Drawbacks of locking

- Lock maintenance costs an overhead.
- The use of locks can result in deadlock and deadlock prevention reduces concurrency severely.
- To avoid cascading aborts, locks cannot be released until the end of the transaction, which may reduce significantly the potential of concurrency.

Exercise about locking (1)

Explain why serial equivalence requires that once a transaction has released a lock on an object, it is not allowed to obtain any more locks.

A server manages the objects a_1, a_2, ... a_n. The server provides two operations for its clients:

read (i) returns the value of a_i

write(i, Value) assigns *Value* to a_i

The transactions *T* and *U* are defined as follows:

T: x= read (i); write(j, 44);

U: write(i, 55);write(j, 66);

Describe an interleaving of the transactions *T* and *U* in which locks are released early with the effect that the interleaving is not serially equivalent (**hint:** the ordering of different pairs of conflicting operations of two transactions must be the same).

Exercise about locking (2)

Because the ordering of different pairs of conflicting operations of two transactions must be the same.

For an example where locks are released early:

T	*T's locks*	*U*	*U's locks*
	lock i		
x:= read (i);			
	unlock i		
			lock i
		write(i, 55);	
			lock j
		write(j, 66);	
		commit	unlock i, j
	lock j		
write(j, 44);			
	unlock j		
commit			

T conflicts with *U* in access to a_i. Order of access is *T* then *U*.

T conflicts with *U* in access to a_j. Order of access is *U* then *T*. These interleavings are not serially equivalent.

Exercise about locking (3)

Initial values of a_i and a_j are 10 and 20. Which of the following interleavings are serially equivalent and which could occur with two-phase locking?

(a)

T	U
x= read (i);	
	write(i, 55);
write(j, 44);	
	write(j, 66);

(b)

T	U
x= read (i);	
write(j, 44);	
	write(i, 55);
	write(j, 66);

(c)

T	U
	write(i, 55);
	write(j, 66);
x= read (i);	
write(j, 44);	

(d)

T	U
	write(i, 55);
x= read (i);	
	write(j, 66);
write(j, 44);	

Exercise about locking (4)

a) serially equivalent but not with two-phase locking.

b) serially equivalent and with two-phase locking.

c) serially equivalent and with two-phase locking.

d) serially equivalent but not with two-phase locking.

Optimistic concurrency control

With locks we had deadlock T→ U at i and U→ T at j. What would happen with the optimistic scheme?

Working phase

Validation phase

Update phase

- If validated, the changes in its tentative versions are made permanent.
- read-only transactions can commit immediately after passing validation.

- a transaction proceeds without restriction until the *closeTransaction* (no waiting, therefore no deadlock)
- it is t
  conflict with other transactions

With optimistic scheme, whichever validates first will pass and commit, the other will abort.

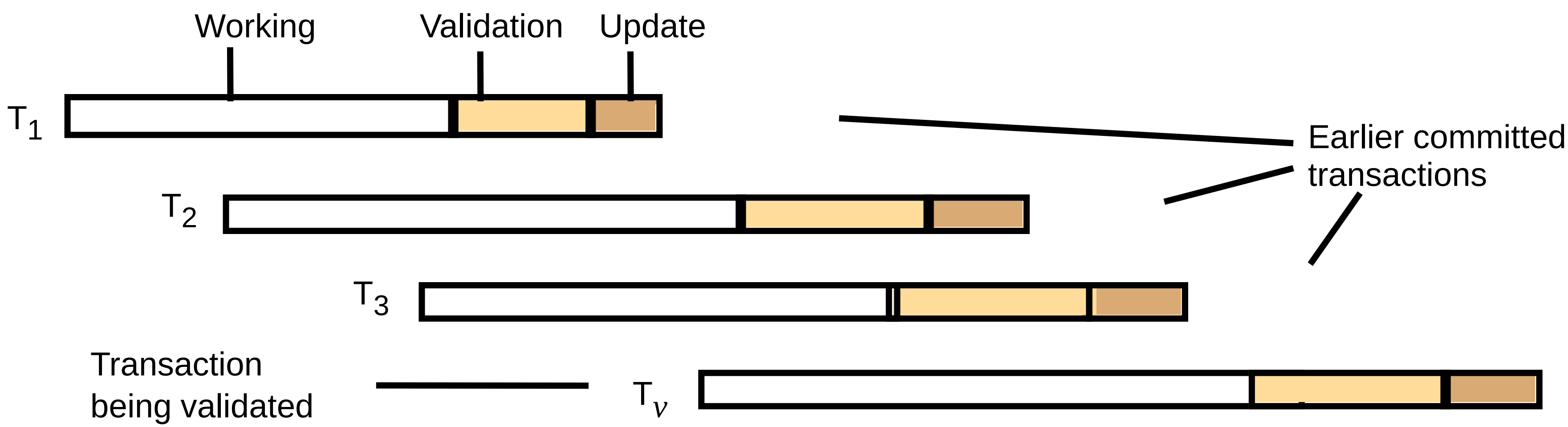
- when a conflict arises, a transaction is aborted
- each transaction has three phases:

• *Validation can be* simplified by *omitting rule 3* (if no overlapping of validate and update phases)

- We use the read-write conflict rules
  - to ensure a particular transaction is serially equivalent with respect to all other overlapping transactions
- each transaction is given a transaction number when it starts validation (the number is kept if it commits)
- the rules ensure serializability of transaction T_v (transaction being validated) with respect to transaction T_i

T_v	T_i	*Rule*		
write	*read*	1.	T_i must not read objects written by T_v	forward
read	*write*	2.	T_v must not read objects written by T_i	backward
write	*write*	3.	T_i must not write objects written by T_v and T_v must not write objects written by T_i	

The earlier committed transactions are *T*1, *T*2 and *T*3. *T1* committed before T_v started. (*earlier* means they started validation earlier)



Rule 1 (T_v's *write* vs T_i's *read*) is satisfied because reads of earlier transactions were done before T_v entered validation (and possible updates)

Rule 2 - check if T_v's read set overlaps with write sets of earlier T_i
*T*2 and *T*3 committed before T_v finished its working phase.

- Backward validation

Rule3 - (*write* vs *write*) assume no overlap of update.

Backward Validation of Transactions

Backward validation of transaction T_v

```
boolean valid = true;
for (int Ti = startTn+1; Ti <= finishTn; Ti++){
        if (read set of Tv intersects write set of Ti) valid = false;
}
```

(Page 709)

to carry out this algorithm, we must keep write sets of recently committed transactions

- *startTn* is the biggest transaction number assigned to some other committed transaction when T_v started its working phase
- *finishTn* is biggest transaction number assigned to some other committed transaction when *Tv* started its validation phase
- In figure, *StartTn* + 1 = T_2 and *finishTn* = T_3. In backward validation, the read set of T_v must be compared with the write sets of T_2 and T_3.
- the only way to resolve a conflict is to abort T_v

Forward validation

Go back to conflict rules and Fig. 16.28

the scheme must allow for the fact that read sets of active transactions may change during validation

- Rule 1. the write set of overlapping active tran
  - In Figure 16.28, the write set of T_v must be compared with the read sets of *active1* and *active2*.
- Rule 2. (read T_v vs write T_i) is automatically fulfilled because the active transactions do not write until after T_v has completed.

as the other transactions are still active, we have a choice of aborting them or T_v
if we abort T_v, it may be unnecessary as an active one may anyway abort

Forward validation of transaction Tv

read only transactions always pass validation

```
boolean valid = true;
for (int Tid = active1; Tid <= activeN; Tid++){
        if (write set of Tv intersects read set of Tid)  valid = false;
}
```

Distributed deadlock detection is very hard to implement!

- In conflict, choice of transaction to abort
  - forward validation allows flexibility, whereas backward validation allows only one choice (the one being validated)
- In general read sets > than write sets.
  - backward validation
    - compares a possibly large read set against the old write sets
    - overhead of storing old write sets
  - forward validation
    - checks a small write set against the read sets of active transactions
    - need to allow for new transactions starting during validation
- Starvation
  - after a transaction is aborted, the client must restart it, but there is no guarantee it will ever succeed

Starvation vs deadlock?

Which is more likely? - starvation or deadlock

16.6 Timestamp ordering concurrency control

- each operation in a transaction is validated when it is carried out
  - if an operation cannot be validated, the transaction is aborted
  - each transaction is given a unique timestamp when it starts.
    - The timestamp defines its position in the time sequence of transactions.
  - requests from transactions can be totally ordered by their timestamps.
- basic timestamp ordering rule (based on operation conflicts)
  - A request to write an object is valid only if that object was last read and written by earlier transactions.
  - A request to read an object is valid only if that object was last written by an earlier transaction
- this rule assumes only one version of each object
- refine the rule to make use of the tentative versions
  - to allow concurrent access by transactions to objects

Operation conflicts for timestamp ordering

When a *write* operation is accepted it is put in a tentative version and given a write timestamp

if necessary) but there is no need for the client to wait

When a *read* operation is accepted it is directed to the tentative version with the maximum write timestamp less than the transaction timestamp

- only wait for earlier ones (no deadlock)

– each read or write operation is checked with the conflict rules

T_c is the current transaction, T_i are other transactions

$T_i > T_c$ means T_i is later than T_c

as usual write operations are in tentative objects

each object has a write timestamp and a set of tentative versions each with its own write timestamp and a set of read timestamps

Operation conflicts for timestamp ordering

Rule	T_c	T_i	
1.	*write*	*read*	T_c must not *write* an object that has been *read* by any T_i where $T_i > T_c$ this requires that $T_c \geq$ the maximum read timestamp of the object.
2.	*write*	*write*	T_c must not *write* an object that has been *written* by any T_i where $T_i > T_c$ this requires that $T_c >$ write timestamp of the committed object.
3.	*read*	*write*	T_c must not *read* an object that has been *written* by any T_i where $T_i > T_c$ this requires that $T_c >$ write timestamp of the committed object.

Figure 16.29

Write operations and timestamps

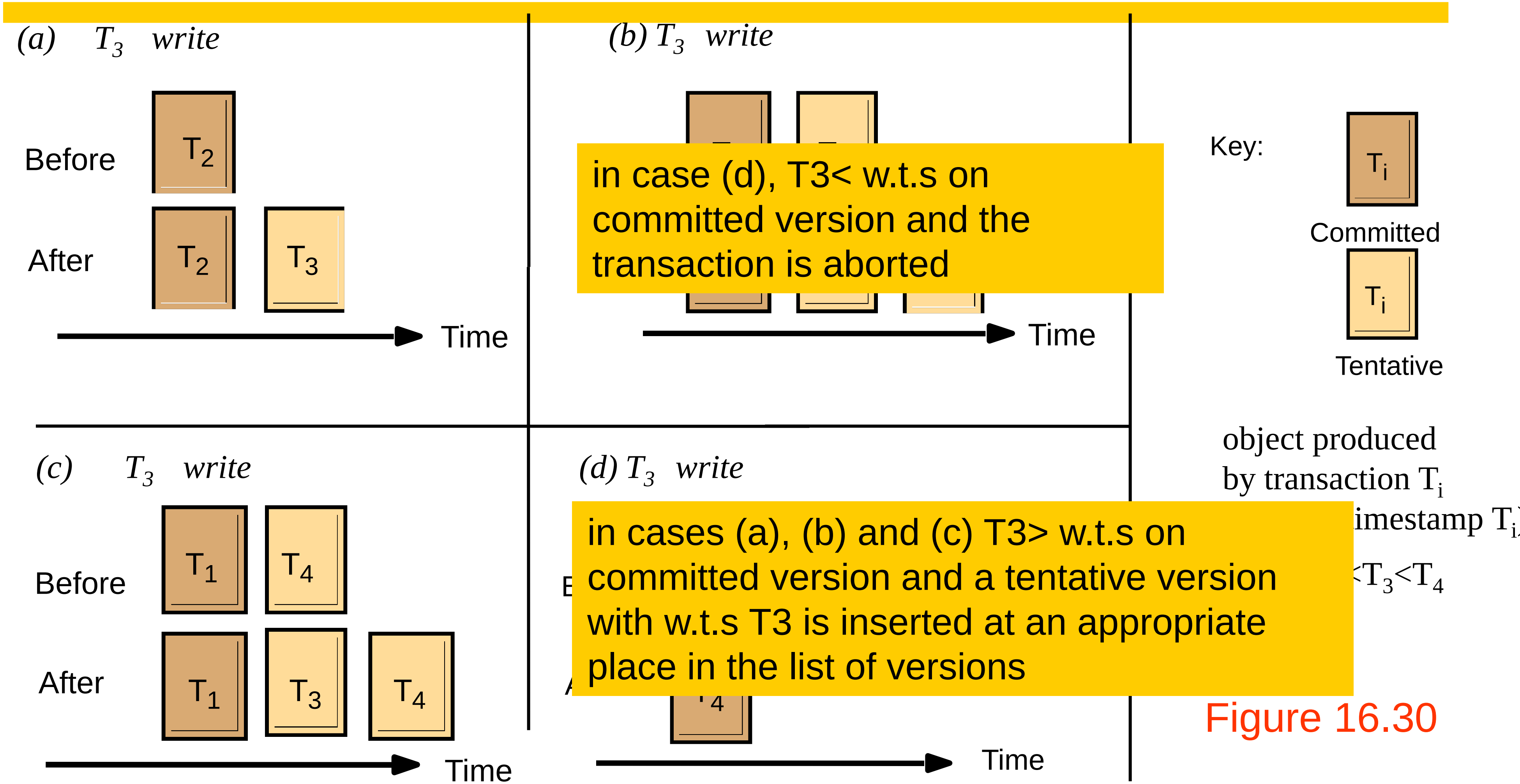


Figure 16.30

- this illustrates the versions and timestamps, when we do *T3 write*. for write to be allowed, $T_3 \geq$ maximum read timestamp (not shown)

Timestamp ordering write rule

- by combining rules 1 (write/read) and 2 (write/write)we have the following rule for deciding whether to accept a write operation requested by transaction T_c on object D
  - rule 3 does not apply to writes
  - Note: It is too late in the sense that a transaction with a later timestamp has arleady read or written the object.

if ($T_c \geq$ maximum read timestamp on D &&
    T_c > write timestamp on committed version of D)
        perform write operation on tentative version of D with write timestamp T_c
else /* write is too late */
    Abort transaction T_c

Page 714

Timestamp ordering read rule

- by using Rule 3 we get the following rule for deciding what to do about a read operation requested by transaction T_c on object *D.* That is, whether to
  - accept it immediately,
  - wait or
  - reject it

if (T_c > write timestamp on committed version of *D*) {
    let $D_{selected}$ be the version of *D* with the maximum write timestamp $\leq T_c$
    if ($D_{selected}$ is committed)
        perform *read* operation on the version $D_{selected}$
    else
        Wait until the transaction that made version $D_{selected}$ commits or aborts
        then reapply the *read* rule
} else
    Abort transaction T_c

Page 714

Read operations and tim

in case (d) there is no suitable version and T3 must abort

in case (c) the read operation is directed to a tentative version and the transaction must wait until the maker of the tentative version commits or aborts

in cases (a) and (b) the read operation is directed to a committed version, in (a) this is the only version. In (b) there is a later tentative version

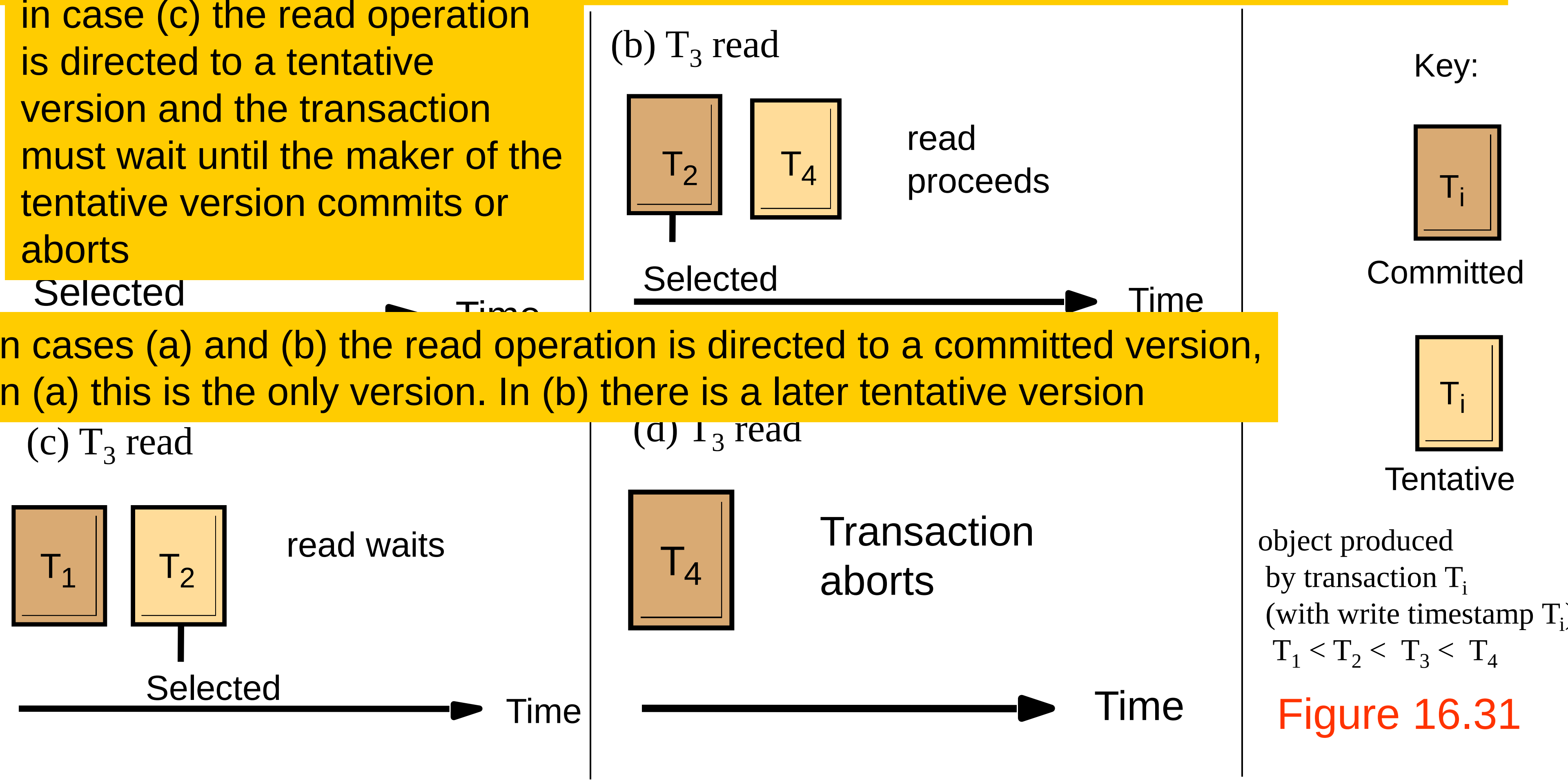


Figure 16.31

- illustrates the timestamp, ordering read rule, in each case we have *T_3 read.* In each case, a version whose write timestamp is <= *T3* is selected

Transaction commits with timestamp ordering

- when a coordinator receives a commit request, it will always be able to carry it out because all operations have been checked for consistency with earlier transactions
  - committed versions of an object must be created in timestamp order
  - the server may sometimes need to wait, but the client need not wait
  - to ensure recoverability, the server will save the 'waiting to be committed versions' in permanent storage
- the timestamp ordering algorithm is strict because
  - the read rule delays each read operation until previous transactions that had written the object had committed or aborted
  - writing the committed versions in order ensures that the write operation is delayed until previous transactions that had written the object have committed or aborted

●

Remarks on timestamp ordering concurrency control

- the method avoids deadlocks, but is likely to suffer from restarts
  - modification known as ‘ignore obsolete write’ rule is an improvement
    - If a write is too late it can be ignored instead of aborting the transaction, because if it had arrived in time its effects would have been overwritten anyway.
    - However, if another transaction has read the object, the transaction with the late write fails due to the read timestamp on the item
  - multiversion timestamp ordering (page 715)
    - allows more concurrency by keeping multiple committed versions
      - late read operations need not be aborted
    - there is not time to discuss the method now

Figure 16.32
Timestamps in transactions *T* and *U*

T	*U*	*Timestamps and versions of objects* *A*		*B*		*C*	
		RTS	*WTS*	*RTS*	*WTS*	*RTS*	*WTS*
		{}	***S***	{}	***S***	{}	***S***
openTransaction							
bal = b.getBalance()				{*T*}			
	openTransaction						
*b.setBalance(bal*1.1)*					***S***, *T*		
	bal = b.getBalance()						
	wait for T						
a.withdraw(bal/10)	● ● ●		***S***, *T*				
commit	● ● ●		***T***		***T***		
	bal = b.getBalance()			{*U*}			
	*b.setBalance(bal*1.1)*				***T***, *U*		
	c.withdraw(bal/10)						***S***, *U*

Assume that S<T<U; RTS records the maximum read timestamp; WTS records the write timestamp of each version with timestamps of committed versions in bold.

Comparison of methods for concurrency control

- pessimistic approach (detect conflicts as they arise)
  - timestamp ordering: serialisation order decided statically
  - locking: serialisation order decided dynamically
  - timestamp ordering is better for transactions where reads >> writes,
  - locking is better for transactions where writes >> reads
  - strategy for aborts
    - timestamp ordering – immediate
    - locking– waits but can get deadlock
- optimistic methods
  - all transactions proceed, but may need to abort at the end
  - efficient operations when there are few conflicts, but aborts lead to repeating work
- the above methods are not always adequate e.g.
  - in cooperative work there is a need for user notification
  - applications such as cooperative CAD need user involvement in conflict resolution

Summary

- Operation conflicts form a basis for the derivation of concurrency control protocols.
  - protocols ensure serializability and allow for recovery by using strict executions
  - e.g. to avoid cascading aborts
- Three alternative strategies are possible in scheduling an operation in a transaction:
  - (1) to execute it immediately, (2) to delay it, or (3) to abort it
  - strict two-phase locking uses (1) and (2), aborting in the case of deadlock
    - ordering according to when transactions access common objects
  - timestamp ordering uses all three - no deadlocks
    - ordering according to the time transactions start.
  - optimistic concurrency control allows transactions to proceed without any form of checking until they are completed.
    - Validation is carried out. Starvation can occur.