departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Concurrency Control (2)

Concurrency and Parallelism — 2017-18

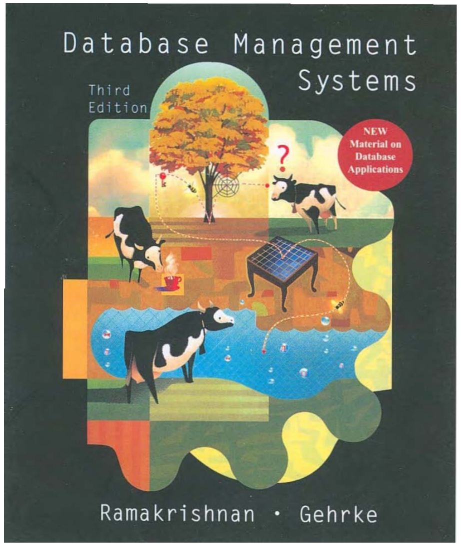
Masters in Computer Science

(Mestrado Integrado em Eng. Informática)

Joao Lourenço <joao.lourenco@fct.unl.pt>

Base on slides from: *https://users.cs.duke.edu/~shivnath/courses/fall06/Lectures/11_serial.ppt*
and: *Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke*

Concurrency Control



- Contents:
  - Conflict Serializable Schedules
  - View Serializable Schedules
  - Two Phase Locking
    - Deadlock prevention and detection
  - Other Concurrency Control methods: Optimistic, Timestamp and Multiversioning
- Reading list:
  - Chap 17 of Database management systems (3rd Ed.)
    McGraw-Hill Education
    Raghu Ramakrishnan, Johannes Gehrke
    ISBN: 0-07-123151-X

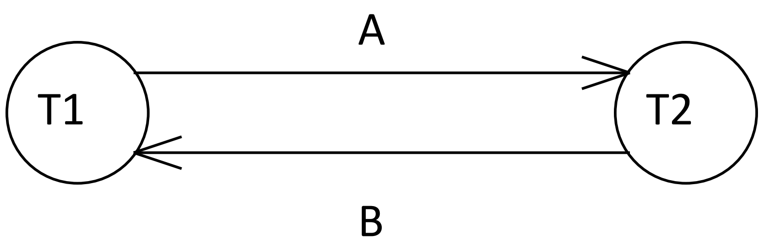
Conflict Serializable Schedules

- Two schedules are **conflict equivalent** if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way
- Schedule S is **conflict serializable** if S is conflict equivalent to some serial schedule

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)	



Dependency graph

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

Dependency Graph

- ***Dependency graph***:
  - One node per Trx;
  - Edge from *Ti* to *Tj* if *Tj* reads/writes an object last written by *Ti*.

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

Two-Phase Locking (2PL)

- Two-Phase Locking Protocol
  - Each Trx must obtain a 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.
  - If an Trx holds an X lock on an object, no other Trx can get a lock (S or X) on that object.

Strict 2PL

- ***Strict Two-phase Locking (Strict 2PL) Protocol***:
  - Each Trx must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes
  - If an Trx holds an X lock on an object, no other Trx can get a lock (S or X) on that object.
- Strict 2PL allows only schedules whose precedence graph is acyclic

View Serializability

- Schedules S1 and S2 are view equivalent if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2

View Serializability

- Schedules S1 and S2 are view equivalent if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2

T1:	R(A)		W(A)	
T2:		W(A)		
T3:				W(A)

T1:	R(A), W(A)		
T2:		W(A)	
T3:			W(A)

View Serializability

- Schedules S1 and S2 are view equivalent if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2

```
T1: R(A)         W(A)
T2:         W(A)
T3:                        W(A)
```

```
T1: R(A), W(A)
T2:                W(A)
T3:                        W(A)
```

View Serializability

- Schedules S1 and S2 are *view equivalent* if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2

T1:	R(A)		W(A)	
T2:		W(A)		
T3:				W(A)

T1:	R(A), W(A)		
T2:		W(A)	
T3:			W(A)

View Serializability

- Schedules S1 and S2 are view equivalent if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2

T1:	R(A)		W(A)	
T2:		W(A)		
T3:				W(A)

T1:	R(A), W(A)		
T2:		W(A)	
T3:			W(A)

View Serializability

- Schedules S1 and S2 are view equivalent if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2

T1:	R(A)		W(A)	
T2:		W(A)		
T3:				W(A)

T1:	R(A), W(A)		
T2:		W(A)	
T3:			W(A)

T1:		R(A)		W(B)		
T2:	R(B), W(A)				W(B)	
T3:			R(A)			W(B)

T1:		R(A), W(B)	
T2:	R(B), W(A), W(B)		
T3:			R(A), W(B)

View Serializability

- Schedules S1 and S2 are view equivalent if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2

```
T1: R(A)        W(A)
T2:       W(A)
T3:                       W(A)
```


```
T1: R(A), W(A)
T2:                 W(A)
T3:                        W(A)
```

```
T1:             R(A)       W(B)
T2: R(B) W(A)                   W(B)
T3:                  R(A)            W(B)
```

```
T1:                  R(A), W(B)
T2: R(B) W(A), W(B)
T3:                               R(A), W(B)
```

View Serializability

- Schedules S1 and S2 are view equivalent if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2

YES

T1:	R(A)		W(A)	
T2:		W(A)		
T3:				W(A)

T1:	R(A), W(A)		
T2:		W(A)	
T3:			W(A)

T1:		R(A)		W(B)		
T2:	R(B), W(A)				W(B)	
T3:			R(A)			W(B)

T1:		R(A), W(B)	
T2:	R(B), W(A), W(B)		
T3:			R(A), W(B)

View Serializability

- Schedules S1 and S2 are view equivalent if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2

T1:	R(A)		W(A)	
T2:		W(A)		
T3:				W(A)

YES

T1:	R(A), W(A)		
T2:		W(A)	
T3:			W(A)

T1:		R(A)		W(B)		
T2:	R(B), W(A)				W(B)	
T3:			R(A)			W(B)

YES

T1:		R(A), W(B)	
T2:	R(B), W(A), W(B)		
T3:			R(A), W(B)

Lock Management

- Lock and unlock requests are handled by the lock manager
- Lock table entry:
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations
- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock

Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection

Deadlock Prevention

- Assign a timestamp to each Trx begin
- Assign priorities based on timestamps
- Assume Ti wants a lock that Tj holds. Two policies are possible:
  - Wait-Die: It Ti has higher priority, Ti waits for Tj; otherwise Ti aborts
  - Wound-wait: If Ti has higher priority, Tj aborts; otherwise Ti waits
- Trxs restart with their original timestamp

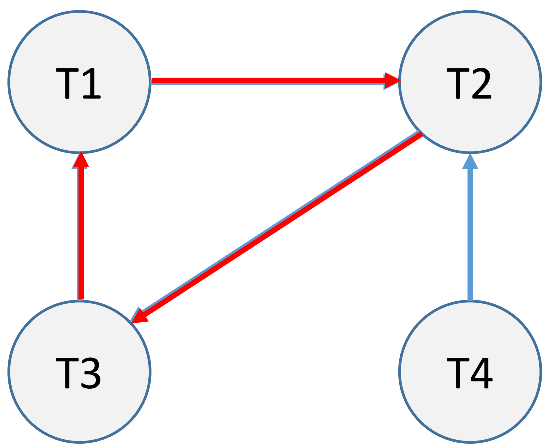
Deadlock Detection

- Create a waits-for graph:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Periodically check for cycles in the waits-for graph

Deadlock Detection (cont.)

Example:

T1:	S(A), R(A),		S(B)				
T2:		X(B), W(B)			X(c)		
T3:				S(C), R(C)			**X(A)**
T4:						X(B)	



Phantom reads

- If we relax the assumption that the DB is a fixed collection of objects, even Strict 2PL will not assure serializability:
  - T1 locks all pages containing sailor records with *rating* = 1, and finds oldest sailor (say, *age* = 71).
  - Next, T2 inserts a new sailor; *rating* = 1, *age* = 96.
  - T2 also deletes oldest sailor with rating = 2 (and, say, *age* = 80), and commits.
  - T1 now locks all pages containing sailor records with *rating* = 2, and finds oldest (say, *age* = 63).
- No consistent DB state where T1 is "correct"!

The Problem

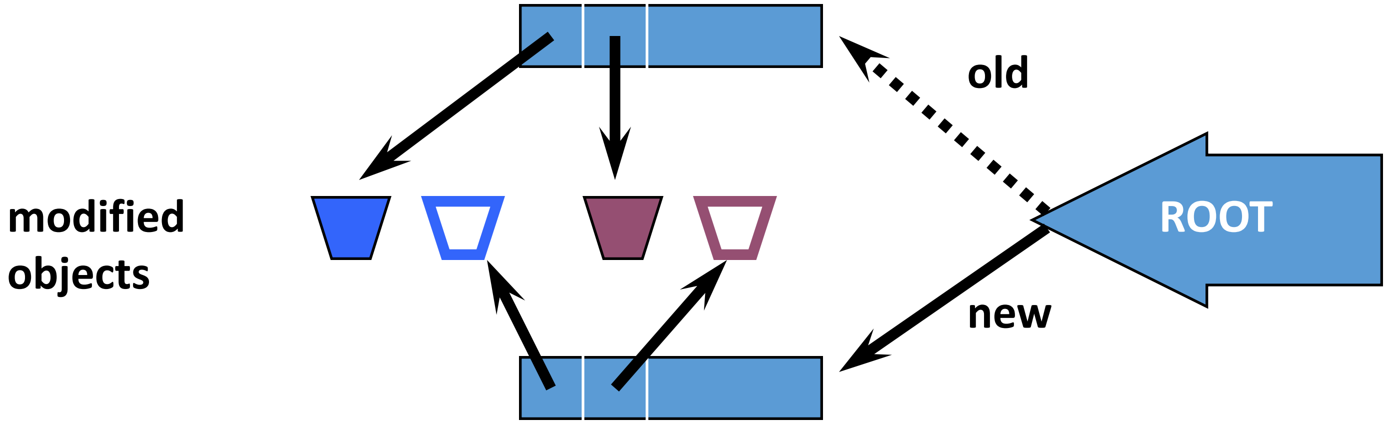
- T1 implicitly assumes that it has locked the set of all sailor records with *rating* = 1.
  - Assumption only holds if no sailor records are added while T1 is executing!
  - Need some mechanism to enforce this assumption. (Index locking or predicate locking.)
- Example shows that conflict serializability guarantees serializability only if the set of objects is fixed!

Optimistic CC (Kung-Robinson)

- Locking is a conservative approach in which conflicts are prevented. Disadvantages:
  - Lock management overhead.
  - Deadlock detection/resolution.
  - Lock contention for heavily used objects.
- If conflicts are rare, we might be able to gain concurrency by not locking, and instead checking for conflicts before Trxs commit.

Kung-Robinson Model

- Trxs have three phases:
  - READ: Trxs read from the database, but make changes to private copies of objects.
  - VALIDATE: Check for conflicts.
  - WRITE: Make local copies of changes public.

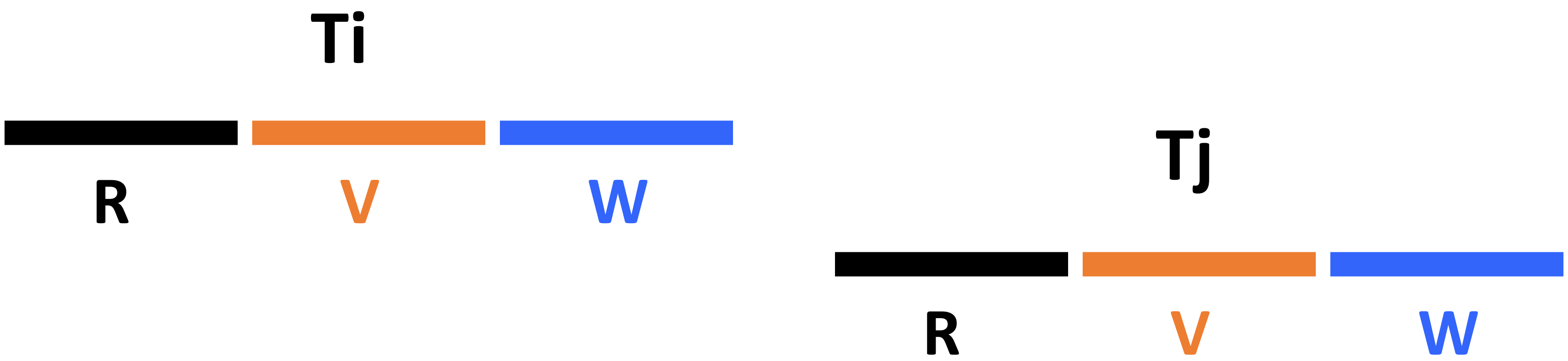


Validation

- Test conditions that are sufficient to ensure that no conflict occurred.
- Each Trx is assigned a numeric id.
  - Just use a **timestamp**.
- Trx ids assigned at end of READ phase, just before validation begins. (Why then?)
- ReadSet(Ti): Set of objects read by Trx Ti.
- WriteSet(Ti): Set of objects modified by Ti.

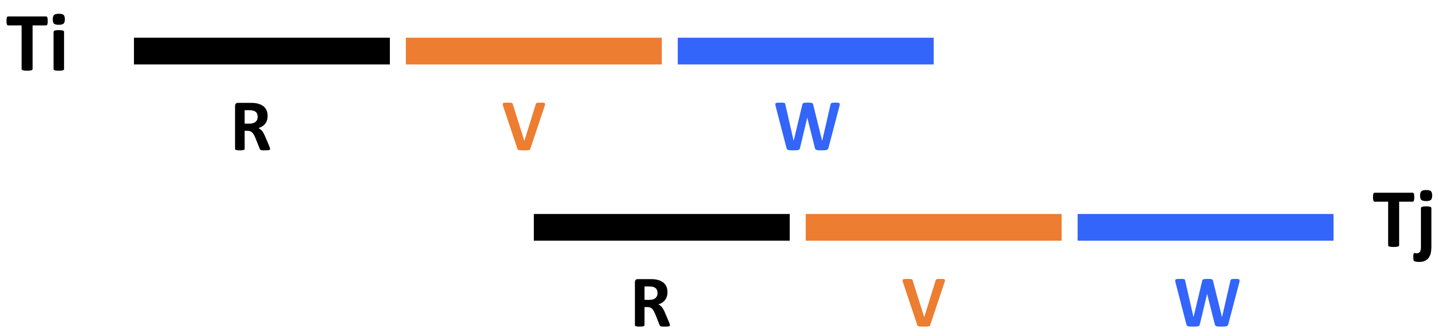
Test 1

- For all i and j such that Ti < Tj, check that Ti completes before Tj begins.



Test 2

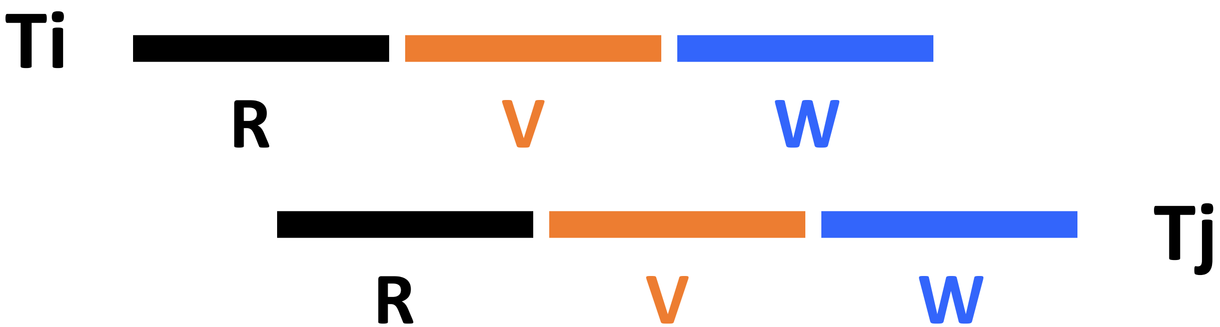
- For all i and j such that Ti < Tj, check that both:
  - Ti completes before Tj begins its Write phase **+**
  - WriteSet(Ti) ∩ ReadSet(Tj) is empty.



Does Tj read dirty data?

Test 3

- For all i and j such that Ti < Tj, check that all:
  - Ti completes Read phase before Tj does **+**
  - WriteSet(Ti) ∩ ReadSet(Tj) is empty **+**
  - WriteSet(Ti) ∩ WriteSet(Tj) is empty.



Does Tj read dirty data? Does Ti overwrite Tj's writes?

Applying Tests 1 & 2: Serial Validation

- To validate Trx T:

```
valid = true;
// S = set of Trxs that committed after Begin(T)
< foreach  Ts in S do {
   if ReadSet(T) ∩ WriteSet(Ts) ≠ ∅
         then valid = false;
   }
   if valid then { install updates; // Write phase
                   Commit T } >
           else Restart T
```

end of critical section

Comments on Serial Validation

- Applies Test 2, with T playing the role of Tj and each Trx in Ts (in turn) being Ti.
- Assignment of Trx id, validation, and the Write phase are inside a **critical section**!
  - I.e., Nothing else goes on concurrently.
  - If Write phase is long, major drawback.
- Optimization for Read-only Trxs:
  - Don't need critical section (because there is no Write phase).

Overheads in Optimistic CC

- Must record read/write activity in ReadSet and WriteSet per Trx.
  - Must create and destroy these sets as needed.
- Must check for conflicts during validation, and must make validated writes ``global''.
  - Critical section can reduce concurrency.
  - Scheme for making writes global can reduce clustering of objects.
- Optimistic CC restarts Trxs that fail validation.
  - Work done so far is wasted; requires clean-up.

"Optimistic" 2PL

- If desired, we can do the following:
  - Set S locks as usual.
  - Make changes to private copies of objects.
  - Obtain all X locks at end of Trx, make writes global, then release all locks.
- In contrast to Optimistic CC as in Kung-Robinson, this scheme results in Trxs being blocked, waiting for locks.
  - However, no validation phase, no restarts (modulo deadlocks).

Timestamp CC

- **Idea:** Give each object a read-timestamp (RTS) and a write-timestamp (WTS), give each Trx a timestamp (TS) when it begins:
  - If action ai of Trx Ti conflicts with action aj of Trx Tj, and TS(Ti) < TS(Tj), then ai must occur before aj. Otherwise, restart violating Trx.

When Trx T wants to read Object O

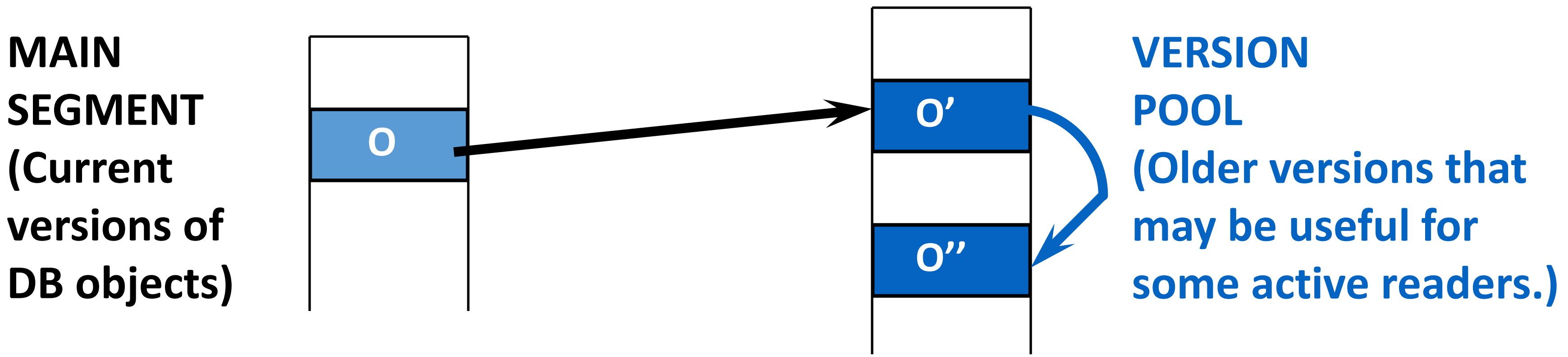
- If TS(T) < WTS(O), this violates timestamp order of T w.r.t. writer of O.
  - So, abort T and restart it with a new, larger TS. (If restarted with same TS, T will fail again! Contrast use of timestamps in 2PL for ddlk prevention.)
- If TS(T) > WTS(O):
  - Allow T to read O.
  - Reset RTS(O) to max(RTS(O), TS(T))
- Change to RTS(O) on reads must be written to disk! This and restarts represent overheads.

When Trx T wants to Write Object O

- If TS(T) < RTS(O), this violates timestamp order of T w.r.t. a reader of O; abort and restart T.
- If TS(T) < WTS(O), violates timestamp order of T w.r.t. another writer of O.
- If TS(T) >= RTS(O) and TS(T) >= WTS(O)
  - Allow T to write O.
  - Reset RTS(O) to TS(T)
  - Reset WTS(O) to TS(T)

Multiversion Timestamp CC

- **Idea:** Let writers make a "new" copy while readers use an appropriate "old" copy:



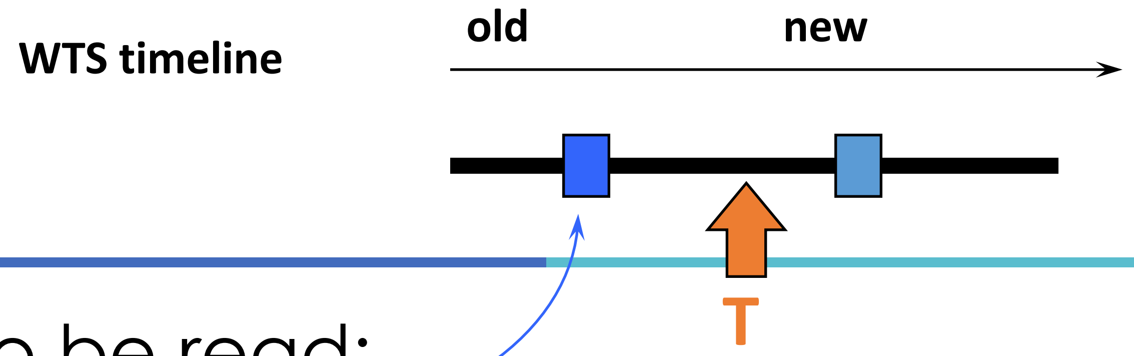
❖ Readers are always allowed to proceed.

– But may be blocked until writer commits.

Multiversion CC (Contd.)

- Each version of an object has its writer's TS as its WTS, and the TS of the Trx that most recently read this version as its RTS.
- Versions are chained backward; we can discard versions that are "too old to be of interest".
- Each Trx is classified as Reader or Writer.
  - Writer *may* write some object; Reader never will.
  - Trx declares whether it is a Reader when it begins.

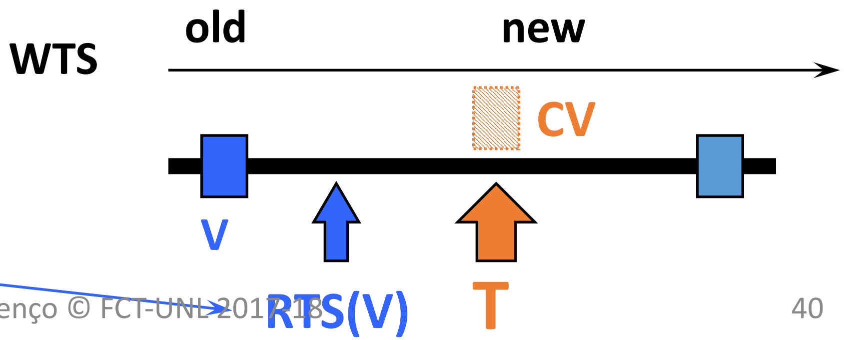
Reader Trx



- For each object to be read:
  - Finds **newest version** with WTS < TS(T). (Starts with current version in the main segment and chains backward through earlier versions.)
- Assuming that some version of every object exists from the beginning of time, Reader Trxs are never restarted.
  - However, might block until writer of the appropriate version commits.

Writer Trx

- To read an object, follows reader protocol.
- To write an object:
  - Finds **newest version V** s.t. WTS < TS(T).
  - If RTS(V) < TS(T), T makes a copy CV of V, with a pointer to V, with WTS(CV) = TS(T), RTS(CV) = TS(T). (Write is buffered until T commits; other Trxs can see TS values but can't read version CV.)
  - Else, reject write.



Summary

- There are several lock-based concurrency control schemes (Strict 2PL, 2PL). Conflicts between transactions can be detected in the dependency graph
- The lock manager keeps track of the locks issued. Deadlocks can either be prevented or detected.
- Naïve locking strategies may have the phantom problem

Summary (Cont.)

- Optimistic CC aims to minimize CC overheads in an ``optimistic'' environment where reads are common and writes are rare.
- Optimistic CC has its own overheads however; most real systems use locking.

Summary (Contd.)

- Timestamp CC is another alternative to 2PL; allows some serializable schedules that 2PL does not (although converse is also true).
- Ensuring recoverability with Timestamp CC requires ability to block Trxs, which is similar to locking.
- Multiversion Timestamp CC is a variant which ensures that read-only Trxs are never restarted; they can always read a suitable older version. Additional overhead of version maintenance.

The END