PAYAP UNIVERSITY

Transactions and Concurrency control

Vaidė Narváez

Computer Information Systems

September 8th, 2010

Concurrency control

PAYAP UNIVERSITY

Introduction

- Transactions must be scheduled so that their effect on shared objects is serially equivalent

- A server can achieve serial equivalence by serialising access to objects, e.g. by the use of locks
  - all access by a transaction to a particular object must be serialized with respect to another transactions access
  - all pairs of conflicting operations of two transactions should be executed in the same order

PAYAP UNIVERSITY

Exclusive locks

Transaction *T*: *balance = b.getBalance()* *b.setBalance(bal*1.1)* *a.withdraw(bal/10)*		**Transaction *U*:** *balance = b.getBalance()* *b.setBalance(bal*1.1)* *c.withdraw(bal/10)*	
Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	lock *B*		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	lock *A*	*bal = b.getBalance()*	waits for *T*'s lock on *B*
closeTransaction	unlock *A*, *B*	• • •	
			lock *B*
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock *C*
		closeTransaction	unlock *B*, *C*

Figure 12.14
same as 12.7

PAYAP UNIVERSITY

Strict two-phase locking

- strict executions prevent dirty reads and premature writes (if transactions abort).
  - a transaction that reads or writes an object must be delayed until other transactions that wrote the same object have committed or aborted.
  - to enforce this, any locks applied during the progress of a transaction are held until the transaction commits or aborts.
  - this is called *strict two-phase locking*
  - For recovery purposes, locks are held until updated objects have been written to permanent storage

PAYAP UNIVERSITY

Strict two-phase locking

- strict executions prevent dirty reads and premature writes (if transactions abort).
  - a transaction that reads or writes an object must be delayed until other transactions that wrote the same object have committed or aborted.
  - to enforce this, any locks applied during the progress of a transaction are held until the transaction commits or aborts.
  - this is called *strict two-phase locking*
  - For recovery purposes, locks are held until updated objects have been written to permanent storage

granularity!

PAYAP UNIVERSITY

Lock compatibility

- The operation conflict rules tell us that:
  - If a transaction *T* has already performed a *read* operation on a particular object, then a concurrent transaction *U* must not *write* that object until *T* commits or aborts.
  - If a transaction *T* has already performed a *write* operation on a particular object, then a concurrent transaction *U* must not *read* or *write* that object until *T* commits or aborts

For one object		*Lock requested*	
		read	*write*
Lock already set	*none*	OK	OK
	read	OK	wait
	write	wait	wait

strict 2PL

1. When an operation accesses an object within a transaction:
   (a) If the object is not already locked, it is locked and the operation proceeds.
   (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule (b) is used.)
2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.

- The sever applies locks when the read/write operations are about to be executed
- the server releases a transactions locks when it commits or aborts

PAYAP UNIVERSITY

Optimistic concurrency control

- the scheme is called *optimistic* because the likelihood of two transactions conflicting is low
- a transaction proceeds without restriction until the *commit* or *abort* (no waiting, therefore no deadlock)
- it is then checked to see whether it has come into conflict with other transactions
- when a conflict arises, a transaction is aborted

PAYAP UNIVERSITY

Optimistic concurrency control

- Each transaction has three phases:
  - *Working phase*
    - the transaction uses a tentative version of the objects it accesses (dirty reads can't occur as we read from a committed version or a copy of it)
    - the coordinator records the readset and writeset of each transaction
  - *Validation phase*
    - at commit request the coordinator validates the transaction (looks for conflicts)
    - if the validation is successful the transaction can commit.
    - if it fails, either the current transaction, or one it conflicts with is aborted
  - *Update phase*
    - If validated, the changes in its tentative versions are made permanent.
    - read-only transactions can commit immediately after passing validation

PAYAP UNIVERSITY

Validation of transactions

- 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
read	*write*	2. T_v must not read objects written by T_i
write	*write*	3. T_i must not write objects written by T_v and T_v must not write objects written by T_i

PAYAP UNIVERSITY

Validation of transactions

- *Backwards validation*: check the transaction with other preceding overlapping transactions- those that entered the validation phase before it. (*rule 2*)
- *Forward validation*: check the transaction with other later transactions, which are still active.(*rule 1*)

Validation of transactions

- *Backwards validation*: check the transaction with other preceding overlapping transactions- those that entered the validation phase before it. (*rule 2*)
- *Forward validation*: check the transaction with other later transactions, which are still active.(*rule 1*)

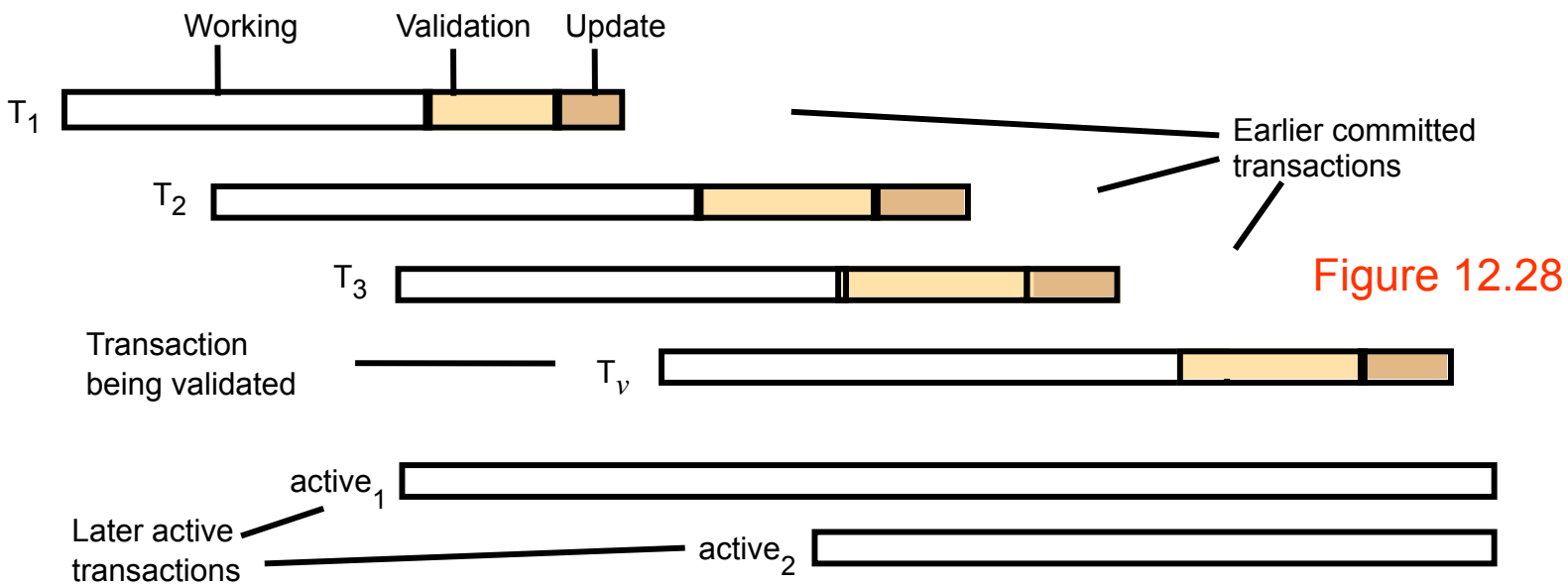


Figure 12.28

PAYAP UNIVERSITY

Timestamp ordering

- 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

Conflicts for timestamp ordering

Refined rule:

- tentative versions are committed in the order of their timestamps (wait if necessary) but there is no need for the client to wait
- but read operations wait for earlier transactions to finish (only wait for earlier ones (no deadlock))
- each read or write operation is checked with the conflict rules

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 12.29

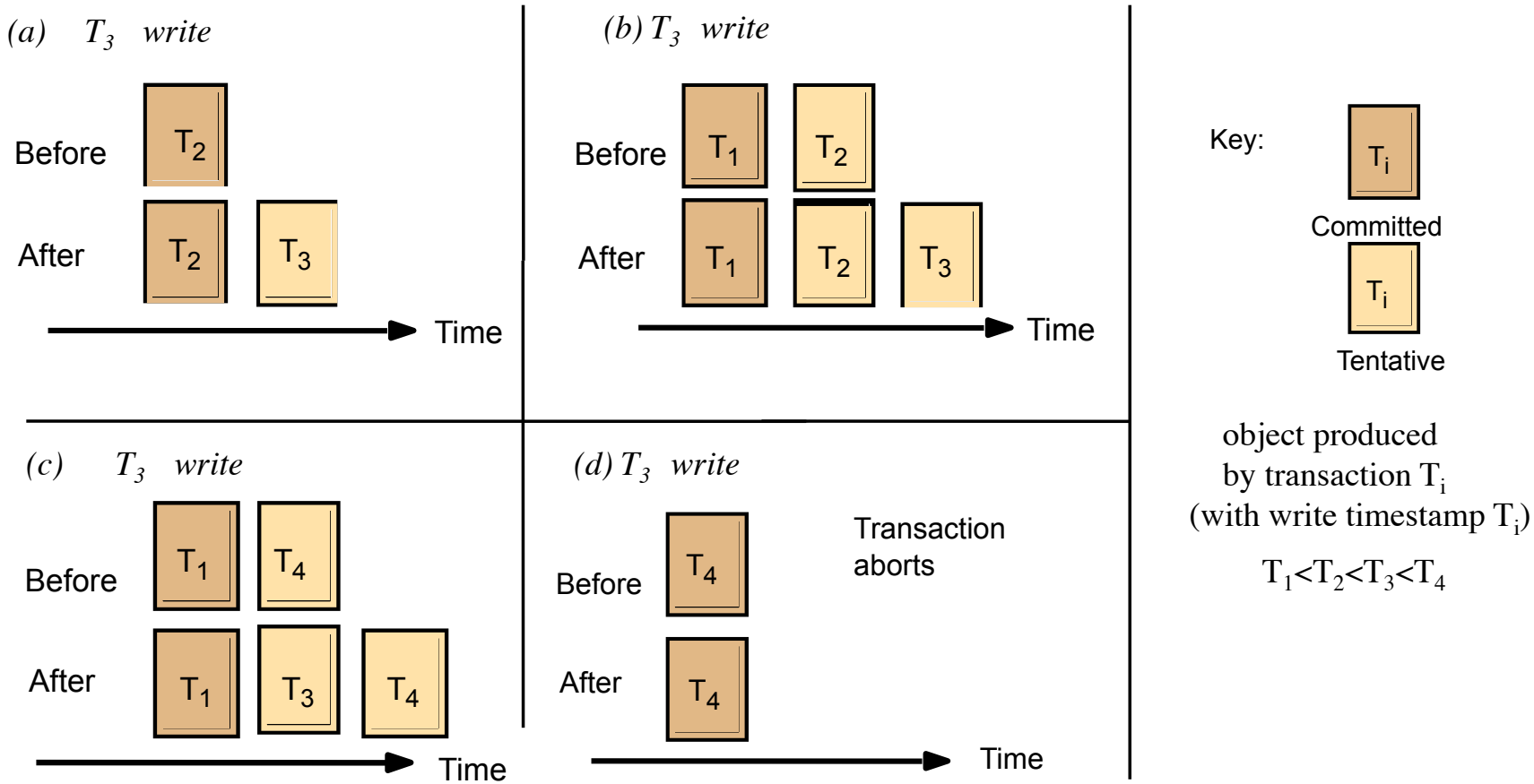
PAYAP UNIVERSITY

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

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

Write rule



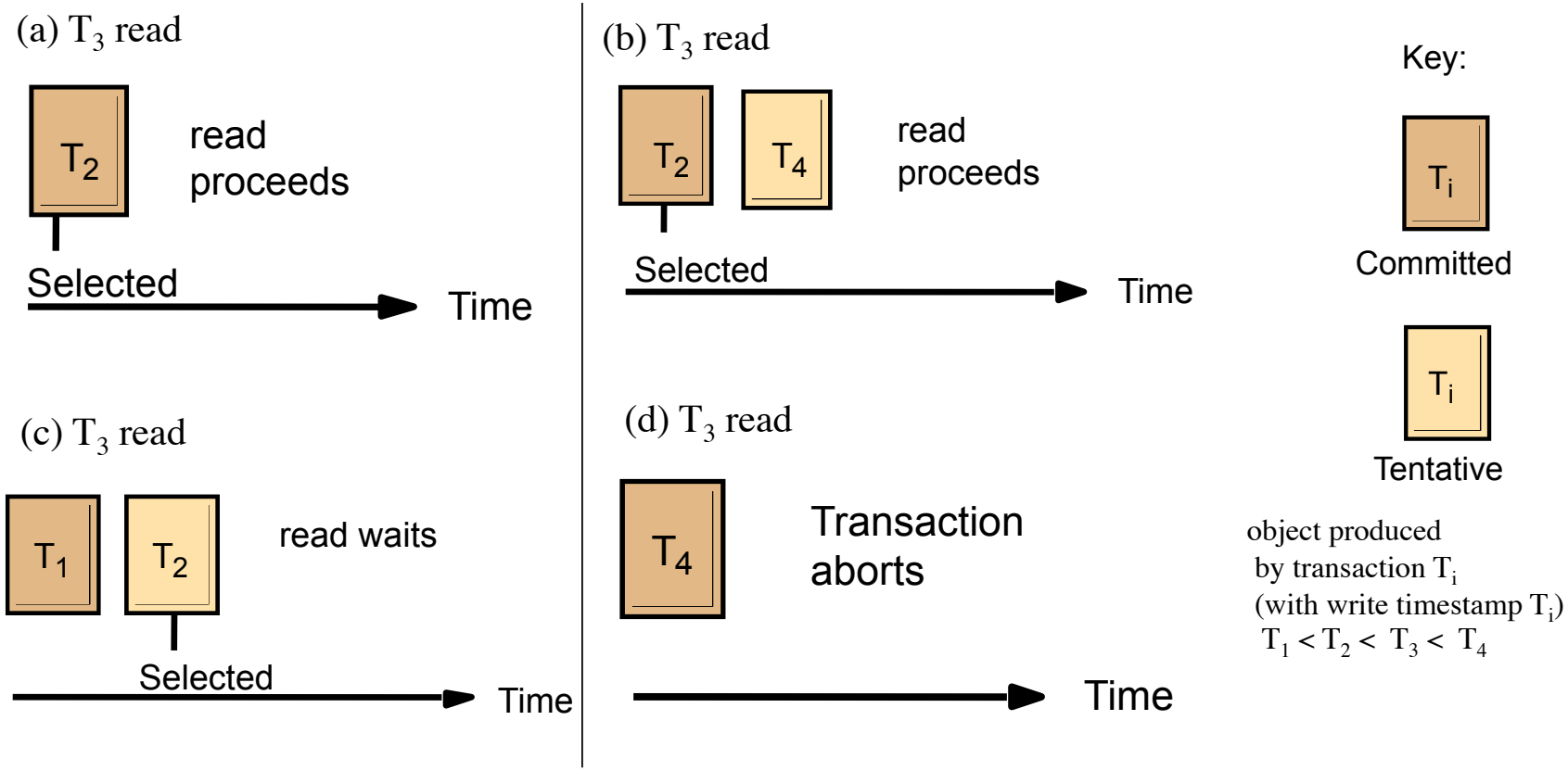
PAYAP UNIVERSITY

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.

if (T_c > write timestamp on committed version of D) {
    let $D_{selected}$ be the version of D with the maximum write timestamp ≤ 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

Read rule



Timestamps in transactions T and U

T	*U*	*Timestamps and versions of objects* A RTS	A WTS	B RTS	B WTS	C RTS	C 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*

PAYAP UNIVERSITY

Comparison

- 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