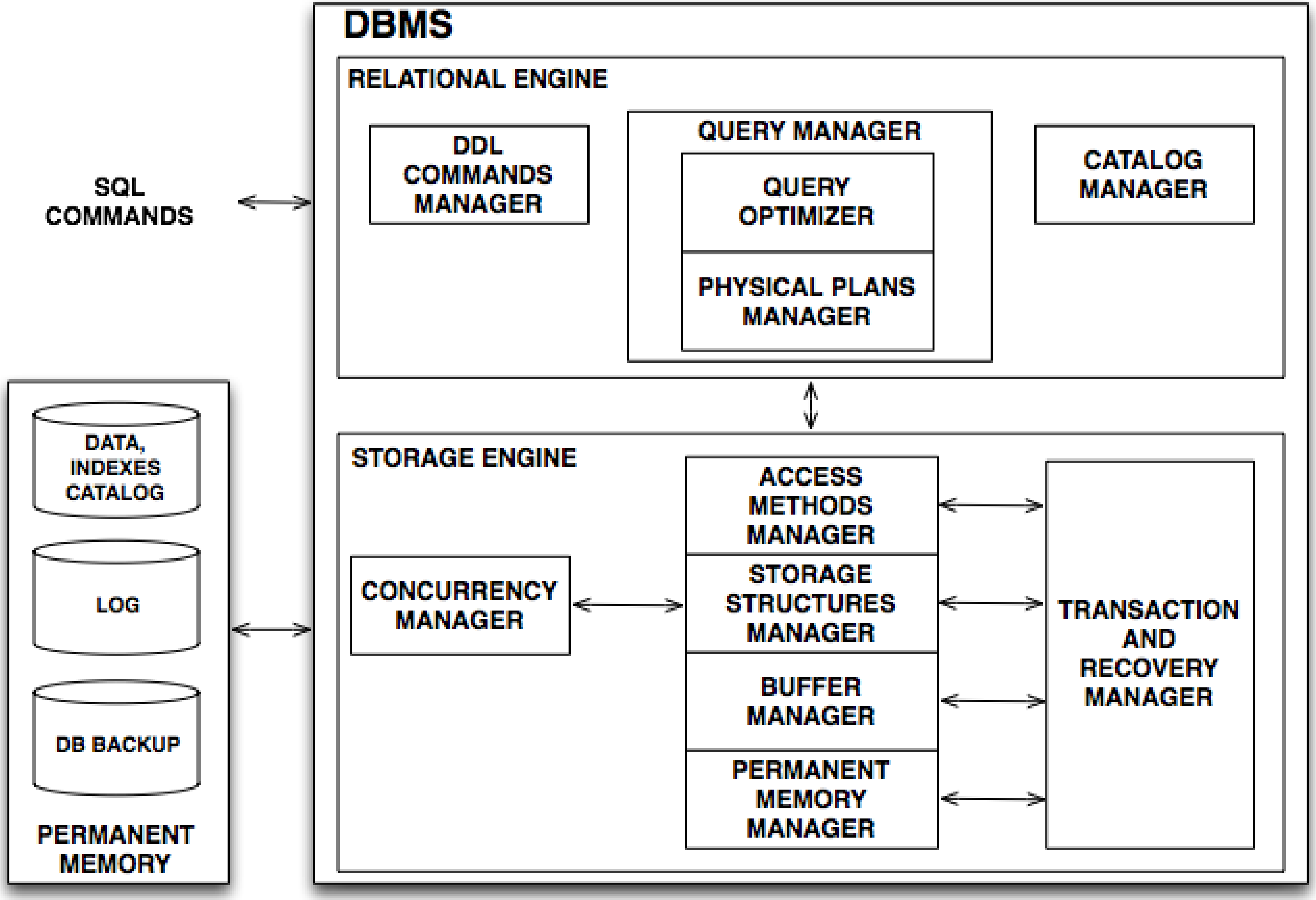
DBMS ARCHITECTURE



Concurrency

- Consider two ATMs running in parallel

T1	T2
r1[x]	
x:=x-250	r2[x]
	x:=x-250
w[x]	
commit	
	w[x]
	commit

- We need a *concurrency manager*

Examples of interference

Lost Updates

T1:	r[x=100]		w[x:=600]	
T2:		r[x=100]		w[x:=500]

Dirty Read

T1:	r[x=200] w[x:=100]		abort
T2:		r[x=100] r[y=500]	

Unrepeatable Read

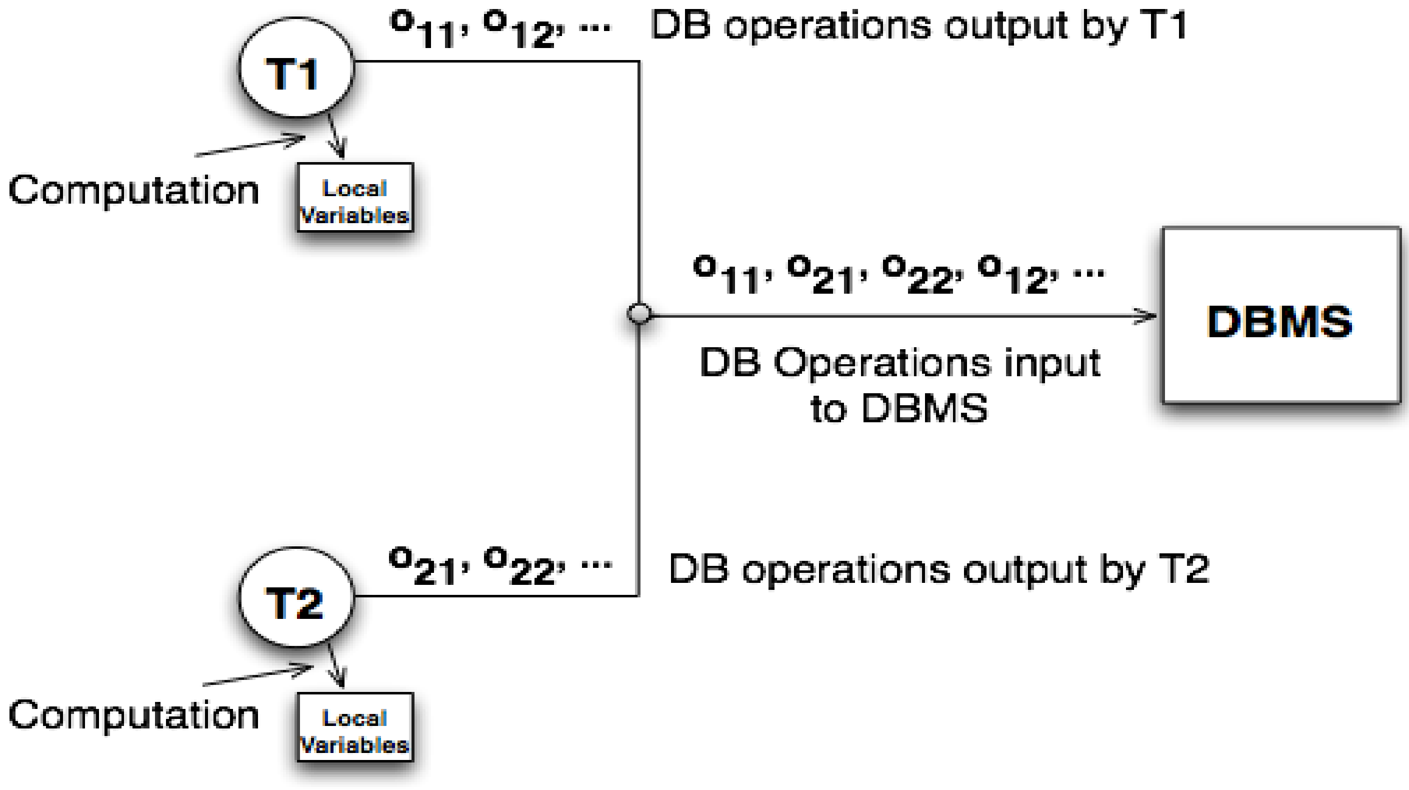
T1:	r[x=100]		r[x=500]
T2:		w[x:=500]	

Seriality and serializability

- *Definition*: A concurrent execution of a set transactions {T_1 , …, T_n } is ***serial*** if, for every pair of transactions T_i and T_j, all the operations of T_i are executed before any of the operations of T_j or vice versa.

Seriality and serializability

- A serial execution is impractical – we need interleaving



Seriality and serializability

- *Definition*: A ***concurrent*** execution of a set transactions $\{T_1, \ldots, T_n\}$ is ***serializable*** if it has the same ***effect*** on the database as some serial execution of the same transactions.

Serializability theory

- Goal of the **concurrence manager** (or **scheduler**): providing concurrency and serializability.
- Correctness of a **scheduler** is proved using a **theory of serializability,** based on:
  - Transactions
  - History of the concurrent execution of a set of T
  - Equivalence relation between histories
  - Serializable histories
  - Properties of the histories generated by a scheduler

Transactions and operations

- Assume a unbounded set of locations x, y, $z \in X$
- Transaction T_i: sequence of operations $r_i[x]$, $w_i[x]$ on elements of X that terminates either with c_i (commit) or a_i (abort)
- We ignore data creation and complex operations such as insertion in a list

Example

The execution of the transaction

```
program T;
var i, j:integer;
begin
    i:= read(x);
    j:= read( y);
    j := i + j;
    write(j,x);
end;
end {program}.
```

is seen by the DBMS as a sequence of operations:
r[x] r[y] w[x] c

History of a set of transactions

Definition Let **T** = $\{T_1, T_2, \ldots, T_n\}$ be a set of transactions.

A **history H** on **T** is an ordered set of operations such that:

1. The operations of **H** are those of T_1, T_2, …, T_n;
2. **H** preserves the ordering among the operations of the same transaction.

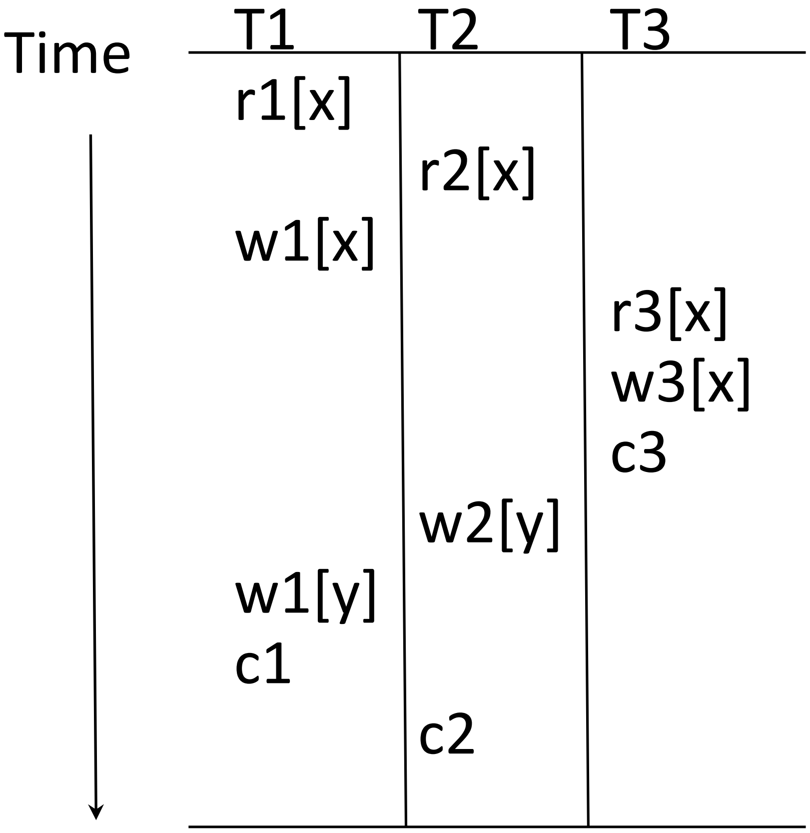
Example of a history

T1 = r1[x] w1[x] w1[y] c1

T2 = r2[x] w2[y] c2

T3 = r3[x] w3[x] c3

H1= r1[x] r2[x] w1[x] r3[x] w3[x]
c3 w2[y] w1[y] c1 c2



Time

T1	T2	T3
r1[x]		
	r2[x]	
w1[x]		
		r3[x]
		w3[x]
		c3
	w2[y]	
w1[y]		
c1		
	c2	

H1

A possible definition of equivalent histories

- **Definition** A history **S** is **serializable** if it is equivalent to a serial history
- **Definition** Two histories **H** and **L** are **equivalent** if
  - they are defined on the same set of transactions,
  - they produce the same effect on the DB (same final state)

A stronger definition

- A simpler notion of equivalence it is used, which is easier to check, based on the notion of **operations in conflict**
- **Definition** Two operations are in **conflict** if
  - they belong to **different transactions**,
  - they are on the **same data**,
  - one of them is a **write operation**
- **Intuition**: Two operation o1 and o2 commute if o1-o2 has the same effect and result as o2-o1. Two perations conflict if they may not commute

C-equivalent histories

- **Definition** Two histories **H** and **L** are **c-equivalent** with respect to **operations in conflict** if:
  - **H** and **L** are defined on the same set of transactions
  - Evey pair of operations in conflict of **committed** transactions are in the same order
- Therefore, each read operation in **H** reads the same data in **L**, and the last data written in **H** and **L** are the same.

C-equivalent histories

H1= r1[x] r2[x] w1[x] r3[x] w3[x] c3 w2[y] w1[y] c1 c2

T1	T2	T3
r1[x]		
	r2[x]	
w1[x]		
		r3[x]
		w3[x]
		c3
	w2[y]	
w1[y]		
c1		
	c2	

H1

T1	T2	T3
	r2[x]	
	w2[y]	
	c2	
r1[x]		
w1[x]		
		r3[x]
		w3[x]
		c3
w1[y]		
c1		

H2 c-equivalent to H1?

YES

T1	T2	T3
r1[x]		
	r2[x]	
w1[x]		
		r3[x]
		w3[x]
		c3
w1[y]		
	w2[y]	
c1		
	c2	

H3 c-equivalent to H1 ?

NO

Serializability and c-serializability

- A history H on the set T={T1,T2,…,Tn } is serial if represent a serial execution of T1,T2,…,Tn.
- *Definition*: A history H on the set T={T1,T2,…,Tn } is ***serializable*** if it has the same ***effect*** on the database as some serial execution of the same transactions.
- *Definition:* A history H on the set T={T1,T2,…,Tn } is ***c-serializable*** if it is ***c-equivalent*** to a serial history on {T1,T2,…,Tn }.
- C-serializable implies serializable

Serializability and c-serializability

T1	T2	T3
r1[x]		
	r2[x]	
w1[x]		
		r3[x]
		w3[x]
		c3
	w2[y]	
w1[y]		
c1		
	c2	

H1

T1	T2	T3
	r2[x]	
	w2[y]	
	c2	
r1[x]		
w1[x]		
		r3[x]
		w3[x]
		c3
w1[y]		
c1		

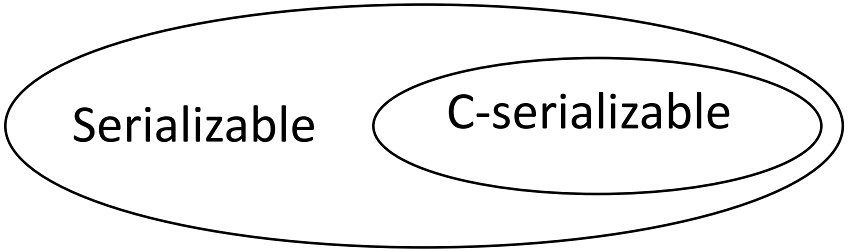
H2 c-equivalent to H1

T1	T2	T3
	r2[x]	
	w2[y]	
	c2	
r1[x]		
w1[x]		
w1[y]		
c1		
		r3[x]
		w3[x]
		c3

Serial history c-equivalent to H2 and H1

Serializability and c-serializability

- Some serializable histories are not c-serializable



T1	T2	T3
r1[y]		
	w2[y]	
	w2[x]	
	c2	
w1[x]		
c1		
		w3[x]
		c3

- Serial history: T1 , T2, T3
- The final DB state is the same.

Using the theory

- We define a scheduling algorithm
- Prove that it only produces c-serializable histories
- Hence, it only produces serializable histories

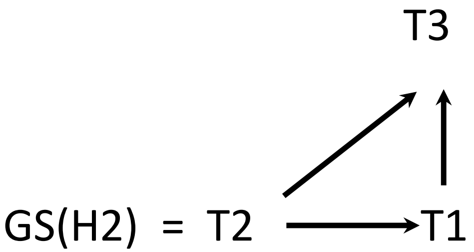
Serialization graph

- We can decide if a schedule is c-serializable by looking at its serialization graph
- **Definition** Given a history H on T = {T_1, T_2, ..., T_n}, the serialization graph of H, SG(H), is a direct graph whose nodes are the committed transaction of H, and arc from T_i to T_j ($i \neq j$) if an operation of T_i precedes and is in conflict with an operation of T_j.

Example

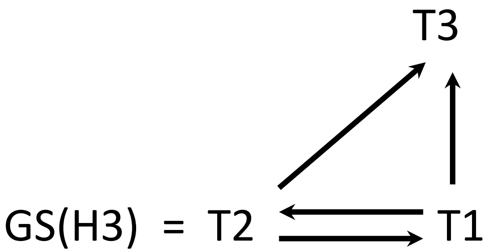
T1	T2	T3
	r2[x]	
	w2[y]	
	c2	
r1[x]		
w1[x]		
		r3[x]
		w3[x]
		c3
w1[y]		
c1		

History H2



T1	T2	T3
r1[x]		
	r2[x]	
w1[x]		
		r3[x]
		w3[x]
		c3
w1[y]		
	w2[y]	
c1		
	c2	

History H3

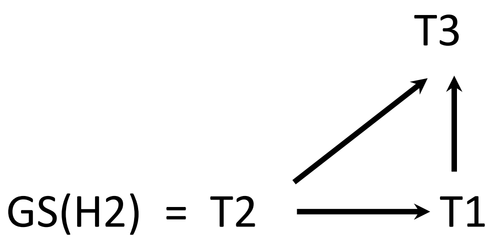


T1	T2
r1[x]	
	r2[x]
w1[x]	
	w2[x]
c1	
	c2

History H4

Serializability theorem

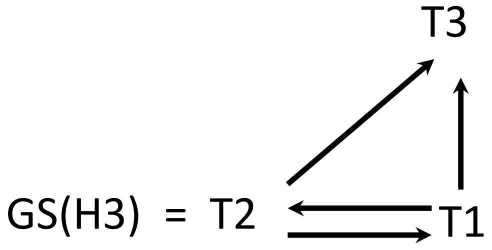
- Serializability theorem: H is c-serializable if and only if the corresponding serialization graph is acyclic.



T3

GS(H2) = T2

T1

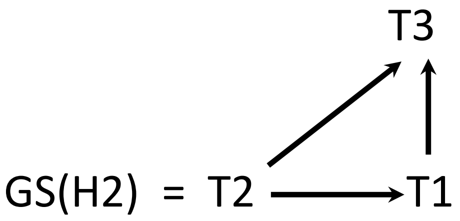


T3

GS(H3) = T2

T1

- If SG is acyclic, a serial schedule can be obtained with a topological ordering on the graph



T3

GS(H2) = T2

T1

Serial history? T_2 , T_1, T_3

Strict 2PL protocol

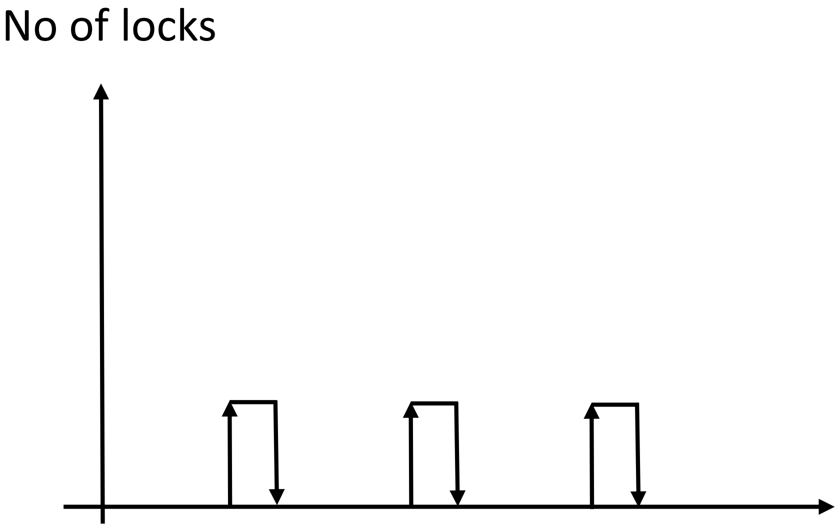
- Strict two-phase locking algorithm (pessimistic approach): the most used scheduling protocol
- A protocol between transactions T_i and a scheduler S:
  - Before acting on X, T_i asks S for the corresponding lock
  - Different transactions are not given conflicting locks by S
  - T_i relases all its locks upon termination, and never before.

Lock vs Strict 2PL

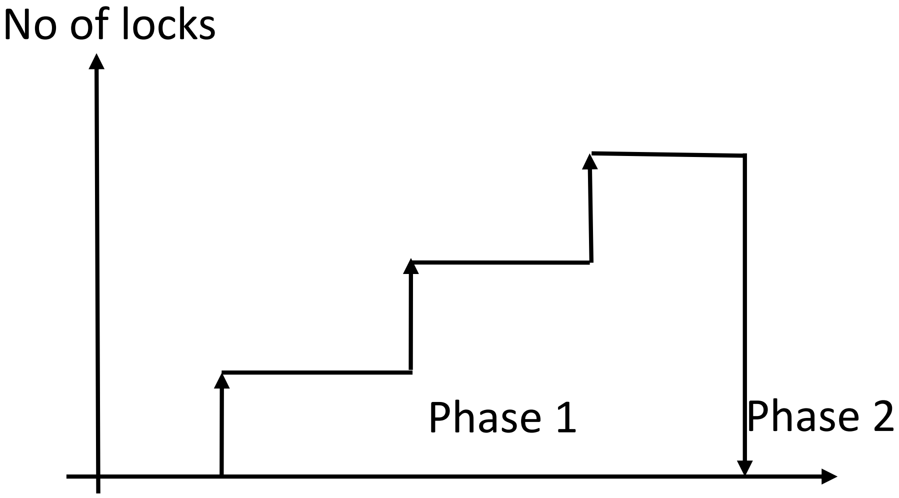
↑ Obtain lock

↓ Release lock

Lock in concurrent programming



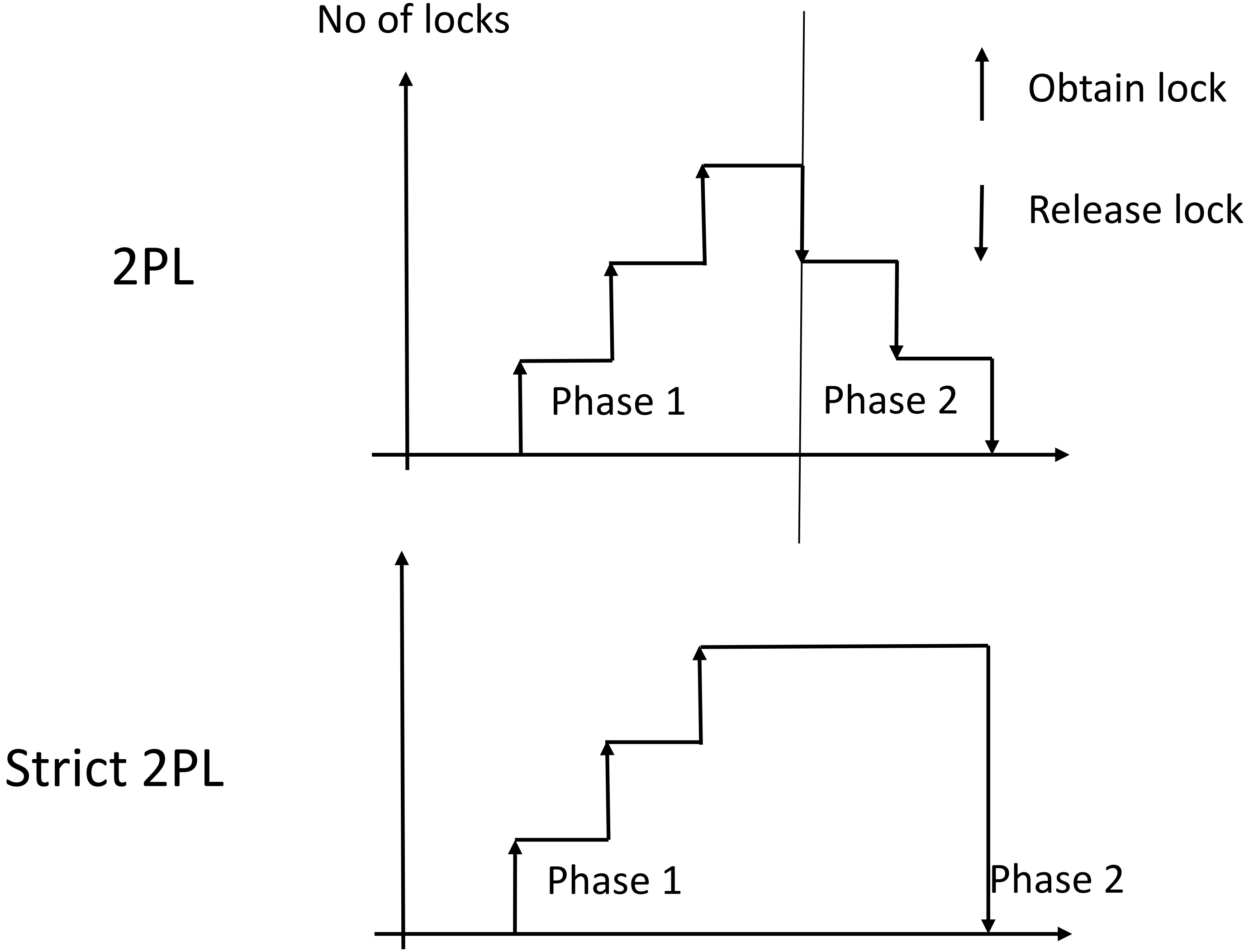
Strict 2PL



Strict 2PL protocol and 2PL

- Strict 2PL:
    1. Before acting on X, T_i asks S for the corresponding lock
    2. Different transactions are not given conflicting locks by S
    3. T_i relases all its locks upon termination, and never before
- Two Phase Locks
    3. After a lock has been released by T_i, T_i will not acquire any new lock
- 2PL suffers the *cascading abort* problem

2PL vs Strict 2PL



Lock modes

- RW – 2PL: two lock modes for each item, Shared (S or R) and Exclusive (X or W)
- Before reading, ask for an S lock. Before writing, ask for an X lock
- Compatibility matrix:

	S	X
S	Yes	No
X	No	No

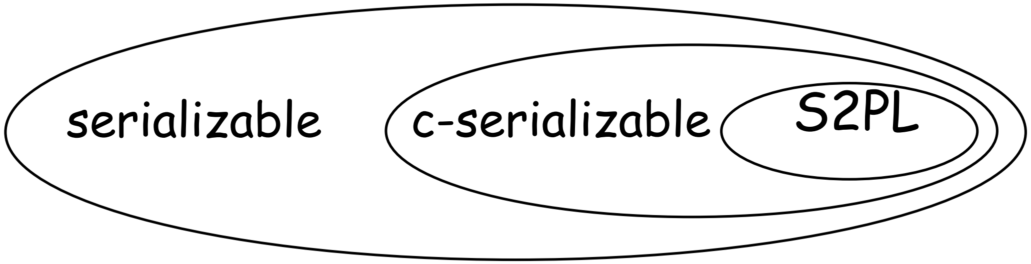
- Richer sets of modes are often used in practice

Implementing the protocol

- A *scheduler* keeps a set of locks – that is, triples (T,mode,x) where mode$\in${S,X} (hashed on *x*):
- When a transaction asks for a lock on :
  - If it possible, the lock is assigned
  - If it is not possible, the transaction is suspended in a wait queue (hashed on *x*)
- When a transaction commits / aborts:
  - All of its locks are released
  - Waiting transactions are notified, with some policy
- The scheduler detects (or prevents) the deadlocks

Strict 2PL and serializability

- Theorem: A strict 2PL schedule is c-serializable



T1	T2	T3
r1[x]		
w1[x]		
	r2[x]	
	w2[x]	
		r3[y]
w1[y]		
c1		
	c2	
		c3

Strict 2PL history

No locks

t1	**t2**	**t3**
r1[x]		
w1[x]		
	r2[x]	
	w2[x]	
		r3[y]
w1[y]		
c1		
	c2	
		c3

S2PL scheduler

t1	**t2**	**t3**
rl[x], **r1[x]**		
wl[x],**w1[x]**		
	rl[x]*	
		rl[y],**r3[y]**
wl[y]*		
		c3,u[y]
wl[y],**w1[y]**		
c1,u[x,y]		
	rl[x],**r2[x]**	
	wl[x],**w2[x]**	
	c2,u[x]	

S2PL history

t1	**t2**	**t3**
r1[x]		
w1[x]		
		r3[y]
		c3
w1[y]		
c1		
	r2[x]	
	w2[x]	
	c2	

SG = t3 → t1 → t2

denied lock requests are marked with*

Deadlocks

- Strict two-phase locking is simple, but the scheduler needs a strategy to manage deadlocks.
- T_1: w1[X], w1[Y], ... T_2: w2[Y], w2[X], ...

T_1	T_2
xl[X]	
w_1[X]	
	xl[Y]
	w_2[Y]
xl[Y] *	xl[X] *

Deadlock !

Deadlocks

- The deadlock problem can be solved with two techniques:
  - Deadlock detection and recovery
  - Deadlock prevention

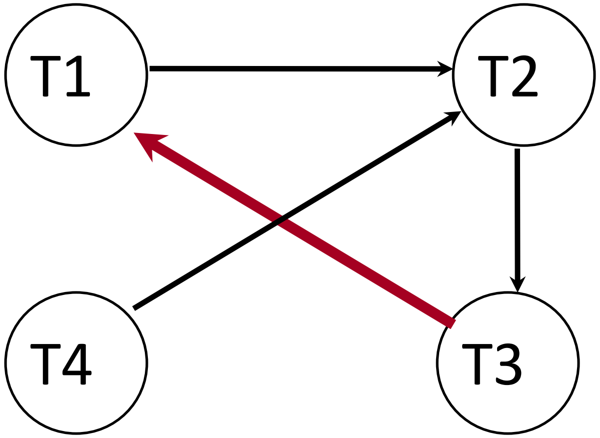
Deadlock detection

- Wait-for graph G = (V, E):
  - V: Vertexes are the active transactions Ti
  - E: Arc Ti -> Tj means that Ti is waiting for a data item locked by Tj
  - Arcs are added and removed when locks are granted and releases
  - A deadlock is present if there is a cycle in the graph
  - A transaction inside the cycle is aborted and restarted
- Otherwise: timeouts

Example of deadlock detection

For simplicity, the lock requests only are shown, and those with * are suspended

T1:	rl[A], rl[D],		rl[B] *,				
T2:		wl[B],			wl[C]*,		
T3:				rl[D], rl[C],			wl[A]*,
T4:						wl[B]*,	



Cycle !

Deadlock prevention

- Each transaction T_i is given a time stamp when it starts and it can wait only
    - for a younger transaction T_j (wait-die) OR
    - for an older transaction T_j (wound-wait)
    - otherwise the younger transaction aborts (dies).
- The aborted T is always the younger, which then later restarts with the same time stamp: no starvation
- No deadlocks

Wait-die

A T may only wait for a **younger one**.

Suppose T_i requests a data item currently held by T_j

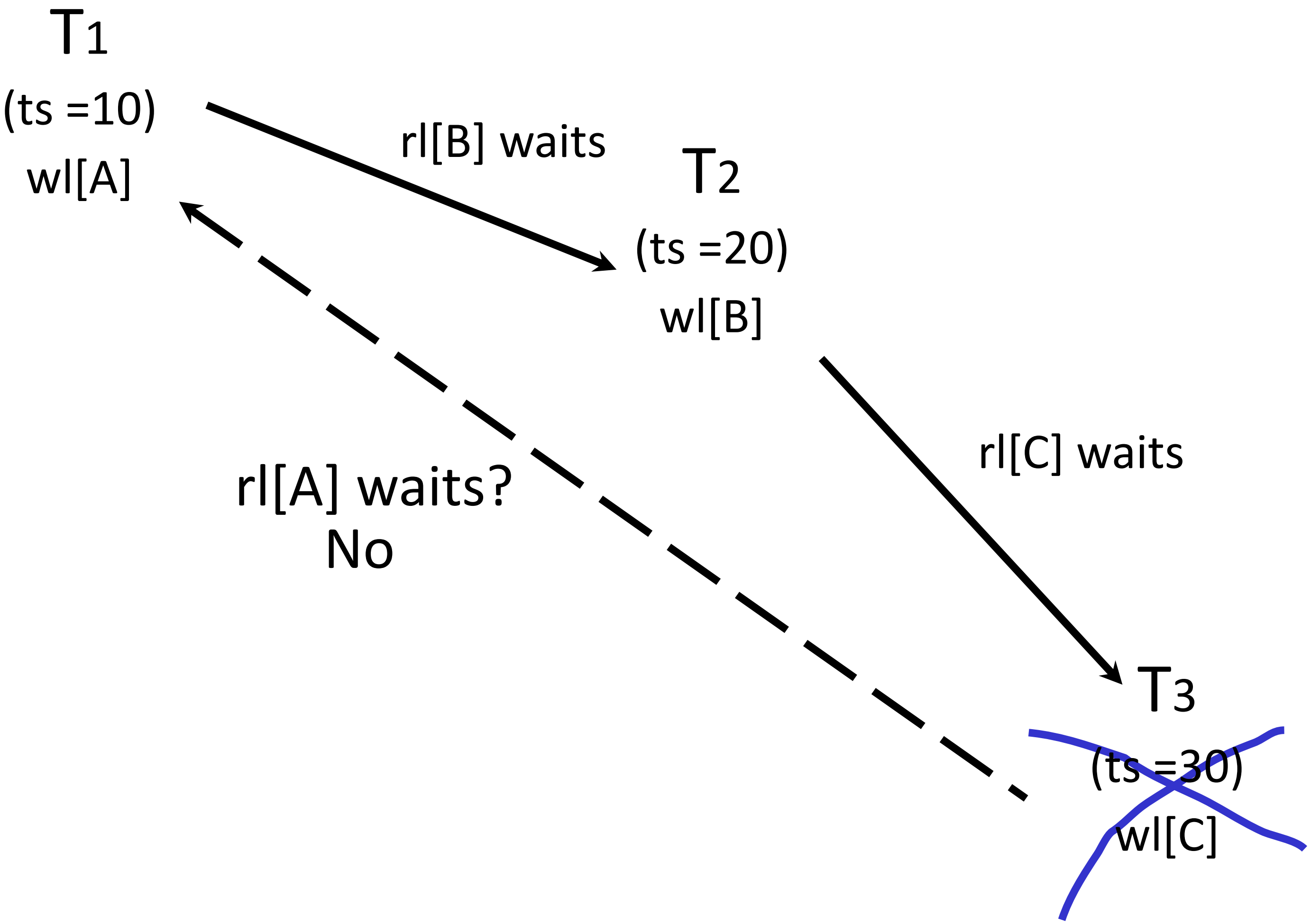
IF $ts(T_i) < ts(T_j)$ (Ti is older than Tj)

THEN T_i wait for T_j (older waits for the younger)

ELSE T_i aborts **(younger dies)**

If **Ti dies** then it later **restarts with the same timestamp**!

Wait-die: example



T_1

(ts =10)

wl[A]

rl[B] waits

T_2

(ts =20)

wl[B]

rl[C] waits

T_3

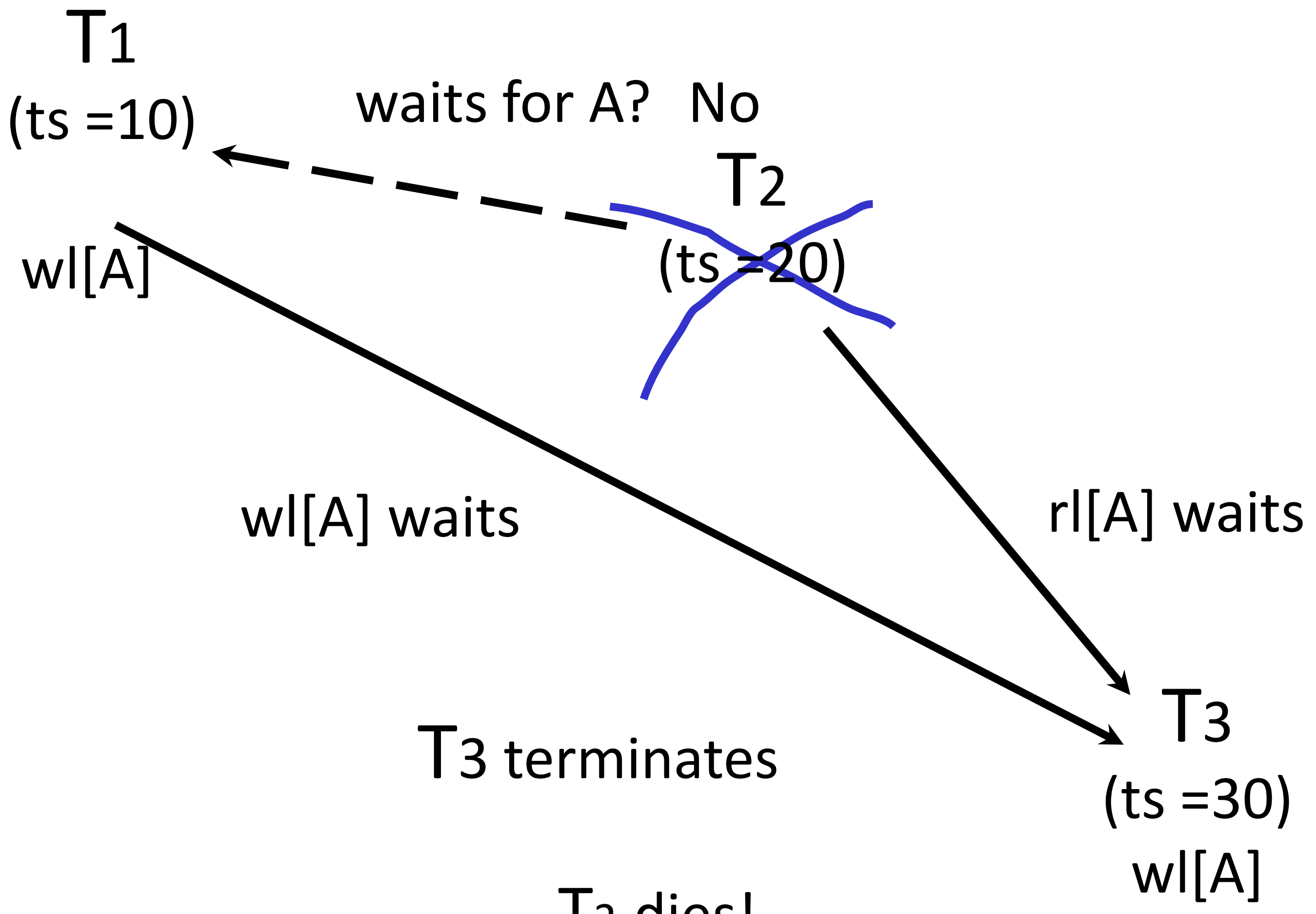
(ts =30)

wl[C]

rl[A] waits?

No

Wait-die: example



T_1
(ts =10)
wl[A]

waits for A? No

T_2
(ts =20)

wl[A] waits

rl[A] waits

T_3
(ts =30)
wl[A]

T_3 terminates

T_2 dies!

T_2 would not die with the wait-for graph

Wound-wait

A T may only wait only for an **older one**.

Suppose Ti requests a data item currently held by Tj

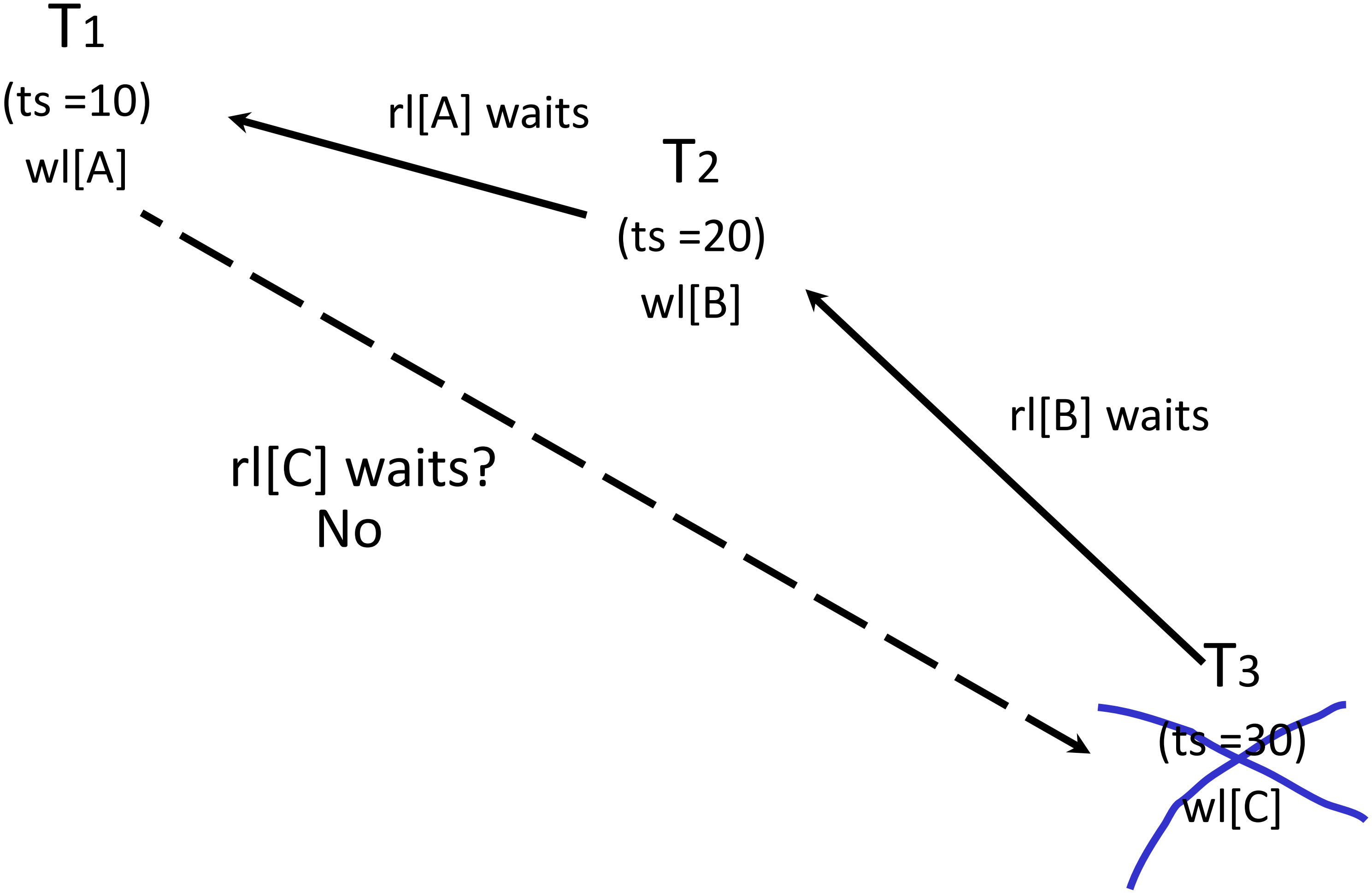
IF ts(T_i) < ts(T_j) (Ti is older than Tj)

THEN T_i wounds T_j and takes the lock (younger dies: lock to older)

ELSE T_i waits (younger waits for older)

If **Tj dies** then it later **restarts with the same timestamp**

Wound-wait: example



T_1

(ts =10)

wl[A]

rl[A] waits

T_2

(ts =20)

wl[B]

rl[B] waits

rl[C] waits?
No

T_3

(ts =30)

wl[C]

Comparing Deadlock Management Schemes

Wait-die and **Wound-wait** ensure **no starvation**

Wait-die (older waits) tends to roll back more transactions then **Wound-wait** (younger waits) but they tend to have done less work

Wait-die and **Wound-wait** are easier to implement than **waits-for graph**

Waits-for graph technique only aborts transactions if there really is a deadlock (unlike the others)

Snapshot isolation

- Optimistic concurrency control
- T always reads data as they were when it started
- T reads/writes without locks in its own snapshot, which is not visible to others.
- First Committer Wins Rule:
  - A T commits only if no other concurrent transaction has already written data that T intends to write (no writeset conflict).

Snapshot isolation: example

	T1	T2	T3
		begin	
Snapshot(T2)		w[y:=1]	
x = y = z = 0		c	
	begin		
	r[x=0]		
Snapshot(T1)	r[y=1]		
x = z = 0			begin
y = 1			w[x:=2]
Snapshot(T3)			w[z:=3]
x = z = 0			c
y = 1	r[z=0]		
	c		

T1	T2	T3
	begin	
	w[y:=1]	
	c	
begin		
r[x=0]		
r[y=1]		
		begin
		w[x:=2]
		w[z:=3]
		c <-- ?
r[z=0]		
w[x:=3]		
c <-- ?		
abort		

All T commit ? Yes

Is strict 2PL ? No

Snapshot isolation: properties

Reading is never blocked and also does not block other T

Avoids the usual anomalies: dirty read, lost update, ...

PROBLEM: **it can produce non-serializable histories**

Snapshot serialization anomalies

Consider two **Ts** that starts (at the same time) with a state **x=3** e **y=17**:

T1 (x:=y)	T2 (y:= x)
begin	begin
r[y= ?]	r[x= ?]
w[x :=y]	w[y :=x]
c	c

Serializable Isolation:

T1, T2: x =17 , y =17

T2, T1: x= 3, y= 3

Snapshot Isolation:

x= 17, y= 3

EXERCISE

Exercise 10.3 Consider the following transactions and the history H:

$T_1 = r_1[a]; w_1[a]; c_1$

$T_2 = r_2[b]; w_2[a]; c_2$

$H = r_1[a]; r_2[b]; w_2[a]; c_2; w_1[a]; c_1$

Answer the following questions:

1. Is H c-serializable?

2. Is H a history produced by a strict 2PL protocol?

3. Suppose that a strict 2PL serializer receives the following requests (where rl and wl means read lock and write lock):

$rl_1[a]; r_1[a]; rl_2[b]; r_2[b]; wl_2[a]; w_2[a]; c_2; wl_1[a]; w_1[a]; c$

Show the history generated by the serializer.

EXERCISE

Exercise 10.4 Consider the following history H of transactions T_1, T_2 and T_3

$H = r_3[B]; r_1[A]; r_2[C];w_1[C];w_2[B];w_2[C[;w_3[A]$

We make the following assumptions:

1. If a transaction ever gets all the locks it needs, then it instantaneously completes work, commits, and releases its locks,
2. If a transaction dies or is wounded, it instantaneously gives up its locks, and restarts only after all current transactions commit or abort,

Answer the following questions:

1. Is H c-serializable?
2. If the strict 2PL is used to handle lock requests, in what order do the transactions finally commit?
3. If the wait-die strategy is used to handle lock requests, in what order do the transactions finally commit?
4. If the wound-wait strategy is used to handle lock requests, in what order do the transactions finally commit?
5. If the snapshot strategy is used, in what order do the transactions finally commit?

EXERCISE

Exercise 10.5 Consider the transactions:
T1 = r1[x];w1[x]; r1[y];w1[y]
T2 = r2[y];w2[y]; r2[x];w2[x]
1. Compute the number of possible histories.
2. How many of the possible histories are c-equivalent to the serial history (T1; T2) and how many to the serial history (T2; T1)?

Exercise 10.6 The transaction T1 precedes T2 in the history S if all actions of T1 precede actions of T2. Give an example of a history S that has the following properties:
1. T1 precedes T2 in S,
2. S is c-serializable, and
3. in every serial history c-equivalent to S, T2 precedes T1.
The schedule may include more than 2 transactions and you do not need to consider locking actions. Please use as few transactions and read or write actions as possible.

Concurrency in real systems

Objects are of different size (granularity), and we try to reduce locks as much as possible, as well as to lock at the smallest possible level

Data is modified also for insertion and removal

When an index is updated, we must use locks!

Multiple granularity locking

- Containment hierarchy:
  DB -> Files -> Pages -> Records -> Fields
- In the containment hierarchy, we can have either low or high lock granularity:
  - **low** (towards the fields): more concurrency, more lock overhead, higher deadlock probability
  - **high** (towards the DB): less concurrency, less overhead, less deadlocks
- Every transaction should lock at its correct granularity

Multiple granularity locking

- A lock on a object – **S** or **X** – is a lock on all its components
- To lock some part of an object, an intention lock on the whole object is required
  - **IS** (intention share lock) allows one to then ask a shared lock on a part of the object
  - **IX** (intention exclusive lock) allows one to then ask an **X** lock on a part of the object
  - **SIX** (share intention exclusive lock) **S** + **IX** lock

Multigranular compatibility table

	IS	IX	S	SIX	X
IS	Y	Y	Y	Y	N
IX	Y	Y	N	N	N
S	Y	N	Y	N	N
SIX	Y	N	N	N	N
X	N	N	N	N	N

- Lock from the root towards the leafs

New kinds of locks and protocols

- Insertion and removal of records
  - To insert or remove a record from a file we must lock-X the entire file
- Concurrency on B-Tree indexes
  - The strict 2PL protocol has terrible performances: when updating an index a transaction should have an X lock on the entire tree
  - New methods have been proposed (for instance, when child node is locked, the lock on the father is released)

Summary

- Correctness criterion for isolation is c-serializability, more restrictive but easier to enforce.
- Pessimistic or optimistic approach
- Pessimistic: Strict 2PL.
  - Deadlocks arise, can either be detected or prevented
  - Multi-granularity locking
- Optimistic:
  - Snapshot
  - Timestamp