# Distributed Systems Course
# Transactions and Concurrency Control



## Introduction to transactions

- The goal of transactions
  - the objects managed by a server must remain in a consistent state
    - when they are accessed by multiple transactions and
    - in the presence of server crashes
- Recoverable objects
  - can be recovered after the
  - objects are stored in perm
- Failure model
  - transactions deal with cra
    failures of communication
- Designed for an asyn
  - It is assumed that messages may be delayed

As transactions use permanent storage
The failure model also deals with disks

File writes may fail
a) By writing nothing
b) By writing a wrong value, but checksums are used so that reads detect bad blocks
Therefore (a) and (b) are omission failures

Writing to the wrong block is an arbitrary failure.

## Operations of the *Account* interface

*deposit(amount)*
deposit amount in the account
*withdraw(amount)*
withdraw amount from the acc
*getBalance() → amount*
return the balance of the accou
*setBalance(amount)*
set the balance of the account t

Used as an example. Each *Account* is represented by a remote object whose

and each *Branch* of the bank is represented by a remote object whose interface *Branch* provides operations for creating a new account, looking one up by name and enquiring about the total funds at the branch. It stores a correspondence between account names and their remote object references

## Operations of the *Branch* interface

*create(name) → account*
create a new account with a given name
*lookUp(name) → account*
return a reference to the account with the given name
*branchTotal() → amount*
return the total of all the balances at the branch



## Atomic operations at server

- first we consider the synchronisation of client operations without transactions
- when a server uses multiple threads it can perform several client operations concurrently
- if we allowed *deposit* and *withdraw* to run concurrently we could get inconsistent results
- objects should be designed for safe concurrent access e.g. in Java use synchronized methods, e.g.
  - *public synchronized void deposit(int amount) throws RemoteException*
- *atomic operations* are free from interference from concurrent operations in other threads.
- use any available mutual exclusion mechanism (e.g. mutex)

## Client cooperation by means of synchronizing server operations

- Clients share resources via a server
- e.g. some clients update server objects and others access them
- servers with multiple threads require atomic objects
- but in some applications, clients depend on one another to progress
  - e.g. one is a producer and another a consumer
  - e.g. one sets a lock and the other waits for it to be released
- it would not be a good idea for a waiting client to poll the server to see whether a resource is yet available
- it would also be unfair (later clients might get earlier turns)
- Java *wait* and *notify* methods allow threads to communicate with one another and to solve these problems
  - e.g. when a client requests a resource, the server thread waits until it is notified that the resource is available



## Transactions

- Some applications require a sequence of client requests to a server to be atomic in the sense that:
  1. they are free from interference by operations being performed on behalf of other concurrent clients; and
  2. either all of the operations must be completed successfully or they must have no effect at all in the presence of server crashes.
- Transactions originate from database management systems
- Transactional file servers were built in the 1980s
- Transactions on distributed objects late 80s and 90s
- Middleware components e.g. CORBA Transaction service.
- Transactions apply to recoverable objects and are intended to be atomic.

Servers 'recover' - they are restarted and get their objects from permanent storage

## A client's banking transaction

*Transaction T:*
*a.withdraw(100);*
*b.deposit(100);*
*c.withdraw(200);*
*b.deposit(200);*

- This transaction specifies a sequence of related operations involving bank accounts named *A*, *B* and *C* and referred to as *a, b* and *c* in the program
- the first two operations transfer $100 from *A* to B
- the second two operations transfer $200 from *C* to *B*



## Atomicity of transactions

The atomicity has two aspects

- All or nothing:
  - it either completes successfully, and the effects of all of its operations are recorded in the objects, or (if it fails or is aborted) it has no effect at all. This all-or-nothing effect has two further aspects of its own:
  - failure atomicity:
    - the effects are atomic even when the server crashes;
  - durability:
    - after a transaction has completed successfully, all its effects are saved in permanent storage.
- Isolation:
  - Each transaction must be performed without interference from other transactions - there must be no observation by other transactions of a transaction's intermediate effects

Concurrency control ensures isolation

## Operations in the *Coordinator* interface

- transaction capabilities may be added to a server of

the client uses *OpenTransaction* to get *TID* from the coordinator

the client passes the *TID* with each request in the transaction

e.g. as an extra argument or transparently (The CORBA transaction service does uses 'context' to do this).

*openTransaction() -> trans;*

starts a new transaction and delivers a unique TID *trans*. This identifier will be used in the other operations in the transaction.

*closeTransaction(trans) -> (commit, abort);*

ends a transaction: a *co*

transaction has commit

it has aborted.

To commit - the client uses *closeTransaction* and the coordinator ensures that the objects are saved in permanent storage

*abortTransaction(trans);*

aborts the transaction.

To abort - the client uses *abortTransaction* and the coordinator ensures that all temporary effects are invisible to other transactions



## Transaction life histories

| Successful | Aborted by client | Aborted by server | |
|---|---|---|---|
| openTransaction | openTransaction | | openTransaction |
| operation | operation | | operation |
| operation | operation | | operation |
| • | • | server aborts | • |
| • | • | transaction → | • |
| operation | operation | | operation ERROR |
| | | | reported to client |
| closeTransaction | abortTransaction | | |

- A transaction is either successful (it commits)
  - the coordinator sees that all objects are saved in permanent storage
- or it is aborted by the client or the server
  - make all temporary effects invisible to other transactions
  - how will the client know when the server has aborted its transaction?

the client finds out next time it tries to access an object at the server.

## Concurrency control

- We will illustrate the 'lost update' and the 'inconsistent retrievals' problems which can occur in the absence of appropriate concurrency control
  - a lost update occurs when two transactions both read the old value of a variable and use it to calculate a new value
  - inconsistent retrievals occur when a retrieval transaction observes values that are involved in an ongoing updating transaction
- we show how serial equivalent executions of transactions can avoid these problems
- we assume that the operations *deposit*, *withdraw*, *getBalance* and *setBalance* are *synchronized* operations - that is, their effect on the account balance is atomic.



## The lost update problem

| **Transaction *T*:** | | **Transaction *U*:** | |
|---|---|---|---|
| *balance = b.getBalance();*<br>*b.setBalance(balance*1.1);*<br>*a.withdraw(balance/10)* | | *balance = b.getBalance();*<br>*b.setBalance(balance*1.1);*<br>*c.withdraw(balance/10)* | |
| *balance = b.getBalance();* | $200 | | |
| | | *balance = b.getBalance();* | $200 |
| | | *b.setBalance(balance*1.1);* | $220 |
| *b.setBalance(balance*1.1);* | $220 | | |
| *a.withdraw(balance/10)* | $80 | | |
| | | *c.withdraw(balance/10)* | $280 |

- the initial balances of accounts A, B, C are $100, $200. $300
- both transfer transactions increase B's balance by 10%

the net effect should be to increase *B* by 10% twice - 200, 220, 242.

but it only gets to 220. *U*'s update is lost.

## The inconsistent retrievals problem

| **Transaction *V*:** *a.withdraw(100)* *b.deposit(100)* | | **Transaction *W*:** *aBranch.branchTotal()* | |
|---|---|---|---|
| *a.withdraw(100);* | $100 | | |
| | | *total = a.getBalance()* | $100 |
| | | *total = total+b.getBalance()* | $300 |
| | | *total = total+c.getBalance()* | |
| *b.deposit(100)* | $300 | ⋮ | |

- *V* transfers $100 from *A* to *B* while *W* calculates branch total (which should be $600)

we see an inconsistent retrieval because *V* has only done the withdraw part when *W* sums balances of *A* and *B*

## Serial equivalence

- if each one of a set of transactions has the correct effect when done on its own
- then if they are done one at a time in some order the effect will be correct
- a *serially equivalent interleaving* is one in which the combined effect is the same as if the transactions had been done one at a time in some order
- the same effect means
  - the read operations return the same values
  - the instance variables of the objects have the same values at the end

The transactions are scheduled to avoid overlapping access to the accounts accessed by both of them

## A serially equivalent interleaving of *T* and *U* (lost updates cured)

| **Transaction *T*:** | | **Transaction *U*:** | |
|---|---|---|---|
| *balance = b.getBalance()* | | *balance = b.getBalance()* | |
| *b.setBalance(balance*1.1)* | | *b.setBalance(balance*1.1)* | |
| *a.withdraw(balance/10)* | | *c.withdraw(balance/10)* | |
| *balance = b.getBalance()* | $200 | | |
| *b.setBalance(balance*1.1)* | $220 | | |
| | | *balance = b.getBalance()* | $220 |
| | | *b.setBalance(balance*1.1)* | $242 |
| *a.withdraw(balance/10)* | $80 | | |
| | | *c.withdraw(balance/10)* | $278 |

- if one of *T* and *U* runs before the other, they can't get a lost update,
- the same is true if they are run in a serially equivalent ordering

their access to *B* is serial, the other part can overlap

## A serially equivalent interleaving of *V* and *W* (inconsistent retrievals cured)

| **Transaction *V*:** | | **Transaction *W*:** | |
|---|---|---|---|
| *a.withdraw(100);* | | *aBranch.branchTotal()* | |
| *b.deposit(100)* | | | |
| *a.withdraw(100);* | $100 | | |
| *b.deposit(100)* | $300 | | |
| | | *total = a.getBalance()* | $100 |
| | | *total = total+b.getBalance()* | $400 |
| | | *total = total+c.getBalance()* | |
| | | ... | |

- if *W* is run before or after *V*, the problem will not occur
- therefore it will not occur in a serially equivalent ordering of *V* and *W*
- the illustration is serial, but it need not be

we could overlap the first line of *W* with the second line of *V*

## *Read* and *write* operation conflict rules

- Conflicting operations: a pair of operations conflicts if their combined effect depends on the order in which they were performed
  - e.g. *read* and *write* (whose effects are the result returned by *read* and the value set by *write)*

| *Operations of different transactions* | | *Conflict* | *Reason* |
|---|---|---|---|
| *read* | *read* | No | Because the effect of a pair of *read* operations does not depend on the order in which they are executed |
| *read* | *write* | Yes | Because the effect of a *read* and a *write* operation depends on the order of their execution |
| *write* | *write* | Yes | Because the effect of a pair of *write* operations depends on the order of their execution |



## Serial equivalence defined in terms of conflicting operations

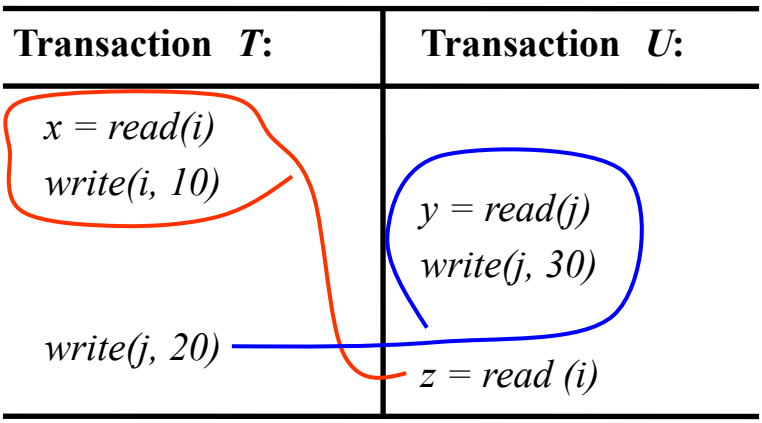
- For two transactions to be serially equivalent, it is necessary and sufficient that all pairs of conflicting operations of the two transactions be executed in the same order at all of the objects they both access
- Consider
  - T: x = read(i); write(i, 10); write(j, 20);
  - U: y = read(j); write(j, 30); z = read (i);
  - serial equivalence requires that either
    - T accesses i before U and T accesses j before U. or
    - U accesses i before T and U accesses j before T.

T and U access i and j

Which of their operations conflict?

- Serial equivalence is used as a criterion for designing concurrency control schemes

## A non-serially equivalent interleaving of operations of transactions *T* and *U*




- Each transaction's access to *i* and *j* is serialised w.r.t one another, but
- *T* makes all accesses to *i* before *U* does
- *U* makes all accesses to *j* before *T* does
- therefore this interleaving is not serially equivalent



## Recoverability from aborts

- if a transaction aborts, the server must make sure that other concurrent transactions do not see any of its effects
- we study two problems:
- 'dirty reads'
  - an interaction between a read operation in one transaction and an earlier write operation on the same object (by a transaction that then aborts)
  - a transaction that committed with a 'dirty read' is not recoverable
- 'premature writes'
  - interactions between write operations on the same object by different transactions, one of which aborts
- (getBalance is a read operation and setBalance a write operation)

## A dirty read when transaction *T* aborts

| Transaction *T*: | | Transaction *U*: | |
|---|---|---|---|
| *a.getBalance()* | | *a.getBalance()* | |
| *a.setBalance(balance + 10)* | | *a.setBalance(balance + 20)* | |
| *balance = a.getBalance()* | $100 | | |
| *a.setBalance(balance + 10)* | $110 | | |
| | | *balance = a.getBalance()* | $110 |
| | | *a.setBalance(balance + 20)* | $130 |
| *T* subsequently aborts. | | *commit transaction* | |
| *abort transaction* | | | |

•*U* reads *A*'s balance (which was set by *T*) and then commits

*U* has performed a *dirty read*

These executions are serially equivalent

What is the problem? *U* has committed, so it cannot be undone



## Recoverability of transactions

- If a transaction (like *U*) commits after seeing the effects of a transaction that subsequently aborted, it is not recoverable

**For recoverability:**
A commit is delayed until after the commitment of any other transaction whose state has been observed

- e.g. *U* waits until *T* commits or aborts
- if *T* aborts then *U* must also abort

So what is the potential problem?

## Cascading aborts

- Suppose that U delays committing until after T aborts.
  - then, U must abort as well.
  - if any other transactions have seen the effects due to U, they too must be aborted.
  - the aborting of these latter transactions may cause still further transactions to be aborted.
- Such situations are called cascading aborts

To avoid cascading aborts

transactions are only allowed to read objects written by committed transactions.

to ensure this, any *read* operation must be delayed until other transactions that applied a *write* operation to the same object have committed or aborted.

  - e.g. U waits to perform getBalance until T commits or aborts
- Avoidance of cascading aborts is a stronger condition than recoverability

For recoverability - delay commits

## Premature writes - overwriting uncommitted values

| **Transaction *T*:** | | **Transaction *U*:** |
|---|---|---|
| *a.setBalance(105)* | | *a.setBalance(110)* |
| | $100 | |
| *a.setBalance(105)* | $105 | |
| | | *a.setBalance(110)* $110 |

before *T* and *U* the balance of A was $100

serially equivalent executions of T and U

interaction between *write* operations when a transaction aborts

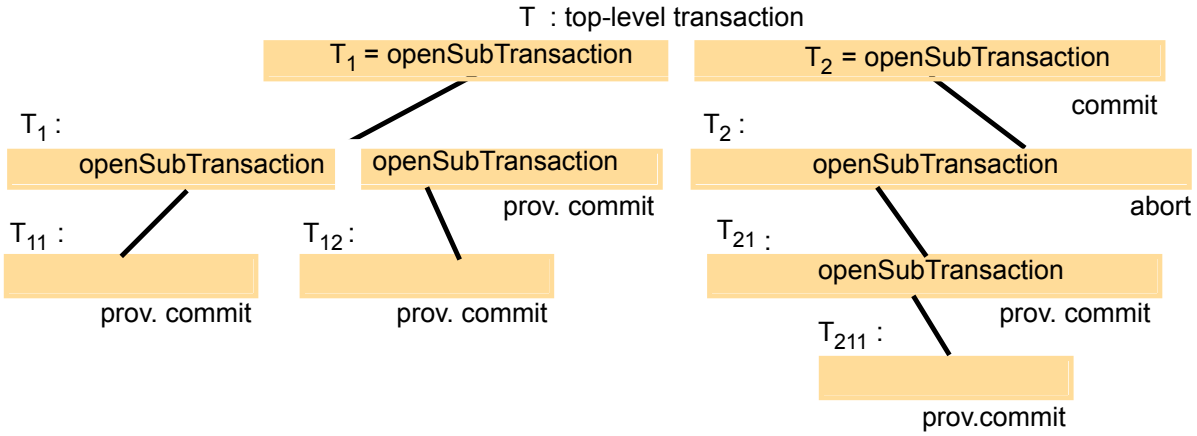
- some database systems keep 'before images' and restore them after aborts.
  - e.g. $100 is before image of T's write, $105 is before image of U's write
  - if U aborts we get the correct balance of $105,
  - But if U commits and then T aborts, we get $100 instead of $110

# Strict executions of transactions

- Curing premature writes:
  - if a recovery scheme uses before images
    - write operations must be delayed until earlier transactions that updated the same objects have either committed or aborted
- Strict executions of transactions
  - to avoid both 'dirty reads' and 'premature writes'.
    - delay both read and write operations
  - executions of transactions are called *strict* if both *read* and *write* operations on an object are delayed until all transactions that previously wrote that object have either committed or aborted.
  - the strict execution of transactions enforces the desired property of isolation
- *Tentative versions* are used during progress of a transaction
  - objects in tentative versions are stored in volatile memory



# Nested transactions




- transactions may be composed of other transactions
  - several transactions may be started from within a transaction
  - we have a top-level transaction and subtransactions which may have their own subtransactions

## Nested transactions

- To a parent, a subtransaction is atomic with respect to failures and concurrent access
- transactions at the same level (e.g. *T*1 and *T*2) can run concurrently but access to common objects is serialised
- a subtransaction can fail independently of its parent and other subtransactions
  - when it aborts, its parent decides what to do, e.g. start another subtransaction or give up
- The CORBA transaction service supports both flat and nested transactions



## Advantages of nested transactions (over *flat* ones)

- Subtransactions may run concurrently with other subtransactions at the same level.
  - this allows additional concurrency in a transaction.
  - when subtransactions run in different servers, they can work in parallel.
    - e.g. consider the branchTotal operation
    - it can be implemented by invoking getBalance at every account in the branch.
      - these can be done in parallel when the branches have different servers
- Subtransactions can commit or abort independently.
  - this is potentially more robust
  - a parent can decide on different actions according to whether a subtransaction has aborted or not

## Commitment of nested transactions

- A transaction may commit or abort only after its child transactions have completed.
- A subtransaction decides independently to *commit provisionally* or to *abort*. Its decision to abort is final.
- When a parent aborts, all of its subtransactions are aborted.
- When a subtransaction aborts, the parent can decide whether to abort or not.
- If the top-level transaction commits, then all of the subtransactions that have provisionally committed can commit too, provided that none of their ancestors has aborted.



## Summary on transactions

- We consider only transactions at a single server, they are:
- atomic in the presence of concurrent transactions
  - which can be achieved by serially equivalent executions
- atomic in the presence of server crashes
  - they save committed state in permanent storage
  - they use strict executions to allow for aborts
  - they use tentative versions to allow for commit/abort
- nested transactions are structured from sub-transactions
  - they allow concurrent execution of sub-transactions
  - they allow independent recovery of sub-transactions

# Introduction to concurrency control

- Transactions must be scheduled so that their effect on shared objects is serially equivalent
- for serial equivalence,
  - A. all access by a transaction to a particular object must be serialized with respect to another transaction's access.
  - B. all pairs of conflicting operations of two transactions should be executed in the same order.
- A server can achieve serial equivalence by serialising access to objects, e.g. by the use of **locks**
- to ensure (B), a transaction is not allowed any new locks after it has released a lock
  - If T was allowed to access A, then unlock it, then access B then access A again, another transaction U might access A while it was unlocked so we have T→ U and U → T at A
- *Two-phase locking* - has a 'growing' and a 'shrinking' phase



# Transactions T and U with exclusive locks

when *T* is about to use *B*, it is locked for *T*

when *U* is about to use *B*, it is still locked for *T* and *U* waits

| **Transaction *T*:** | | **Tr** | |
|---|---|---|---|
| *balance = b.getBalance()* | | *ba* | |
| *a.withdraw(bal/10)* | | *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* |

*U* can now continue

when *T* commits, it unlocks *B*

- initially the balances of A, B and C unlocked

the use of the lock on *B* effectively serialises access to *B*

## 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 - apply locks to small things e.g. bank balances
  - there are no assumptions as to granularity in the schemes we present



## Read-write conflict rules

- concurrency control protocols are designed to deal with conflicts between operations in different transactions on the same object
- we describe the protocols in terms of *read* and *write* operations, which we assume are atomic
- read operations of different transactions do not conflict
- therefore exclusive locks reduce concurrency more than necessary
- The 'many reader/single writer' scheme allows several transactions to read an object or a single transaction to write it (but not both)
- It uses read locks and write locks
  - read locks are sometimes called *shared locks*

## Lock compatibility

- The operation conflict rules tell us that:
  1. 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.
  2. 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 |

to enforce 1, a request for a write lock is delayed by the presence of a read lock belonging to another transaction

to enforce 2, a request for a read lock or write lock is delayed by the presence of a write lock belonging to another transaction

## Lock promotion

- **Lost updates** – two transactions read an object and then use it to calculate a new value.
- Lost updates are prevented by making later transactions delay their reads until the earlier ones have completed.
  - each transaction sets a read lock when it reads and then promotes it to a write lock when it writes the same object
  - when another transaction requires a read lock it will be delayed
- Lock promotion: the conversion of a lock to a stronger lock – that is, a lock that is more exclusive
  - demotion of locks (making them weaker) is not allowed

## Use of locks in strict two-phase locking

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 server applies locks when the read/write operations are about to be executed
- the server releases a transaction's locks when it commits or aborts



## Lock implementation

- The granting of locks will be implemented by a separate object in the server that we call the lock manager.
- the lock manager holds a set of locks, for example in a hash table.
- each lock is an instance of the class Lock and is associated with a particular object.
  - its variables refer to the object, the holder(s) of the lock and its type
- the lock manager code uses wait (when an object is locked) and notify when the lock is released
- the lock manager provides setLock and unLock operations for use by the server

# *Lock* class

```
public class Lock {
        private Object object;                          // the object being protected by the lock
        private Vector holders;          // the TIDs of current holders
        private LockType lockType;       // the current type
        public synchronized void acquire(TransID trans,   LockType aLockType ){
                while(/*another transaction holds the lock in conflicing mode*/) {
                        try {
                                wait();
                        }catch ( InterruptedException e){/*...*/ }
                }
                if(holders.isEmpty()) { // no TIDs  hold lock
                        holders.addElement(trans);
                        lockType  = aLockType;
                } else if(/*another transaction holds the lock, share it*/ ) ){
                        if(/* this transaction not a holder*/)
holders.addElement(trans);
                } else if (/* this transaction is a holder but needs a more exclusive lock*/)
                                lockType.promote();
                }
        }
```

Continues on next slide



# continued

```
public synchronized void release(TransID trans ){
                holders.removeElement(trans);       // remove this holder
                // set locktype to none
                notifyAll();
        }
}
```

## *LockManager* class

```
public class LockManager {
   private Hashtable theLocks;

   public  void setLock(Object object, TransID trans,  LockType
lockType){
            Lock foundLock;
            synchronized(this){
                        // find the lock associated with object
                        // if there isn't one, create it and add to the hashtable
            }
      foundLock.acquire(trans, lockType);
   }

  // synchronize this one because we want to remove all entries
   public synchronized void unLock(TransID trans) {
            Enumeration e = theLocks.elements();
            while(e.hasMoreElements()){
         Lock aLock = (Lock)(e.nextElement());
         if(/* trans is a holder of this lock*/ ) aLock.release(trans);
      }
   }
}
```



## Deadlock with write locks

| Transaction *T* | | Transaction *U* | |
|---|---|---|---|
| **Operations** | **Locks** | **Operations** | **Locks** |
| *a.deposit(100);* | write lock *A* | | |
| | | *b.deposit(200)* | write lock *B* |
| *b.withdraw(100)* | | | |
| ••• | waits for *U*'s | *a.withdraw(200);* | waits for *T*'s |
| | lock on *B* | ••• | lock on *A* |
| ••• | | ••• | |
| ••• | | ••• | |

- The deposit and withdraw methods are atomic.
- When locks are used, each of T and U acquires a lock on one account and then gets blocked when it tries to access the account the other one has locked.
- We have a '**deadlock**'.
- The lock manager must be designed to deal with deadlocks.

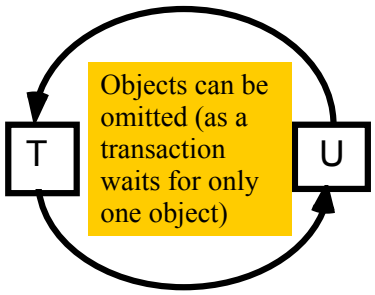
T accesses A → B
U accesses B → A

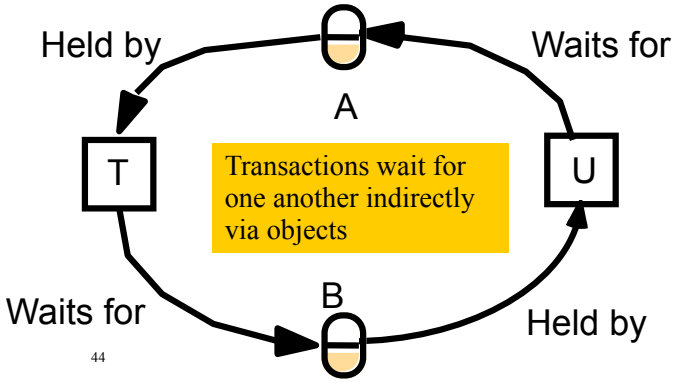
Is this serially equivalent?

## The wait-for graph for the previous figure

- Definition of deadlock
  - deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock.
  - a *wait-for graph* can be used to represent the waiting relationships between current transactions

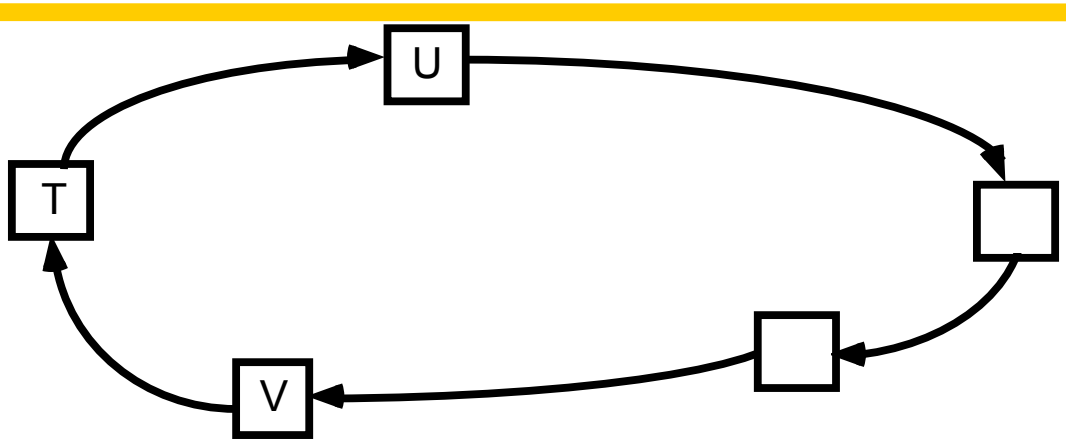
In a wait-for graph the nodes represent transactions and the edges represent wait-for relationships between transactions



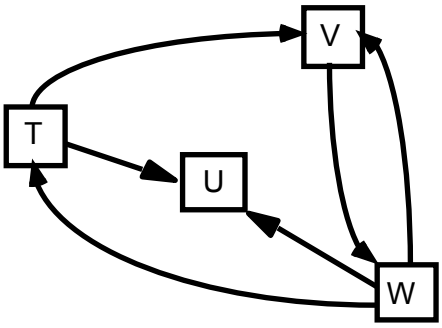


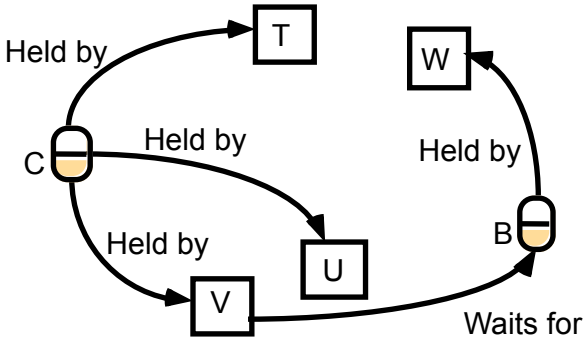

## A cycle in a wait-for graph




- Suppose a wait-for graph contains a cycle T → U → … → V → T
  - each transaction waits for the next transaction in the cycle
  - all of these transactions are blocked waiting for locks
  - none of the locks can ever be released (the transactions are deadlocked)
  - If one transaction is aborted, then its locks are released and that cycle is broken

## Another wait-for graph




- *T*, *U* and *V* share a read lock on *C* and
- *W* holds write lock on *B* (which *V* is waiting for)
- *T* and *W* then request write locks on *C* and deadlock occurs e.g. *V* is in two cycles - look on the left



## Deadlock prevention

- We can adopt policies and protocols which avoid the insurgence of deadlocks
- e.g. lock all of the objects used by a transaction when it starts
  - unnecessarily restricts access to shared resources.
  - it is sometimes impossible to predict at the start of a transaction which objects will be used.
- Deadlock can also be prevented by requesting locks on objects in a predefined order
  - but this can result in premature locking and a reduction in concurrency
- For these reasons, deadlock prevention is usually unrealistic

## Deadlock detection

- by finding cycles in the wait-for graph.
  - after detecting a deadlock, a transaction must be selected to be aborted to break the cycle
  - the software for deadlock detection can be part of the lock manager
  - it holds a representation of the wait-for graph so that it can check it for cycles from time to time
  - edges are added to the graph and removed from the graph by the lock manager's setLock and unLock operations
  - when a cycle is detected, choose a transaction to be aborted and then remove from the graph all the edges belonging to it
  - it is hard to choose a victim - e.g. choose the oldest or the one in the most cycles



## Timeouts on locks

- Lock timeouts can be used to resolve deadlocks
  - each lock is given a limited period in which it is invulnerable.
  - after this time, a lock becomes vulnerable.
  - provided that no other transaction is competing for the locked object, the vulnerable lock is allowed to remain.
  - but if any other transaction is waiting to access the object protected by a vulnerable lock, the lock is broken
    - (that is, the object is unlocked) and the waiting transaction resumes.
  - The transaction whose lock has been broken is normally aborted

- problems with lock timeouts
  - locks may be broken when there is no deadlock
  - if the system is overloaded, lock timeouts will happen more often and long transactions will be penalized
  - it is hard to select a suitable length for a timeout

## Resolution of the deadlock in slide 43

| Transaction T | | Transaction U | |
|---|---|---|---|
| **Operations** | **Locks** | **Operations** | **Locks** |
| *a.deposit(100);* | write lock *A* | | |
| | | *b.deposit(200)* | write lock *B* |
| *b.withdraw(100)* | | | |
| ••• | waits for *U*'s lock on *B* | *a.withdraw(200);* | waits for T's lock on *A* |
| | (timeout elapses) | ••• | |
| | *T*'s lock on *A* becomes vulnerable, unlock *A*, abort T | ••• | |
| | | *a.withdraw(200);* | write locks *A* unlock *A*, *B* |



## Increasing concurrency in locking schemes

- two-version locking
  - *optimistic* scheme, allows writing of tentative versions with reading of committed versions
- hierarchic locks
  - e.g. the *branchTotal* operation locks all the accounts with one lock whereas the other operations lock individual accounts (reduces the number of locks needed)

## Increasing concurrency: two-version locking

- Transaction can write tentative versions of objects
  - others read from the committed (original) version
- Read operations wait if another transaction is committing the same object
- Allows for more concurrency than read-write locks
  - writing transactions risk waiting or rejection when they attempt to commit
  - transactions cannot commit if other uncompleted transactions have read the objects
  - these transactions must wait until the reading transactions have committed



## Lock compatibility (*read*, *write* and *commit* locks)

- Three types of locks:
  - read lock, write lock, commit lock
- Transaction cannot get a read or write lock if there is a commit lock

| *For one object* | | *Lock to be set* | | |
|---|---|---|---|---|
| | | *read* | *write* | *commit* |
| *Lock already set* | *none* | OK | OK | OK |
| | *read* | OK | OK | wait |
| | *write* | OK | wait | |
| | *commit* | wait | wait | |

## Two-version locking

- When the transaction coordinator receives a request to commit:
  - converts all that transaction's write locks into commit locks
  - If any objects have outstanding read locks, transaction must wait until the transactions that set these locks have completed and locks are released
- Compare with read/write locks:
  - read operations delayed only while transactions are committed
  - read operations of one transaction can cause delay in the committing of other transactions



## Optimistic concurrency control

- the scheme is called **optimistic** because the likelihood of two transactions conflicting is low
- a transaction proceeds without restriction until the *closeTransaction* (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

## Optimistic concurrency control

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

- each transaction has three phases

1. 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
2. Validation phase
   - at closeTransaction 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
3. Update phase
   - If validated, the changes in its tentative versions are made permanent.
   - read-only transactions can commit immediately after passing validation.



## Validation of transactions

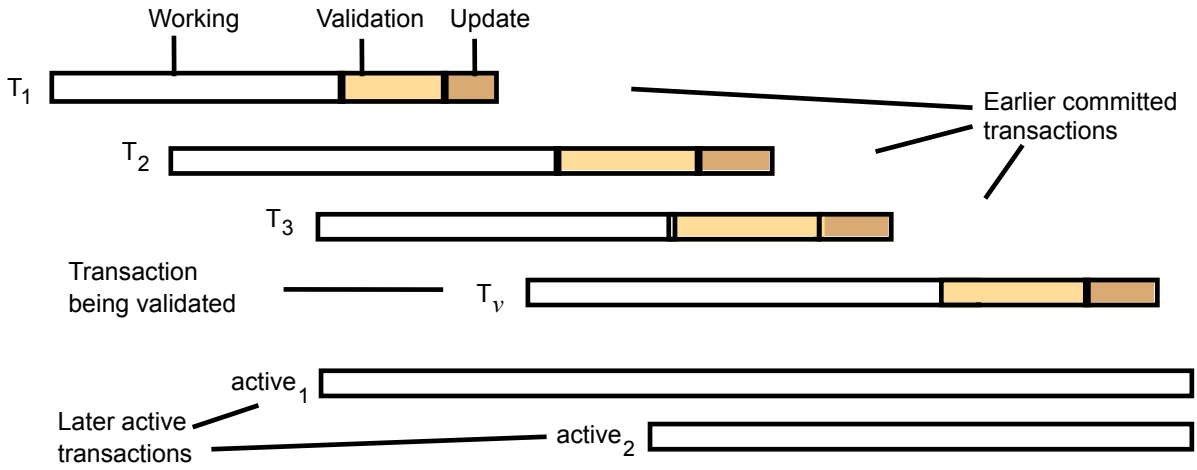
- We use the read-write conflict rules
  - to ensure a particular transaction is serially equivalent with respect to all other overlapping transactions
- each transaction is given a transaction number when it starts validation (the number is kept if it commits)
- the rules ensure serializability of transaction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

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

| $T_v$ | $T_i$ | *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 $T_i$ | |

# Backward validation of transactions

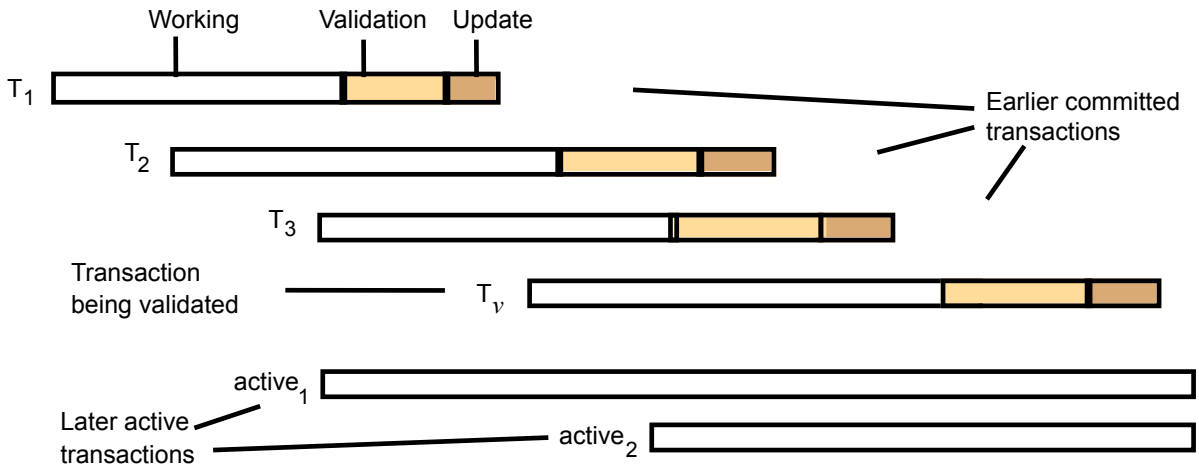
- Backward validation: check $T_v$ with preceding overlapping transactions




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



# Backward validation of transactions




- 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.
- Rule3 - (*write* vs *write*) assume no overlap of validate and commit.

## 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;
}
```

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

- Rule 1. the write set of $T_v$ is compared with the read sets of all overlapping active transactions
  - In backward validation, 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.

Forward validation of transaction Tv

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

read only transactions always pass validation

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

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

## Comparison of forward and backward validation

- 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 are larger 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

## Comparison of optimistic vs pessimistic

- Starvation
  - Optimistic strategies do not lead to deadlocks, but may lead to starvation
  - after a transaction is aborted, the client must restart it, but there is no guarantee it will ever succeed
- Starvation vs Deadlocks?
  - In both cases, aborted transactions are not guaranteed future success
  - so pessimistic strategies are not fairer than optimistic ones
  - Which is more likely, starvation or deadlock?
    - deadlock is less likely than starvation because locks make transactions wait
- But *distributed* deadlock detection is very hard to implement!
  - Hence in distributed settings, optimistic strategies are often preferred

## 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

- 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
- 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
  - When a write operation is accepted it is put in a tentative version and given a write timestamp
  - When a read operation is accepted it is directed to the tentative version with the maximum write timestamp less than the transaction timestamp

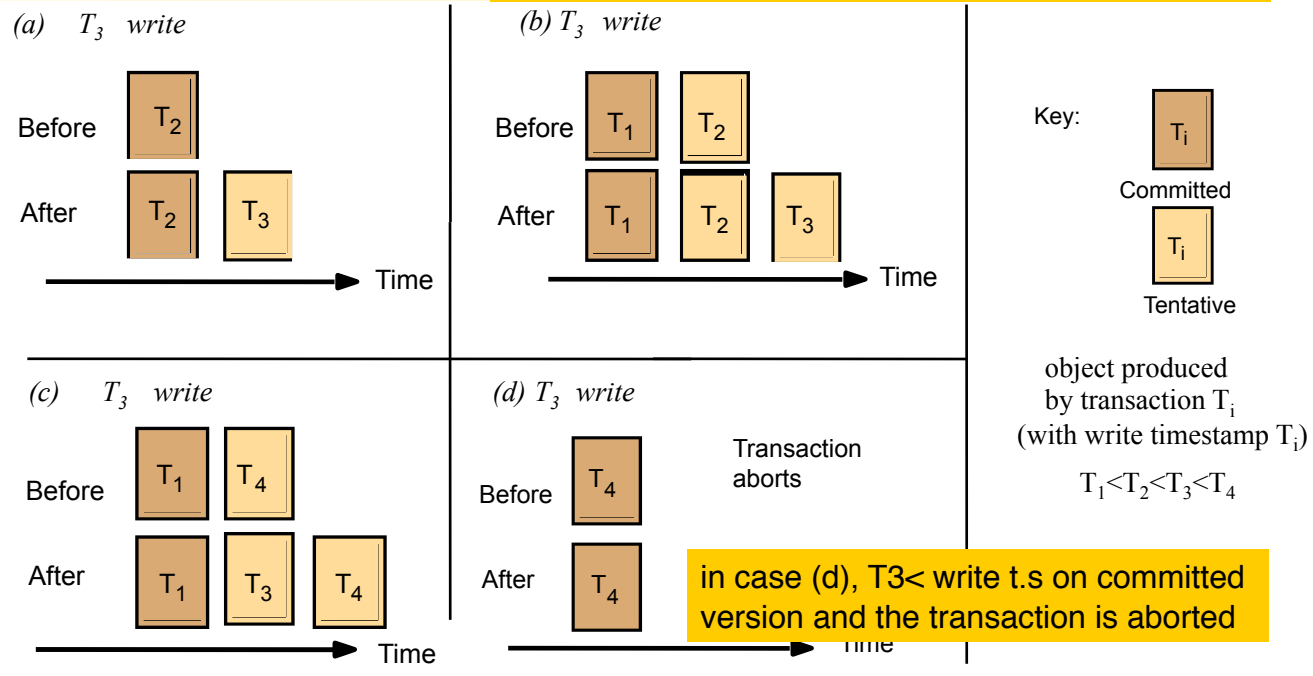
# Operation conflicts for timestamp ordering

- $T_c$ is the current transaction, $T_i$ are other transactions
- $T_i > T_c$ means $T_i$ is later than $T_c$

| 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. |

# Write operations and t

in cases (a), (b) and (c) T3> write t.s on committed version and a tentative version with write t.s T3 is inserted at an appropriate place in the list of versions




in case (d), T3< write t.s on committed version and the transaction is aborted

- this illustrates the versions and timestamps, when we do *T*3 *write*. For write to be allowed, T3≥ maximum read timestamp (not shown)

# Timestamp ordering write rule

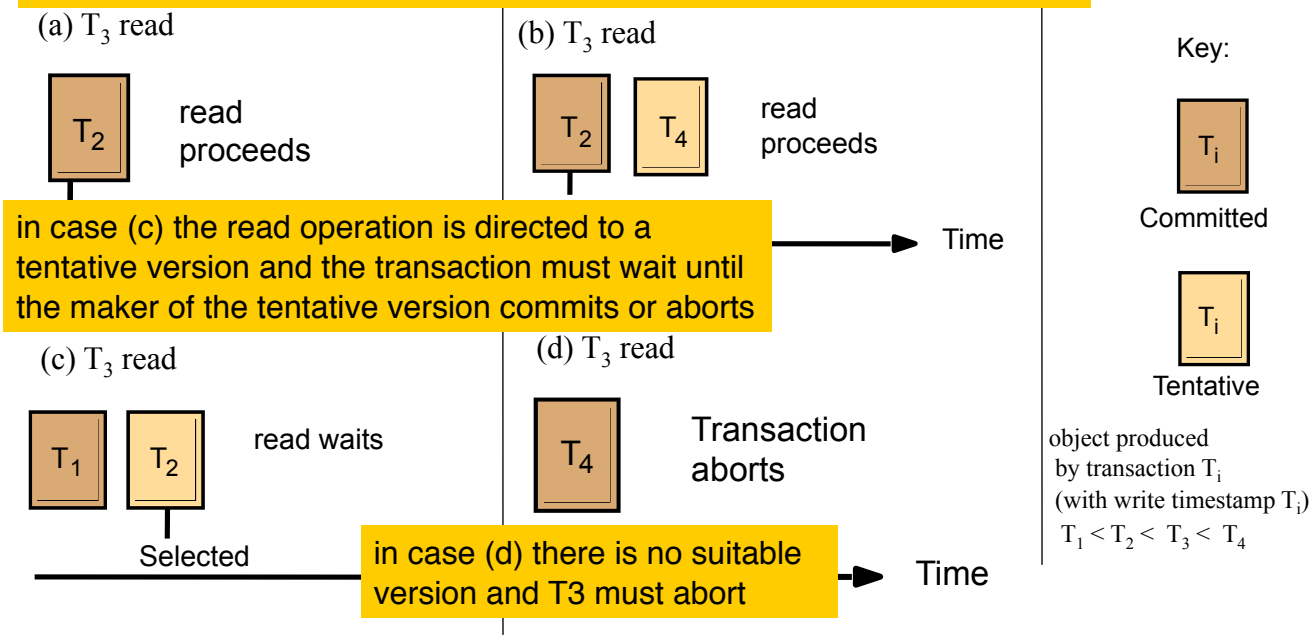
- by combining rules 1 (write/read) and 2 (write/write) we have the following rule for deciding whether to accept a write operation requested by transaction $T_c$ on object $D$
  - rule 3 does not apply to writes

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 $T_c$



# Read operations and timestamps

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




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

## 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 ≤ $T_c$
    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 $T_c$



## 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
- the method avoids deadlocks, but is likely to suffer from restarts

## 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
    - allows more concurrency by keeping multiple committed versions
      - late read operations need not be aborted

## Timestamps in transactions *T* and *U*

| *T* | *U* | *Timestamps and versions of objects* A | B | C |
|---|---|---|---|---|
| | | *RTS WTS* {} ***S*** | *RTS WTS* {} ***S*** | *RTS WTS* {} ***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* |

## Comparison of methods for concurrency control

- **pessimistic methods** (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
  - **optimistic concurrency control** allows transactions to proceed without any form of checking until they are completed.
    - Validation is carried out. Starvation can occur.
  - **timestamp ordering** uses all three - no deadlocks
    - ordering according to the time transactions start.

## Examples of Internet-based applications

Many Internet-based applications use optimistic forms of concurrency control followed by conflict resolution

- **Dropbox:** distributed file sharing service
  - uses an optimistic form of concurrency control, keeping track of consistency and preventing clashes between users' updates, at the granularity of whole files.
  - If two users make concurrent updates to the same file, the first write will be accepted and the second rejected (possibly generating conflicting version).
  - Previous versions are kept for rollbacks.
- **Google Apps:** a cloud service that provides web-based applications (word processor, spreadsheet, …).
  - Users are left to resolve conflicts on documents which they edit simultaneously
  - but conflicts are generally avoided because users are continuously aware of each other's activities.
  - If two users access the same cell simultaneously, the last update wins.



## Examples of Internet-based applications (cont.)

- **Amazon Dynamo**: key-value distributed structured storage system, inside Amazon S3
  - Uses single *get* and *put* operations rather than transactions
  - Does not provide isolation, but only a weak consistency.
  - Optimistic methods are used for concurrency control: in cases two versions differ, they must be reconciled.
    - Application logic can be used to merge versions;
    - otherwise, timestamp-based reconciliation is applied: 'last write wins' – the version with the largest timestamp becomes the new one.
- **Riak** is similar to Dynamo