
  - the graph should be updated
  - cycle absence should be checked

# VSR versus CRS

➤ CSR schedules are a subset of VSR schedules





# Database Management Systems

## 2 Phase Locking

- A *lock* is a block on a resource which may prevent access to others
- Lock operation
  - Lock
    - Read lock (R-Lock)
    - Write lock (W-Lock)
  - Unlock
- Each read operation
  - is preceded by a request of R-Lock
  - is followed by a request of unlock
- Similarly for write operation and W-Lock

- The read lock is *shared* among different transactions
- The write lock is *exclusive*
  - it is not compatible with any other lock (R/W) on the same data
- Lock escalation
  - request of R-Lock followed by W-Lock on the same data



- The scheduler becomes a lock manager
- It receives transaction requests and grants locks based on locks already granted to other transactions
  - When the lock request is granted
    - The corresponding resource is acquired by the requesting transaction
    - When the transaction performs unlock, the resource becomes again available
  - When the lock is not granted
    - The requesting transaction is put in a waiting state
    - Wait terminates when the resource is unlocked and becomes available

# Lock manager

- The lock manager exploits
- the information in the *lock table* to decide if a given lock can be granted to a transaction
  - the *conflict table* to manage lock conflicts

# Conflict table

Request	Resource State		
	Free	R-Locked	W-Locked
R-Lock			
W-Lock			
Unlock			

# Conflict table

Request	Resource State		
	Free	R-Locked	W-Locked
R-Lock	Ok/R-Locked	Ok/R-Locked	No/W-Locked
W-Lock	Ok/W-Locked	No/R-Locked	No/W-Locked
Unlock	Error	Ok/It depends (free if no other R-Locked)	Ok/Free

## ➤ Read locks are shared

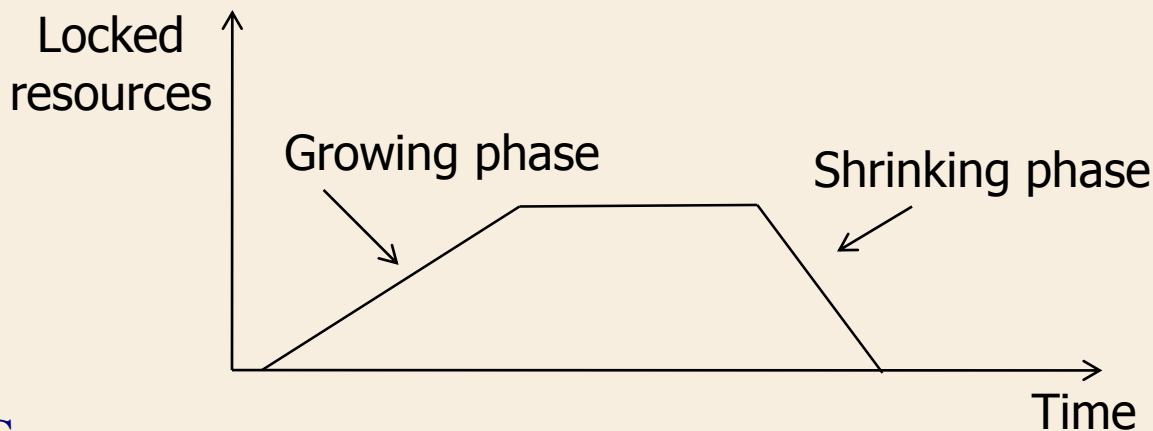
- Other transactions may lock the same resource
- A counter is used to count the number of transactions currently holding the R-Lock
  - Free when count = 0

➤ The lock manager exploits

- the information in the *lock table* to decide if a given lock can be granted to a transaction
  - stored in main memory
  - for each data object
    - 2 bits to represent the 3 possible object states (free, r\_locked, w\_locked)
    - a counter to count the number of waiting transactions

## 2 Phase Locking

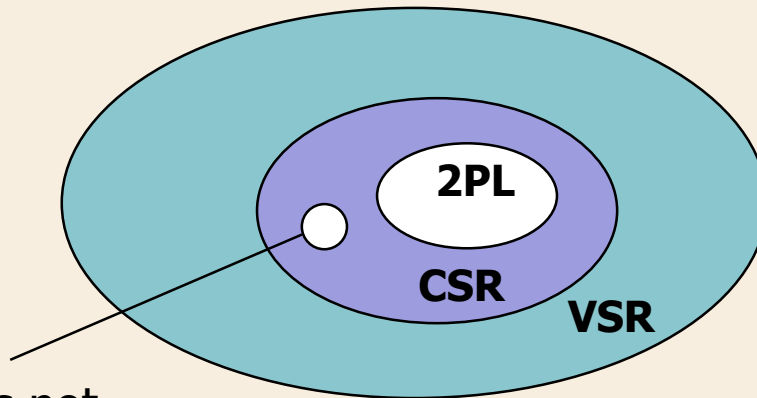
- Exploited by most commercial DBMS
- It is characterized by two phases
  - Growing phase
    - needed locks are acquired
  - Shrinking phase
    - all locks are released



## 2 Phase Locking

➤ 2 Phase Locking guarantees serializability

*A transaction cannot acquire a new lock after having released any lock*



This schedule is not  
accepted by 2PL but  
it is serializable

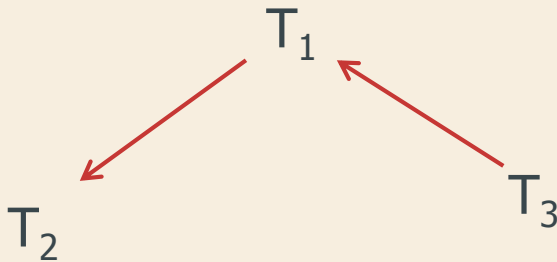


## Example

$S = r_1(x) \ w_1(x) \ r_2(x) \ w_2(x) \ r_3(y) \ w_1(y)$

$T_1$  releases  
the lock on x

$T_1$  should acquire  
a new lock on y



➤ The schedule is CSR but not 2PL

# Ghost update (a)

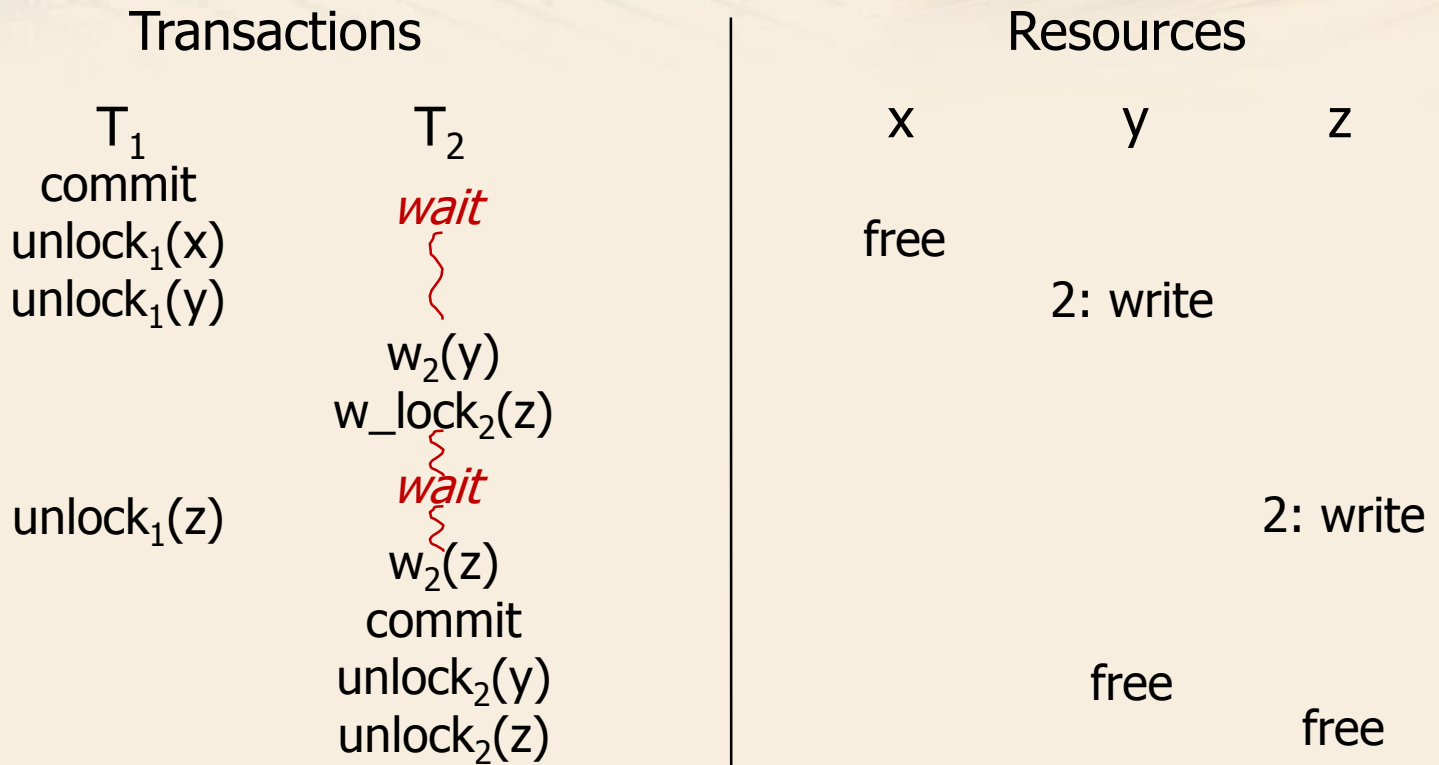
## Transactions

$T_1$	$T_2$
bot	
$r\_lock_1(x)$	
$r_1(x)$	
	bot
	$r\_lock_2(y)$
	$r_2(y)$
$r\_lock_1(y)$	
$r_1(y)$	
	$r\_lock_2(z)$
	$r_2(z)$
	$w\_lock_2(y)$
$r\_lock_1(z)$	
$r_1(z)$	
	<i>wait</i>

## Resources

x	y	z
free	free	free
1: read		
	2: read	
	1,2: read	
		2: read
		1,2: read

# Ghost update (a)



## Strict 2 Phase Locking

- *Strict* 2 Phase Locking allows dropping the commit projection hypothesis
  - A transaction locks may be released only *at the end* of the transaction
    - After COMMIT/ROLLBACK
- After the end of the transaction, data is stable
  - It avoids the dirty read anomaly

# Lock Manager service interface

## ➤ Primitives

- R-Lock (T, x, ErrorCode, TimeOut)
- W-Lock (T, x, ErrorCode, TimeOut)
- UnLock (T, x)

## ➤ Parameters

- T: Transaction ID of the requesting transaction
- x: requested resource
- ErrorCode: return parameter
  - Ok
  - Not Ok (request not satisfied)
- TimeOut
  - Maximum time for which the transaction is willing to wait

# Techniques to manage locking

- A transaction requests a resource  $x$
- If the request *can be satisfied*
  - The lock manager modifies the state of resource  $x$  in its internal tables
  - It returns control to the requesting transaction
- The processing delay is very small

# Techniques to manage locking

- If the request *cannot be satisfied* immediately
  - The requesting transaction is inserted in a waiting queue and suspended
  - When the resource becomes available
    - the first transaction (process) in the waiting queue is resumed and is granted the lock on the resource
- Probability of a conflict  $\approx (K \times M) / N$ 
  - K is the number of active transactions
  - M is the average number of objects accessed by a transaction
  - N is the number of objects in the database

# Techniques to manage locking

- When a *timeout* expires while a transaction is still waiting, the lock manager
  - extracts the waiting transaction from the queue
  - resumes it
  - returns a not ok error code
- The requesting transaction may
  - perform rollback (and possibly restart)
  - request again the same lock after some time
    - without releasing locks on other acquired resources





# Database Management Systems

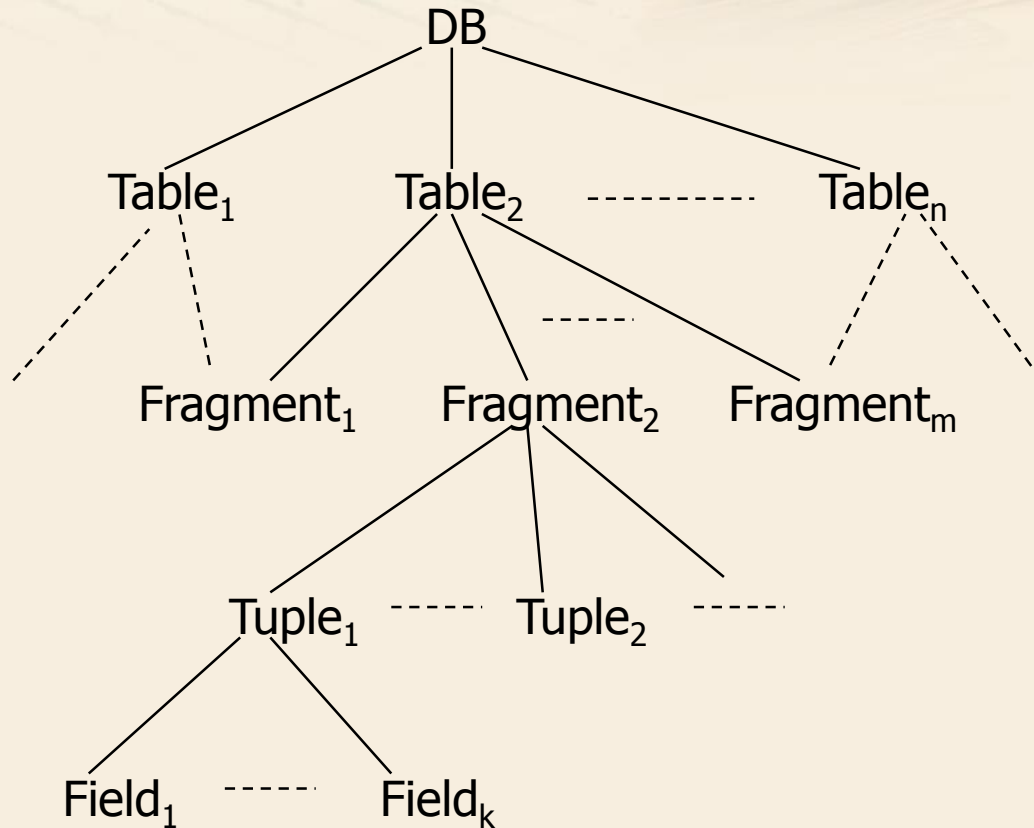
## Hierarchical Locking

# Hierarchical locking

➤ Table locks can be acquired at different *granularity* levels

- Table
- Group of tuples (fragment)
  - Physical partitioning criteria
    - e.g., data page
  - Logical partitioning criteria
    - e.g. tuples satisfying a given property
- Single tuple
- Single field in a tuple

# Hierarchical locking



# Hierarchical locking

- Hierarchical locking is an extension of traditional locking
- It allows a transaction to request a lock at the appropriate level of the hierarchy
  - It is characterized by a larger set of locking primitives

# Locking primitives

- Shared Lock (SL)
- eXclusive Lock (XL)
- Intention of Shared Lock (ISL)
  - It shows the intention of shared locking on an object which is in a lower node in the hierarchy
    - i.e., a descendant of the current node
- Intention of eXclusive Lock (IXL)
  - Analogous to ISL, but for exclusive lock

# Locking primitives

- Shared lock and Intention of eXclusive Lock (SIXL)
  - Shared lock of the current object and intention of exclusive lock for one or more objects in a descendant node

# Request protocol

1. Locks are always requested starting from the tree root and going down the tree
2. Locks are released starting from the blocked node of smaller granularity and going up the tree
3. To request a SL or an ISL on a given node, a transaction must own an ISL (or IXL) on its parent node in the tree
4. To request an XL, IXL or SIXL on a given node, a transaction must own an IXL or SIXL on its parent node in the tree

# Compatibility matrix

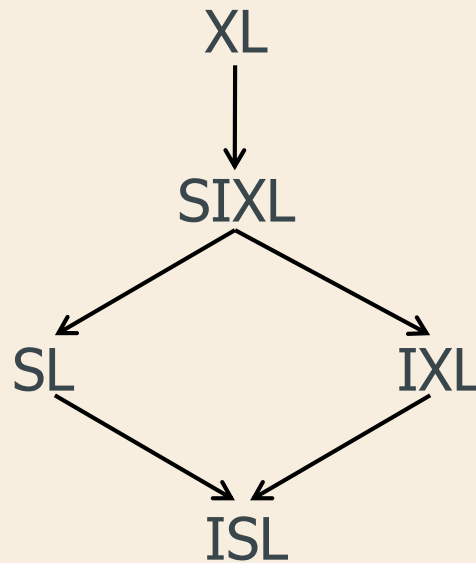
	Resource State				
Request	ISL	IXL	SL	SIXL	XL
ISL					
IXL					
SL					
SIXL					
XL					



# Compatibility matrix

	Resource State				
Request	ISL	IXL	SL	SIXL	XL
ISL	Ok	Ok	Ok	Ok	No
IXL	Ok	Ok	No	No	No
SL	Ok	No	Ok	No	No
SIXL	Ok	No	No	No	No
XL	No	No	No	No	No

# Precedence graph for locks



# Selection of lock granularity

➤ It depends on the application type

- if it performs *localized* reads or updates of few objects
  - low levels in the hierarchy (detailed granularity)
- if it performs *massive* reads or updates
  - high levels in the hierarchy (rough granularity)

➤ Effect of lock granularity

- if it is too coarse, it reduces concurrency
  - high likeliness of conflicts
- if it is too fine, it forces a significant overhead on the lock manager

# Predicate locking

- It addresses the ghost update of type b (insert) anomaly
  - for 2PL a read operation *is not* in conflict with the insert of a new tuple
    - the new tuple can't be locked in advance
- *Predicate locking* allows locking all data satisfying a given predicate
  - implemented in real systems by locking indices

# Locking in SQL2 standard

## ➤ Transaction types

- read-write (default case)
- read only
  - no data or schema modifications are allowed
  - shared locks are enough

## ➤ The *isolation level* of a transaction specifies how it interacts with the other executing transactions

- it may be set by means of SQL statements

# Isolation levels

## ➤ SERIALIZABLE

- the highest isolation level
- it includes predicate locking

## ➤ REPEATABLE READ

- strict 2PL without predicate locking
- reads of existing objects can be correctly repeated
- no protection against ghost update (b) anomaly
  - the computation of aggregate functions cannot be repeated

# Isolation levels

## ➤ READ COMMITTED

- not 2PL
- the read lock is released as soon as the object is read
- reading intermediate states of a transaction is avoided
  - dirty reads are avoided

## ➤ READ UNCOMMITTED

- not 2PL
- data is read without acquiring the lock
  - dirty reads are allowed
- only allowed for read only transactions

# Locking in SQL2 standard

- The isolation level of a transaction may be set by means of the statement

SET TRANSACTION

[ISOLATION LEVEL <IsolationLevel>]

[READ ONLY]

[READ WRITE]

- The isolation level may be reduced only for read operations
- Write operations are always executed under strict 2PL with exclusive lock

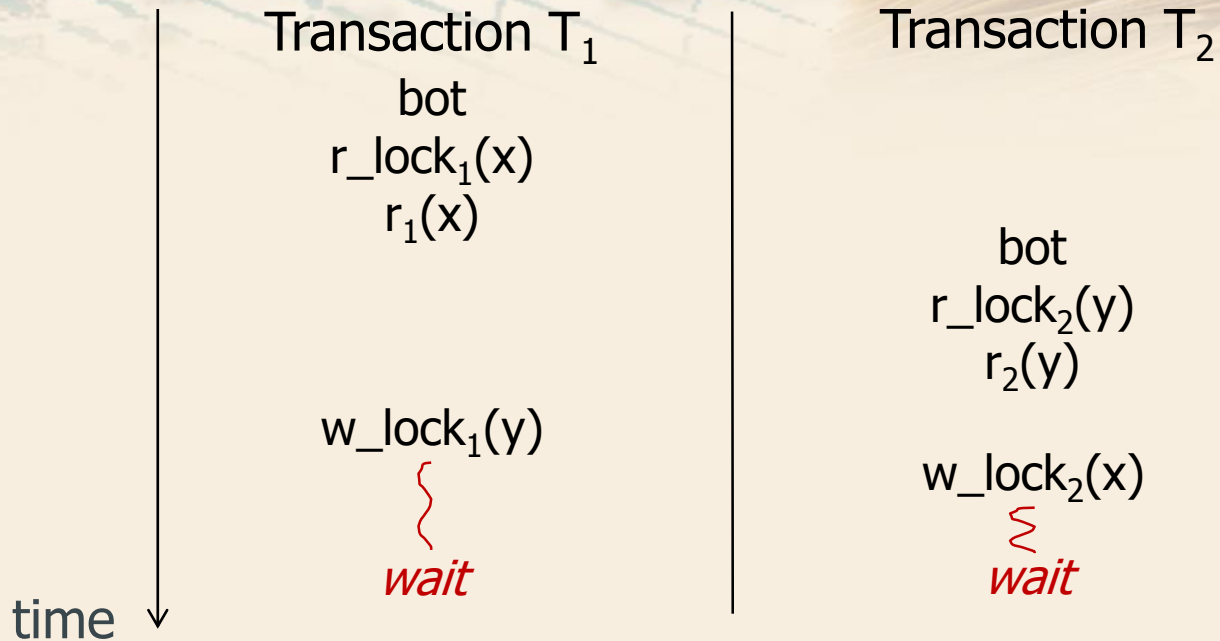




# Database Management Systems

## Deadlock

# Deadlock



➤ Typical situation for concurrent systems managed by means of

- locking
- waiting conditions

# Solving deadlocks

## ➤ Timeout

- the transaction waits for a given time
- after the expiration of the timeout
  - it receives a negative answer and it performs rollback

## ➤ Typically adopted in commercial DBMS

## ➤ Length of the timeout interval

- long
  - long waiting before solving the deadlock
- short
  - overkill, which overloads the system

# Deadlock prevention

## ➤ Pessimistic 2PL

- All needed locks are acquired before the transaction starts
  - not always feasible

## ➤ Timestamp

- only “younger” (or older) transactions are allowed to wait
  - it may cause overkill

# Deadlock detection

➤ Based on the *wait graph*

- nodes are transactions
- an edge represents a waiting state between two transactions



- A cycle in the graph represents a deadlock
- Expensive to build and maintain
  - used in distributed DBMS