

# Comp115: Databases

## Concurrency Control

Instructor: Manos Athanassoulis

# Concurrency Control

**Serializability**      Readings: Chapter 17.1

Two phase locking

Lock management and deadlocks

Locking granularity

Tree locking

Phantoms and predicate locking

# Review

DBMSs support **ACID Transaction semantics**

Concurrency control and Crash Recovery are key components

For Isolation property, serial execution of transactions is safe but slow

- Try to find schedules equivalent to serial execution

# Formal Properties of Schedules

*Serial schedule*: Schedule that does not interleave the actions of different transactions

*Equivalent schedules*: For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule

*Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions

Note: If each transaction preserves consistency, every serializable schedule preserves consistency.

# Conflicting Operations

We need a formal notion of equivalence that can be implemented efficiently

- Base it on the notion of “conflicting” operations

Definition: Two operations **conflict** if:

- They are done by different transactions,
- They are done on the same object,
- And at least one of them is a write

# Conflict Serializable Schedules

Definition: Two schedules are **conflict equivalent** iff:

- They involve the same actions of the same transactions, and
- every pair of conflicting actions is ordered the same way

Definition: Schedule S is **conflict serializable** if:

- S is conflict equivalent to some serial schedule

Note, some “serializable” schedules are NOT conflict serializable

- A price we pay to achieve efficient enforcement

# Conflict Serializability – Intuition

A schedule  $S$  is conflict serializable if:

- You are able to transform  $S$  into a serial schedule by swapping **consecutive non-conflicting** operations of different transactions

*Example:*

$$\begin{array}{ccc}
 R(A) \ W(A) & & R(B) \ W(B) \\
 & R(A) \ W(A) & R(B) \ W(B) \\
 & \equiv & \\
 R(A) \ W(A) \ R(B) \ W(B) & & \\
 & & R(A) \ W(A) \ R(B) \ W(B)
 \end{array}$$

# Conflict Serializability (Continued)

Here's another example:

$R(A)$   $W(A)$   
 $R(A)$   $W(A)$

Serializable or not?

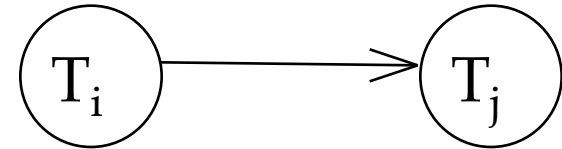
**NOT!**



# Dependency Graph

## Dependency graph:

- One node per transaction
- Edge from  $T_i$  to  $T_j$  if:
  - An operation  $O_i$  of  $T_i$  conflicts with an operation  $O_j$  of  $T_j$  and
  - $O_i$  appears earlier in the schedule than  $O_j$

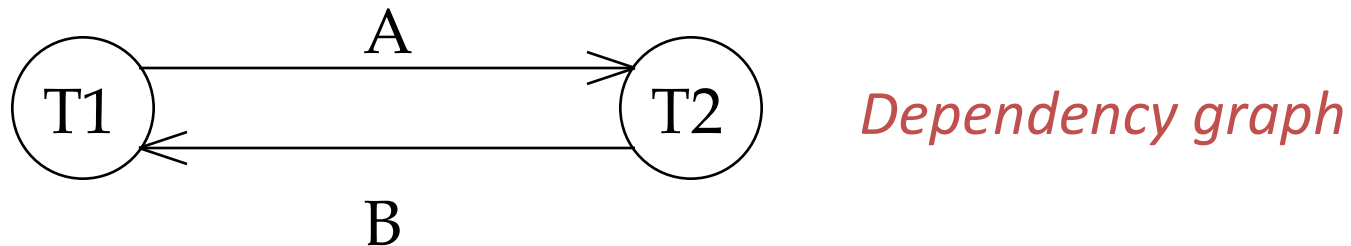


Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic

# Example

A schedule that is not conflict serializable:

T1:	R(A), W(A),	R(B), W(B)
T2:	R(A), W(A), R(B), W(B)	



The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa

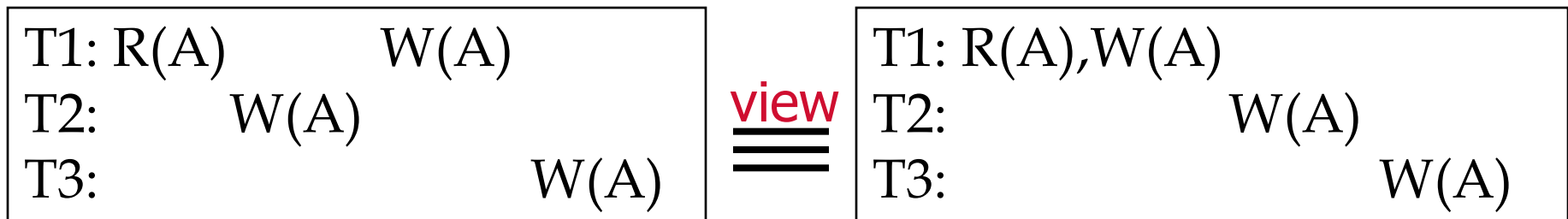
# View Serializability

Alternative (weaker) notion of serializability

Schedules S1 and S2 are **view equivalent** if:

1. If  $T_i$  reads initial value of A in S1, then  $T_i$  also reads initial value of A in S2
2. If  $T_i$  reads value of A written by  $T_j$  in S1, then  $T_i$  also reads value of A written by  $T_j$  in S2
3. If  $T_i$  writes final value of A in S1, then  $T_i$  also writes final value of A in S2

Basically, allows all conflict serializable schedules  
+ “blind writes”



# Notes on Serializability Definitions

View Serializability allows (slightly) more schedules than Conflict Serializability does

- Problem: it is difficult to enforce efficiently

Neither definition allows all schedules that you would consider “serializable”

- This is because they don’t understand the meanings of the operations or the data

In practice, Conflict Serializability is used, because it can be enforced efficiently

- To allow more concurrency, some special cases do get handled separately, such as travel reservations

# Concurrency Control

Serializability

**Two phase locking**      Readings: Chapter 17.1

Lock management and deadlocks

Locking granularity

Tree locking

Phantoms and predicate locking

# Two-Phase Locking (2PL)

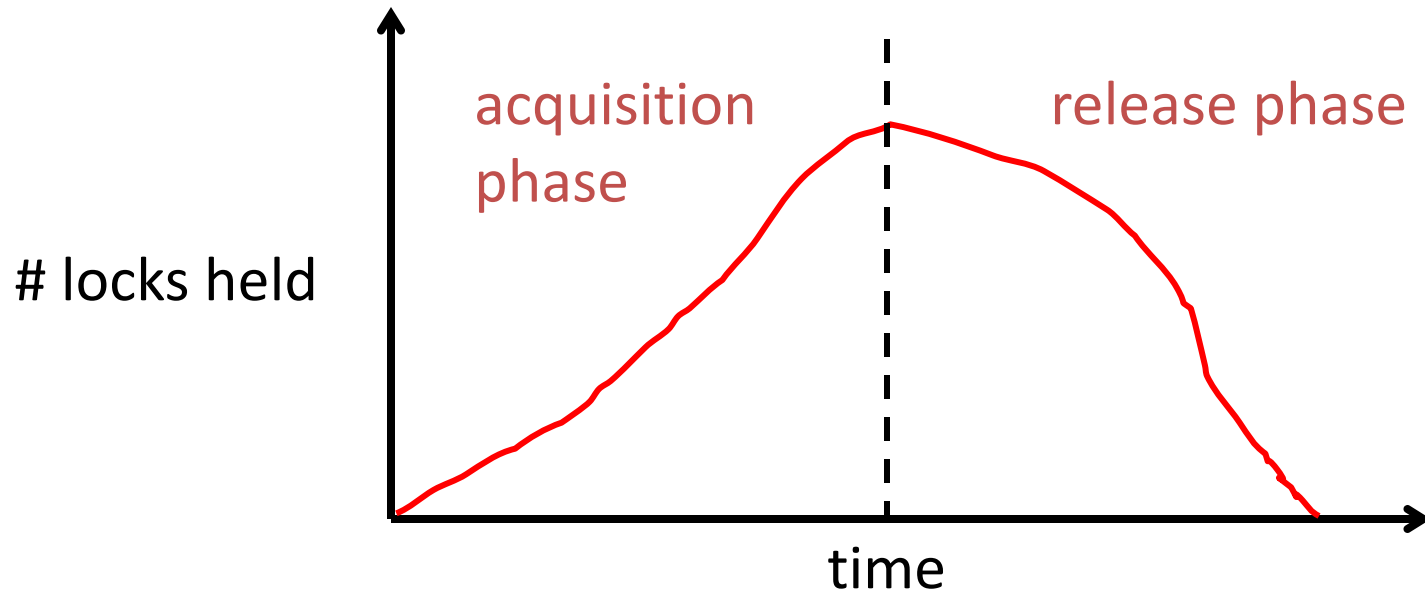
Lock  
Compatibility  
Matrix

	S	X
S	✓	–
X	–	–

## Locking Protocol

- Each transaction must obtain an S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing
- A transaction can not request additional locks once it releases any locks
- Thus, there is a “growing phase” followed by a “shrinking phase”

# Two-Phase Locking (2PL)



2PL on its own is sufficient to guarantee conflict serializability (i.e., schedules whose precedence graph is acyclic), but, it is subject to **Cascading Aborts**

# Strict 2PL

Problem: Cascading Aborts

Example: rollback of T1 requires rollback of T2!

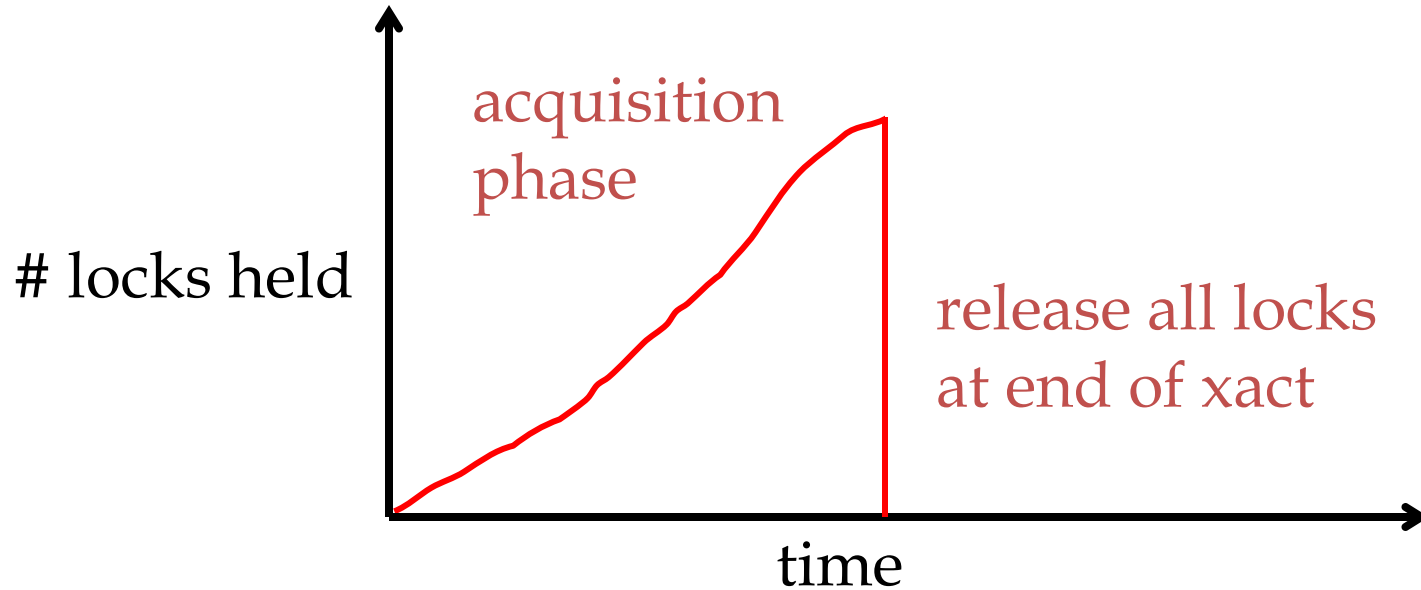
T1:	R(A), W(A),	R(B), W(B), Abort
T2:	R(A), W(A)	

To avoid Cascading aborts, use Strict 2PL

Strict Two-phase Locking (Strict 2PL) Protocol:

- Same as 2PL, except:
- All locks held by a transaction are released only when the transaction completes





Allows only conflict serializable schedules, but it is actually stronger than needed for that purpose

In effect, “shrinking phase” is delayed until

- Transaction has committed (commit log record on disk), or
- Decision has been made to abort the transaction (locks can be released after rollback)

# Non-2PL, A= 1000, B=2000, Output =?

Lock_X(A)	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Unlock(A)	
	Read(A)
	Unlock(A)
	Lock_S(B)
Lock_X(B)	
	Read(B)
	Unlock(B)
	PRINT(A+B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	

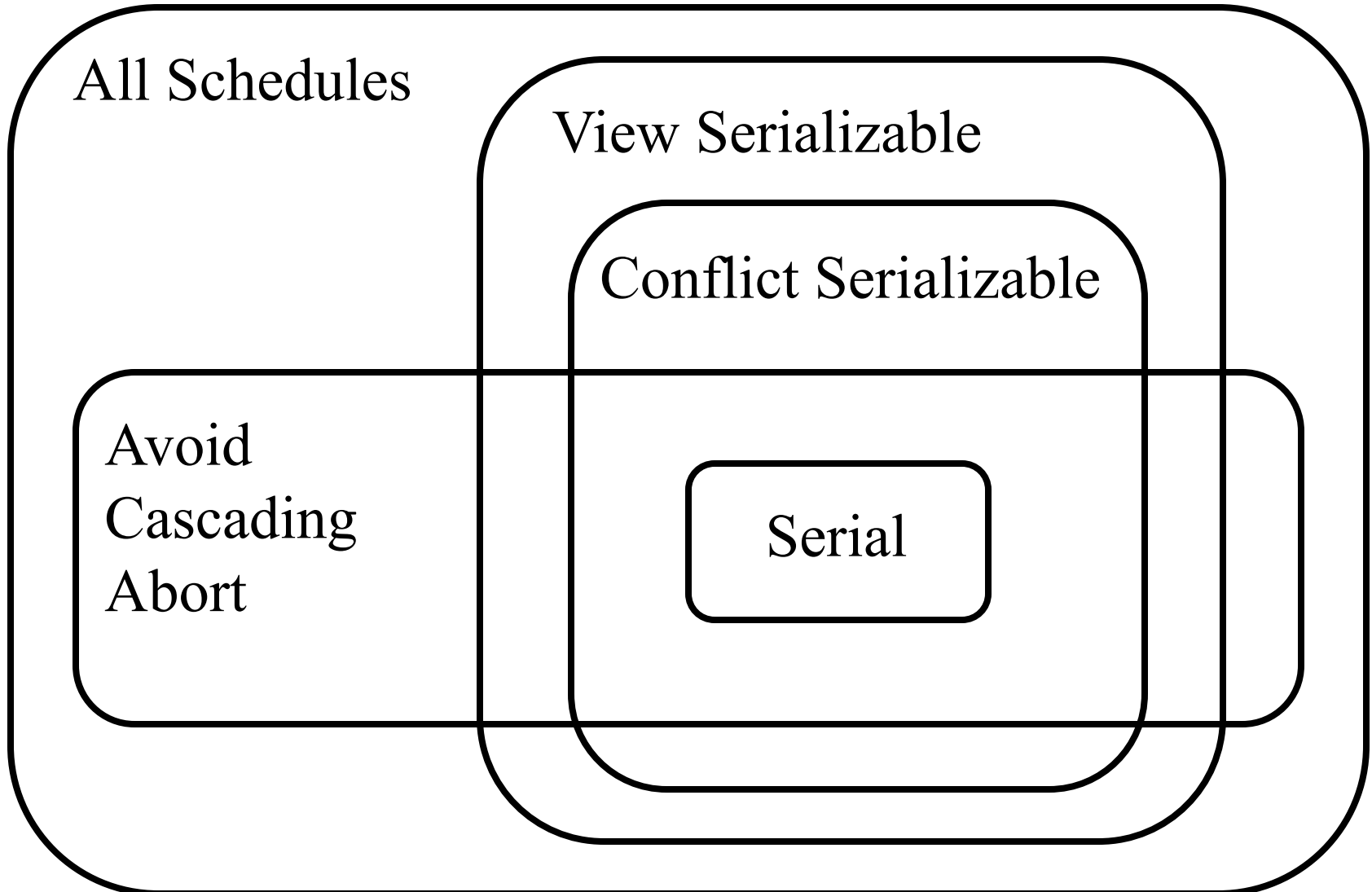
# 2PL, A= 1000, B=2000, Output =?

Lock_X(A)	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B)	
Unlock(A)	
	Read(A)
	Lock_S(B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	Unlock(A)
	Read(B)
	Unlock(B)
	PRINT(A+B)

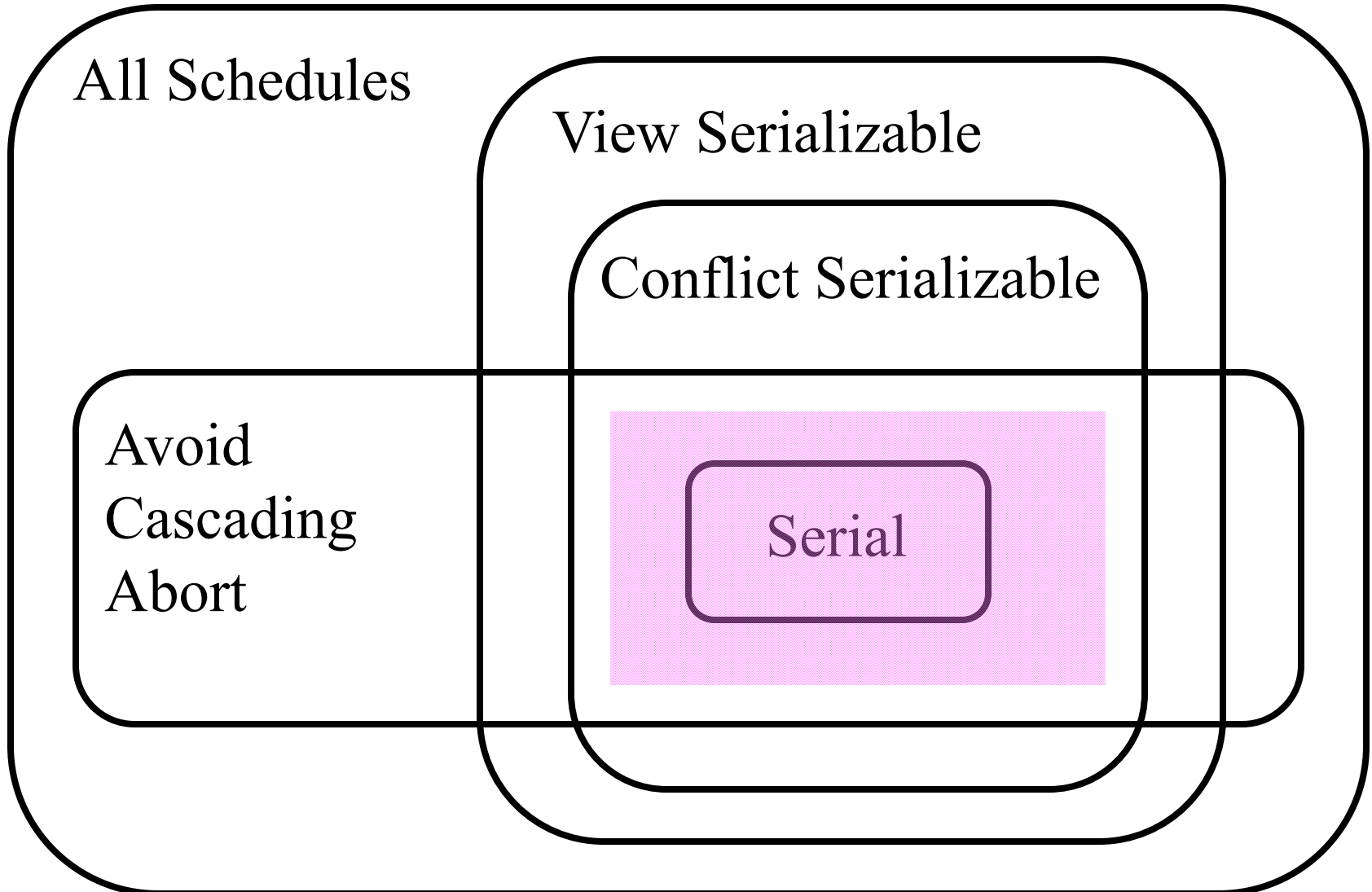
# Strict 2PL, A= 1000, B=2000, Output =?

Lock_X(A)	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B)	
Read(B)	
B := B +50	
Write(B)	
Unlock(A)	
Unlock(B)	
	Read(A)
	Lock_S(B)
	Read(B)
	PRINT(A+B)
	Unlock(A)
	Unlock(B)

# Venn Diagram for Schedules



## Q: Which schedules does Strict 2PL allow?



# Two phase locking: Summary

Locks implement the notions of conflict directly

2PL has:

- Growing phase where locks are acquired and no lock is released
- Shrinking phase where locks are released and no lock is acquired

Strict 2PL requires all locks to be released at once, when transaction ends

# Concurrency Control

Serializability

Two phase locking

**Lock management and deadlocks**

Readings: Chapters 17.2-17.4

Locking granularity

Tree locking

Phantoms and predicate locking



# Lock Management

Lock and unlock requests handled by the Lock Manager

Lock Manager contains an entry for each currently held lock

Lock table entry:

- Pointer to list of transactions currently holding the lock
- Type of lock held (shared or exclusive)
- Pointer to queue of lock requests

# Lock Management, continued

Basic operation: when lock request arrives  
see if any other transaction holds a  
conflicting lock

- If not, create an entry and grant the lock
- Else, put the requestor on the wait queue

Lock upgrade: transaction that holds a  
shared lock can be upgraded to hold an  
exclusive lock

Two-phase locking is simple enough, right?



# Deadlocks

**Deadlock:** Cycle of transactions waiting for locks to be released by each other

Two ways of dealing with deadlocks:

- Deadlock **prevention**
- Deadlock **detection**

Many systems just “punt” and use Timeouts

- What are the dangers with this approach?

# Deadlock Detection

Create a **waits-for graph**:

- Nodes are transactions
- Edge from  $T_i$  to  $T_j$  if  $T_i$  is waiting for  $T_j$  to release a lock

Periodically check for cycles in waits-for graph

# Deadlock Detection (Continued)

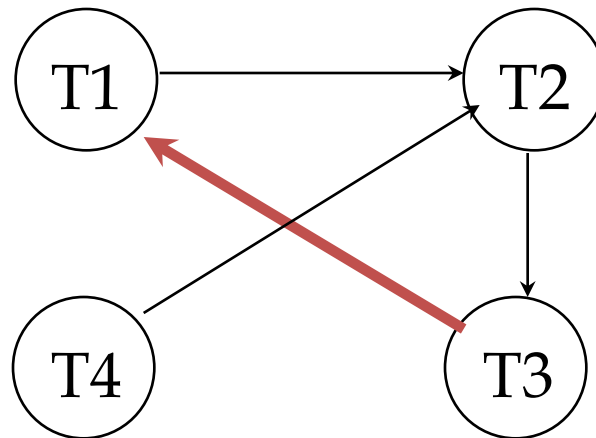
Example:

T1: S(A), S(D), S(B)

T2: X(B)

T3: S(D), S(C), X(A)

T4: X(B)



# Deadlock Prevention

Assign priorities based on timestamps

Say  $T_i$  wants a lock that  $T_j$  holds

Two policies are possible:

**Wait-Die:** If  $T_i$  has higher priority,  $T_i$  waits for  $T_j$ ;  
otherwise  $T_i$  aborts

**Wound-wait:** If  $T_i$  has higher priority,  $T_j$  aborts;  
otherwise  $T_i$  waits

Why do these schemes guarantee no deadlocks?

Important detail: If a transaction re-starts, make sure it gets its original timestamp. -- Why?

# Deadlocks: summary

The lock manager keeps track of the locks issued

Deadlock is a cycle of transactions waiting for locks to be released to each other

Deadlocks may arise and can be:

- Prevented, e.g. using timestamps
- Detected, e.g. using waits-for graphs



# Concurrency Control

Serializability

Two phase locking

Lock management and deadlocks

**Locking granularity**      *Readings: Chapter 17.5.2*

Tree locking

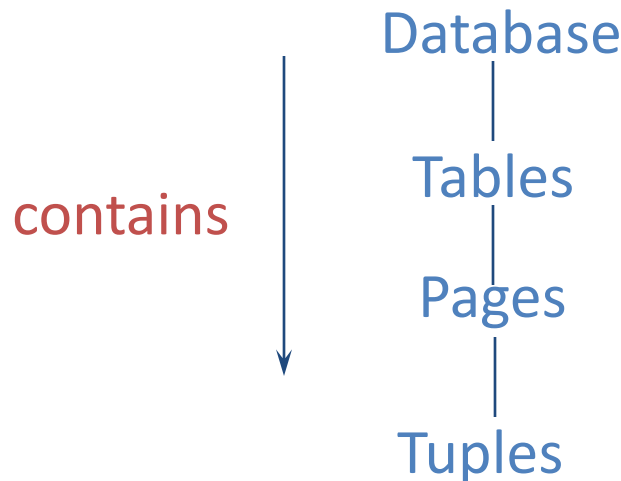
Phantoms and predicate locking

# Multiple-Granularity Locks

Hard to decide what granularity to lock  
(tuples vs. pages vs. tables)

Shouldn't have to make same decision for all  
transactions!

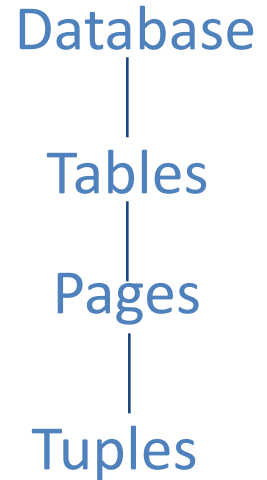
Data “containers” are nested:



# Solution: New Lock Modes, Protocol

Allow transaction to lock at each level, but with a special protocol using new “**intention**” locks:

Still need S and X locks, but before locking an item, transaction must have proper intension locks on all its ancestors in the granularity hierarchy



**IS** – Intent to get S lock(s) at finer granularity

**IX** – Intent to get X lock(s) at finer granularity

**SIX mode**: Like S & IX at the same time. Why is it useful?

# Multiple Granularity Lock Protocol

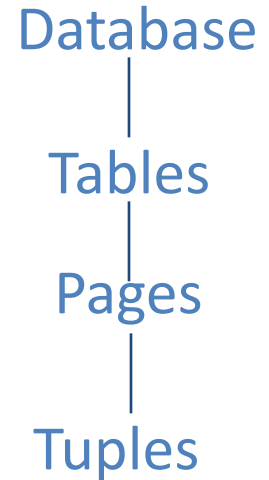
Each transaction starts from the root of the hierarchy

To get S or IS lock on a node, must hold IS or IX on parent node

- What if transaction holds SIX on parent? S on parent?

To get X or IX or SIX on a node, must hold IX or SIX on parent node

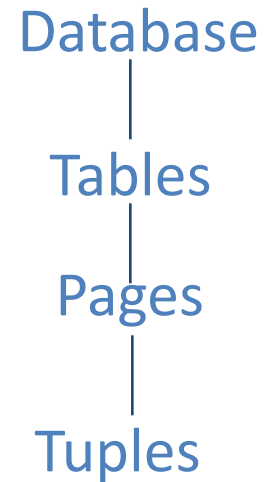
Must release locks in bottom-up order



Protocol is equivalent to directly setting locks at the leaf levels of the hierarchy.

# Lock Compatibility Matrix

	IS	IX	SIX	S	X
IS	✓	✓	✓	✓	-
IX	✓	✓	-	-	-
SIX	✓	-	-	-	-
S	✓	-	-	✓	-
X	-	-	-	-	-

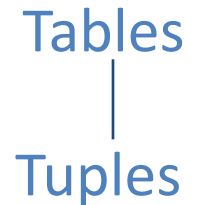


**IS** – Intent to get S lock(s) at finer granularity

**IX** – Intent to get X lock(s) at finer granularity

**SIX mode**: S & IX at the same time

# Examples – 2 level hierarchy



T1 scans R, and updates a few tuples:

- T1 gets an SIX lock on R, then get X lock on tuples that are updated

T2 uses an index to read only part of R:

- T2 gets an IS lock on R, and repeatedly gets an S lock on tuples of R

T3 reads all of R:

- T3 gets an S lock on R
- OR, T3 could behave like T2
- We can use **lock escalation** to decide
- Lock escalation dynamically asks for coarser-grained locks when too many low level locks acquired

	IS	IX	SIX	S	X
IS	✓	✓	✓	✓	
IX	✓	✓			
SIX	✓				
S	✓			✓	
X					

# Multiple granularity locking: Summary

Allows flexibility for each transaction to choose locking granularity independently

Introduces hierarchy of objects

Introduces intention locks

# Concurrency Control

Serializability

Two phase locking

Lock management and deadlocks

Locking granularity

**Tree locking**      *Readings: Chapter 17.5.2*

Phantoms and predicate locking



# Locking in B+ Trees

Tree-based indexes present a potential concurrency bottleneck:

If you ignore the tree structure & just lock pages while traversing the tree, following 2PL

- Root node (and many higher level nodes) become bottlenecks because every tree access begins at the root

How can we efficiently lock a particular leaf node?

- Don't confuse this with multiple granularity locking!

# Two Useful Observations

1. In a B+Tree, higher levels of the tree only direct searches for leaf pages
2. For inserts, a node on a path from root to modified leaf must be locked (in X mode, of course), only if a split can propagate up to it from the modified leaf (Similar point holds w.r.t. deletes)

We can exploit these observations to design efficient locking protocols that guarantee serializability *even though they violate 2PL*

# A Simple Tree Locking Algorithm: “crabbing”

**Search:** Start at root and go down; repeatedly, S lock child then unlock parent

**Insert/Delete:** Start at root and go down, obtaining X locks as needed. Once child is locked, check if it is safe:

- If child is safe, release all locks on ancestors

**Safe node:** Node such that changes will not propagate up beyond this node.

- Insertions: Node is not full
- Deletions: Node is not half-empty

# Example

ROOT



A

- Do:**

  - 1) Search 38\*
  - 2) Delete 38\*
  - 3) Insert 45\*
  - 4) Insert 25\*



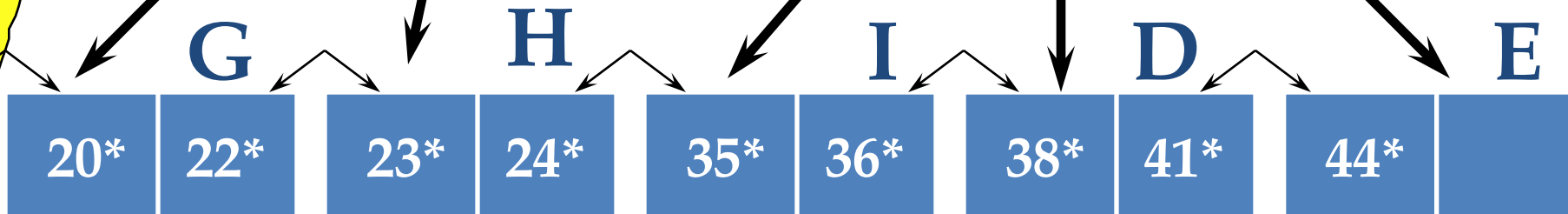
B



F



C



# Concurrency Control

Serializability

Two phase locking

Lock management and deadlocks

Locking granularity

Tree locking

**Phantoms and predicate locking**

Readings: Chapter 17.5.1

## Dynamic Databases – The “Phantom” Problem

If we relax the assumption that the DB is a fixed collection of objects, even Strict 2PL (**on individual items**) will not ensure serializability:

Consider T1 – “Find oldest sailor”

- T1 locks all records, and finds oldest sailor (say, *age* = 71)
- Next, T2 inserts a new sailor; *age* = 96 and commits
- T1 (within the same transaction) checks for the oldest sailor again and finds sailor aged 96!

The sailor with age 96 is a “phantom tuple” from T1’s point of view --- first it’s not there then it is

No serial execution where T1’s result could happen!

# The “Phantom” Problem – ex. 2

Consider T3 – “Find oldest sailor for each rating”

- T3 locks all pages containing sailor records with *rating* = 1, and finds oldest sailor (say, *age* = 71)
- Next, T4 inserts a new sailor; *rating* = 1, *age* = 96
- T4 also deletes oldest sailor with *rating* = 2 (and, say, *age* = 80), and commits
- T3 now locks all pages containing sailor records with *rating* = 2, and finds oldest (say, *age* = 63)

T3 saw only part of T4’s effects!

No serial execution where T3’s result could happen!

# The Problem

T1 and T3 implicitly assumed that they had locked the set of all sailor records satisfying a predicate

- Assumption only holds if no sailor records are added while they are executing!
- Need some mechanism to enforce this assumption  
(Index locking and predicate locking)

Examples show that conflict serializability on reads and writes of individual items guarantees serializability only if the set of objects is fixed!



# Predicate Locking

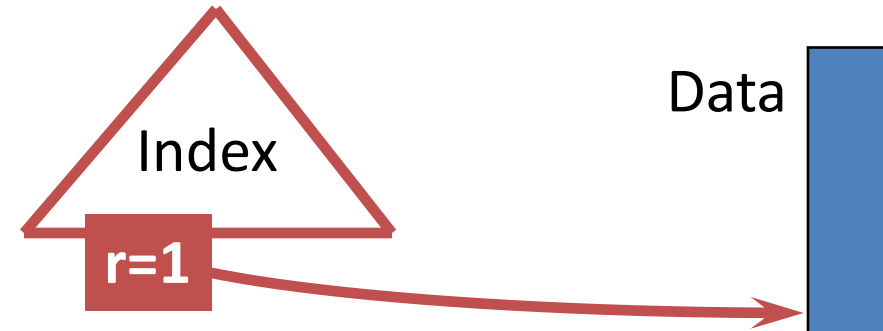
Grant lock on all records that satisfy some logical predicate, e.g. *age > 2\*salary*

Index locking is a special case of predicate locking for which an index supports efficient implementation of the predicate lock

- What is the predicate in the sailor example?

In general, predicate locking has a lot of locking overhead

# Index Locking



If there is a dense index on the *rating* field using Alternative (2), T3 should lock the index page containing the data entries with *rating* = 1

- If there are no records with *rating* = 1, T3 must lock the index page where such a data entry *would* be, if it existed!

If there is no suitable index, T3 must obtain:

1. A lock on every page in the table file  
→ prevent a record's rating from being changed to 1

AND

2. The lock for the file itself  
→ prevent records with *rating* = 1 from being added or deleted

# Transaction Support in SQL-92

SERIALIZABLE – No phantoms, all reads repeatable, no “dirty” (uncommitted) reads

REPEATABLE READS – phantoms may happen

READ COMMITTED – phantoms and unrepeatable reads may happen

READ UNCOMMITTED – all of them may happen.

# Phantom problem: Summary

If database objects can be added/removed, need to guard against **Phantom Problem**

Must lock logical sets of records

Efficient solution: index locking