Transactions and Concurrency Control

- Introduction
- Transactions
- Nested transactions
- Locks
- Optimistic concurrency control
- Timestamp ordering
- Summary

Introduction

- The goal of transactions
  - All objects remain in a consistent state when they are accessed by multiple transactions and in the presence of server crashes
- Enhance reliability
  - Recovery from failures
    - Recoverable objects.
  - Record in permanent storage
- The banking example

Simple synchronization (without transaction)

- Atomic Operations at Server:
  - Multi-threaded banking server
  - Only one thread can access an *account* at a time
    - *Public synchronized void deposit(int amount) {…}*
      - *Object is locked and another thread is blocked until lock released.*
  - *Atomic Operations: Operations that are free from interference from concurrent operations being performed in other threads are called atomic operations.*

- Enhancing client operations by synchronization of server operations:
  - E.g. producer and consumer
  - Wait and notify methods in JAVA
  - Wait(): gives up its lock and suspends itself as a single atomic operation.
  - Notify(): completes execution of that method before releasing the lock of an object.

Failure model for transactions [lamport1981]

- **Writes to permanent storage may fail**
  - Write nothing or wrong value
  - file storage may decay
  - reading bad data can be detect (by checksum)
- **Servers may crash occasionally**
  - Memory recover to the last updated state
  - continue recovery using information in permanent storage
  - no arbitrary failure
- **An arbitrary delay of a message**
  - A message may be lost, duplicated or corrupted
  - The recipient can detect corrupted messages

Transactions and Concurrency Control

- Introduction
- Transactions
- Nested transactions
- Locks
- Optimistic concurrency control
- Timestamp ordering
- Summary

Transactions

- What is a transaction?
  - A sequence of separate operations that execute in a atomic manner in the presence of multiple clients and server crashes.
    - Free from interference by operations that belong to different transaction
    - Nothing-or-all semantics of the transaction
- An example

Context of Transaction

- Access to database
- Transactional File server.
- As a middleware.
- Two aspects of atomicity:
  - All or nothing
    - Failure atomicity: effects are atomic even when the server crashes.
    - Durability: after successful completed transaction, all effects are saved in permanent storage.
  - Isolation: intermediate effect of transaction must not be visible to other transaction.

Use a transaction

- Transaction coordinator
  - Each transaction is created and managed by a coordinator
- Result of a transaction
  - Success
  - Aborted
    - Initiated by client
    - Initiated by server
- Example

Concurrency Control

- Process of managing simultaneous operations on the database without having them interfere with one another.
- Prevents interference when two or more users are accessing database simultaneously and at least one is updating data.
- Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.

Problems of Concurrency Control

Lost Update Problem:

Successfully completed updates is overridden by another user.

E.g. Account balance is $100

T1 withdraws $10

T2 deposits $100

Final balance would be $190.

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	read(bal_x)	100
t_3	read(bal_x)	$bal_x = bal_x + 100$	100
t_4	$bal_x = bal_x - 10$	write(bal_x)	200
t_5	write(bal_x)	commit	90
t_6	commit		90

Lost T2's update: can be avoided by preventing T1 from reading balx until after update.

Concurrency Control

- Inconsistent Retrieval Problem:
  - Occurs when transaction reads several values but second transaction updates some of them during execution of first.
  - T6 : is totalling balances of account x($100), y ($50), and z($25)
  - Meantime T5 has transferred $10 from balx to balz. So T6 is now has wrong result.

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	read(bal_x)	read(bal_x)	100	50	25	0
t_4	$bal_x = bal_x - 10$	sum = sum + bal_x	100	50	25	100
t_5	write(bal_x)	read(bal_y)	90	50	25	100
t_6	read(bal_z)	sum = sum + bal_y	90	50	25	150
t_7	$bal_z = bal_z + 10$		90	50	25	150
t_8	write(bal_z)		90	50	35	150
t_9	commit	read(bal_z)	90	50	35	150
t_{10}		sum = sum + bal_z	90	50	35	185
t_{11}		commit	90	50	35	185

Problem avoided by preventing T6 from reading balx and balz until T5 completed updates.

Conflicting operations

- Conflict between a pair of operations
  - The combined effect depends on the order in which they are executed
  - Effect: Value of the object set by write operation and the result returned by a read operation.
- Conflicting rules
- Serial equivalence of two transactions
  - All pairs of conflicting operations of the two transactions be executed in the same order at all of the objects they both access
- A non-serially equivalent example

- Serial equivalence required in previous example by one of the following conditions:
  - T accesses i before U and T accesses j before U.
  - U accesses I before T and U accesses j before T.
- Serial Equivalence: criteria for concurrency control. 3 approaches
  - Locking
  - Optimistic Concurrency Control
  - Timestamp ordering

Recoverability from aborts

- **Dirty reads**
  - Isolation: transaction do not see the uncommitted state of other transaction.
  - Recoverability of transactions:
    - Strategy: any commits must be delayed until after the commitment of any other transaction whose uncommitted state has been observed

- **Cascading aborts**
  - Aborted of one transaction cause more transactions to be aborted.
  - Strategy: any *read* operation must be delayed until other transactions that applied a write operation to the same object have committed or aborted

Premature writes

- Some DBMS implement the action of abort by restoring "before images" of all the writes of a transaction
  - \$100 before image of T's write
  - \$105 before image of U's write
    - If U aborts- correct balance \$105
    - If U commits and T aborts- correct: \$110 Wrong:\$100
    - If T & U aborts- Correct : \$100 Wrong: \$105
- Strategy: any *write* operations must be delayed until earlier transactions that updated the same objects have either committed or aborted

Recoverability from aborts …*continued*

- Strict executions of transactions
  - Delays both *read* and *write* operations on an object until all transactions that previously *wrote* that object have either committed or aborted
  - Property of isolation.
- Tentative versions
  - Recoverable object: all of the update operations performed during a transaction are done in tentative versions of objects in volatile memory
  - Update operation: a transaction store value in the transaction's own private set.
  - Access Operation: read values from own private set or if fails take from object.

Transactions and Concurrency Control

- Introduction
- Transactions
- Nested transactions
- Locks
- Optimistic concurrency control
- Timestamp ordering
- Summary

The advantages of nested transactions

- Nested transactions
- Allowing transactions to be composed of other transactions.
- Flat Transactions.
- The advantages of nested transactions
  - Additional concurrency
    - Subtransactions at one level may run concurrently with other subtransactions at the same level
    - E.g. concurrent *getBalances* in *branchTotal* operation
  - More robust
    - Subtransactions can commit or abort independently

The rules for commitment of nested transactions

- Transaction commit(abort) after its child complete
  - A transaction may commit or abort only after its child transactions have completed
- Child completes: commit provisionally or abort
  - When a subtransaction completes, it makes an independent decision either to commit provisionally or to aborts. Its decision to abort is final
- Parent abort, children abort
  - When a parent aborts, all of its subtransactions are aborted

The rules for commitment of nested transactions

- Child abort, parent abort or not
  - When a subtransaction aborts, the parent can decide whether to abort or not
- Top level transaction commit, all provisionally committed subtransactions commit
  - 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

Transactions and Concurrency Control

- Introduction
- Transactions
- Nested transactions
- Locks
- Optimistic concurrency control
- Timestamp ordering
- Comparison of methods for concurrency control
- Summary

Simple exclusive locks

- **Lock any object that is about to be used by any operation of a client's transaction**

Transaction : *T*

bal = b.getBalance()

*b.setBalance(bal*1.1)*

a.withdraw(bal/10)

Operations	Locks
openTransaction	
bal = b.getBalance()	Lock *B*
*b.setBalance(bal*1.1)*	
a.withdraw(bal/10)	lock *A*
closeTransaction	unlock *A,B*

Transaction : *U*

bal = b.getBalance()

*b.setBalance(bal*1.1)*

c.withdraw(bal/10)

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

Two phase locking

- To ensure serial equivalence of any two transactions
  - A transaction is not allowed any new locks after it has released a lock
    - Growing phase: acquire locks
    - Shrinking phase: release locks
- Strict two-phase locking
  - Any locks applied during the progress of a transaction are held until the transaction commits or aborts
    - In fact, the lock between two reads are unnecessary

Lock rules

- **Lock granularity**
  - as small as possible: enhance concurrency
- **Read lock / write lock**
  - Before access an object, acquire its lock firstly
- **Lock compatibility**
  - If a transaction *T* has already performed a *read* operation on an 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 an object, then a concurrent transaction *U* must not *read* or *write* that object until T commits or aborts

Lock rules … *continued*

- Prevent *lost update* and *inconsistent retrieval*
- Promotion of a lock
  - From read lock to write lock
  - Promotion can not be conducted if the read lock is shared by another transaction
- Two-phase locking implementation

Lock implementation

- **The Lock class**
  - The identifier of the locked object
  - The identifiers of the transactions that currently hold the lock
  - A lock type
- **The lock manager**
  - All requests to set locks and to release them are sent to the an instance of *Lockmanager*

Locking requirement for nested transactions

- **Each set of nested transaction**
  - Must be prevented from observing the partial effects of any other set of nested transactions
    - Locks that are acquired by subtransactions are inherited by its parent when it completes
    - Prevent other set of nested transactions getting the locks

Locking requirement for nested transactions

- **Each transaction within a set of nested transactions**
  - Must be prevented from observing the partial effects of the other transactions in the set
    - Parent is not allowed to run concurrently with their children, parent's lock could be allocated to its children temporarily
    - Subtransactions at the same level are allowed to run concurrently
    - So that, subtransactions in the set access same object in a sequential order

Locking rules for nested transactions

- **Commit**
  - When a subtransaction commits, its locks are inherited by its parent, allowing the parent to retain the locks in the same mode as the child
- **Abort**
  - When a subtransaction aborts, its locks are discarded. If the parent already retains the locks, it can continue to do so.

Dead locks

- **Example of dead lock with write locks**
- **Definition of deadlock**
  - each member of a group of transactions is waiting for some other member to release a lock
  - *Wait-for graph*
- **Example of a cycle in wait-for graph**
- **Example of a deadlock happening**
  - T, U and V share a read lock on object C
  - W holds a write lock on object B
  - V is waiting to obtain lock
  - T and W request write lock on object C.
  - Deadlock: T waits for U &V, V waits for W and W waits for T, U, V

V->W->T->V   V->W->V

Solution: Transaction V should be aborted.

Deadlock prevention

- Lock all of the objects used by a transaction when it starts
  - A single atomic step
  - Restrict access to shared resources
  - Impossible to predict at the start of a transaction which objects will be used
- Request locks on objects in a predefined order
  - premature locking
  - Reduction in concurrency
- Upgrade Lock: A transaction with upgrade lock on data item is permitted to read.

Deadlock detection

- Lock manager
  - find cycles in the wait-for graph periodically
  - Select one transaction to abort
- Timeouts
  - Each lock is *invulnerable* in a limited period *T*
  - After *T*, lock becomes *vulnerable*
  - If no other transactions competes for the *vulnerable* lock, the original transaction remains it
  - Otherwise, the lock is broken and the original transaction aborts

Two-version locking

- Read/Write
  - Write to tentative versions of objects
  - Read from the committed version
- <u>Locking rules</u>
  - Read lock: set on an object when attempt to read it
  - write lock: set on an object when attempt to write it
  - commit lock: convert the write lock to commit lock of the object when attempt to commit it

Two-version locking …*continued*

- Significance
  - Read is delayed only while the transaction is being committed
  - More concurrency than read-write locks

Hierarchic locks

- Mixed granularity locks
- Locking rules
  - Read lock, write lock, intentional read lock, intentional write lock
    - Control the conflicts between parent access and children access
  - Before a child node is granted a read/write lock, an intention to read/write lock is set on the parent node and its ancestors
- Significance
  - Reduce the number of locks
  - Control the granularity of locks

Transactions and Concurrency Control

- Introduction
- Transactions
- Nested transactions
- Locks
- Optimistic concurrency control
- Timestamp ordering
- Comparison of methods for concurrency control
- Summary

Pessimistic/Optimistic measures

- **Lock**
  - Overhead: Even for read-only operation
  - Deadlock: Unsatisfactory resolution
  - Reduce concurrency: Restrict two stage locking scheme
- **Optimistic measures**
  - Observation
    - The likelihood of two transactions accessing the same object is low
  - Scheme
    - No check when accessing object
    - Check conflict when committing
    - If there is conflict, abort transactions

Optimistic concurrency control

- **Working phase**
  - a *tentative version* of objects per transaction
    - Initially, it is a copy of the most recently committed version
    - Read are performed on the tentative version
    - Written values are recorded as tentative version
  - *Reading set* / *write set* per transaction
- **Validation phase**
  - check the conflicts between overlapped transactions when *closeTransaction* is issued
    - Success: commit
    - Fail: abort
- **Update phase**
  - Updates in tentative versions are made permanent

Validation of transaction

- Transaction number
  - Each transaction is assigned a transaction number when it enters the validation phase
  - An integer number assigned in ascending sequence
  - Transactions enter validation phase according to the their transaction number
  - Transactions commit according to the transaction number
    - Since the validation and update phase are short, so there is only one transaction at a time
- Conflict rules
  - T_v is serializable with respect to an overlapping transaction T_i

Backward validation Example

- Test the previous overlapped transactions
  - *startTn*
    - The biggest transaction number assigned to some other committed transaction at the time when transaction T_v started its working phase
  - *finishTn*
    - The biggest transaction number assigned to some other committed transactionat the time when T_v entered the validation phase
  - Serial equivalence of all committed transactions
    - Since backward validation can ensure the result that T_v commits after all previously committed transactions, so all transactions are committed in a serial equivalent order

Backward validation … *continued*

- Backward validation algorithm

Boolean valid = true

For (int $T_i = startT_n + 1$; $T_i <= finishT_n$; T_i ++){

    if (read set of T_v intersects write set of T_i) valid = false

}

- How to resolve a conflict?
  - Abort T_v

Forward validation **Example**

- *Active transactions*
    - Transactions that are still in their working phase when T_v enter validation phase
  - Serial equivalence of all committed transactions
    - Since forward validation can ensure the result that T_v commits before all behind committed transactions
- **Algorithm**

Boolean valid = true
for (int T_{id} = $active_1$; T_{id} <= $active_n$; T_{id} ++){
if (write set of T_v intersects read set of T_{id})
valid = false
}

Forward validation …*continued*

- How to resolve a conflict?
  - Defer the validation until a later time when the conflicting transactions have finished
    - Some conflicting transactions may have aborted
  - Abort all the conflicting active transactions and commit the transaction being validated
  - Abort the transaction being validated
    - The future conflicting transactions may abort, so the aborting becomes unnecessary

Comparison of forward and backward validation

- **Backward validation**
  - Overhead of comparison
    - for read set is bigger than write set, so comparison in backward validation is heavier than that in forward validation
  - Overhead of storage
    - Storing old write sets until they are no longer needed
- **Forward validation**
  - Overhead of time
    - To validate a transaction must wait until all active transactions finished

Transactions and Concurrency Control

- Introduction
- Transactions
- Nested transactions
- Locks
- Optimistic concurrency control
- Timestamp ordering
- Comparison of methods for concurrency control
- Summary

The basic idea

- **An unique timestamp**
  - Each transaction is assigned an unique timestamp value when it starts
  - Defines its position in the time sequence of transactions
  - Can not lead to deadlock
- **Conflicts rule**
  - A transaction's request to write an object is valid only if that object was last read and written by earlier transactions
  - A transactions's request to read an object is valid only if that object was last written by an earlier transaction

Timestamp ordering mechanism

- **tentative versions**
  - Write operations are recorded in tentative versions
- **A set of write timestamp & a set of read timestamp per object**
  - The write timestamp of the committed object is earlier than that of any of its tentative version
  - The set of read timestamp can be represented by its maximum member
  - Read operation is directed to the version with the maximum write timestamp which is earlier than the read timestamp

Timestamp ordering write rule

- Algorithm that decides whether to accept a write operation requested by transaction T_c on object D
- Example

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

Timestamp ordering read rule

- **Algorithm that decides whether to accept immediately, to wait or to reject a read operation requested by transaction T_c on object D**
- **Example**

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

Multiversion timestamp ordering

- Idea
  - Each object maintains a list of old committed versions as well as tentative versions
  - When a read arrives late, it can be allowed to read from an old committed version, but not reject
- Conflict rules
  - T_c must not write objects that have been read by any T_i where $T_i >= T_c$
  - Rule 2 has been met by multiversion committed object
  - Rule 3 has been met by tentative version object

Multiversion timestamp ordering write rules

- The server direct the read operation to the most recent version of an object
- Example

if (read timestamp of $D_{maxEarlier}$ <= T_c)

perform *write* operation on a tentative version of D with write timestamp T_c

else /* write is too late */

Abort transaction T_c

Transactions and Concurrency Control

- Introduction
- Transactions
- Nested transactions
- Locks
- Optimistic concurrency control
- Timestamp ordering
- Comparison of methods for concurrency control
- Summary

Timestamp ordering vs. two phase locking

- Timestamp ordering
  - Decide the serialization order statically
  - Better than locking for read-dominated transactions
- Two phase lock
  - Decide the serialization order dynamically
  - Better than timestamp ordering for update-dominated transactions
- Both are pessimistic methods

Pessimistic methods vs. optimistic methods

- Optimistic methods
  - Efficient when there are few conflicts
  - A substantial amount of work may have to be repeated when a transaction is aborted
- Pessimistic methods
  - Less concurrency but simple in relative to optimistic methods

Transactions and Concurrency Control

- Introduction
- Transactions
- Nested transactions
- Locks
- Optimistic concurrency control
- Timestamp ordering
- Comparison of methods for concurrency control
- Summary

Summary

- **ACID of transaction**
  - Atomic, Consistency, Isolation, Duration
- **Nested transactions**
  - Concurrent execution of subtransactions in separate servers
  - Independent recovery of parts of a transaction
- **Concurrency control**
  - Criterion: serial equivalence
  - Two problems that should avoid
    - Lost update, inconsistent retrieval
  - Recoverability from aborts
    - Avoid dirty read, premature writes

- **Concurrency control methods**
  - **Two strict-phase locking**
    - When conflicts happen, delay some operations
    - Deadlock prevention & detection
  - **Timestamp ordering**
    - Order transactions accesses to objects
    - There may be transactions aborting
    - Multiversion timestamp ordering: effective
  - **Optimistic methods**
    - Allow transactions to proceed without any form of conflicts checking
    - Detect conflicts when validating
    - Backward validation / forward validation
    - Effective in expense of overhead in storage and complexity

Operations of the *Account* Interface

deposit(amount)

deposit amount in the account

withdraw(amount)

withdraw amount from the account

getBalance() -> amount

return the balance of the account

setBalance(amount)

set the balance of the account to amount

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

A client's banking transaction

Transaction T:

a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);

Operations in *Coordinator* interface

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 *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.

abortTransaction(trans);

aborts the transaction.

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*		

The lost update problem

a=100 b=200 c=300

Transaction *T*	**Transaction *U***
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*	*balance = b.getBalance();* *b.setBalance(balance*1.1);* *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 inconsistent retrieval problem

a=200 b=200

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

A serially equivalent interleaving of *T* and *U*

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

A serially equivalent interleaving of *V* and *W*

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

Read and *write* operation conflict rules

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* operation depends on the order of their execution

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

Transaction *T*:	Transaction *U*:
$x = read(i)$	
$write(i, 10)$	
	$y = read(j)$
	$write(j, 30)$
$write(j, 20)$	
	$z = read(i)$

A dirty read when transaction *T* aborts

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

If *T* aborts and *U* commits, the final *balance* is \$130 which is wrong

Overwriting uncommitted values

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

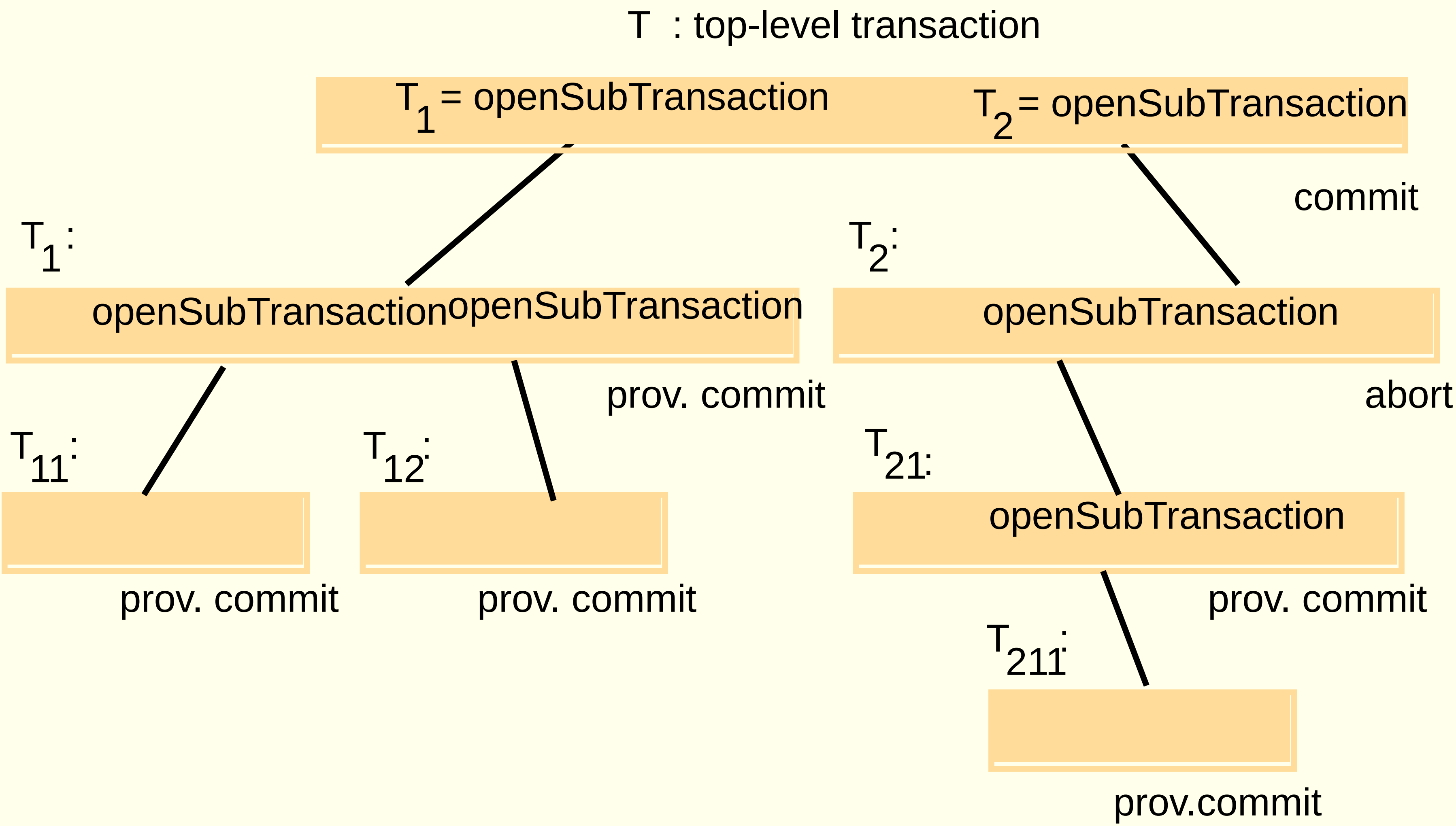
before image of *T* is *a=\$100*, before image of *U* is *a=\$105*.

If T commits but U aborts, then *a* is restored to *\$105* which is correct.

If *U* commits but *T* aborts, then *a* is restored to *\$100* which should be *\$110*

If *T* abort then *U* aborts, then *a* is restored to *\$105* which should be *\$100*

Nested transactions



Lock compatibility

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

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.

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();
    }
}
```

Lock class … *continued*

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

Lock manager class

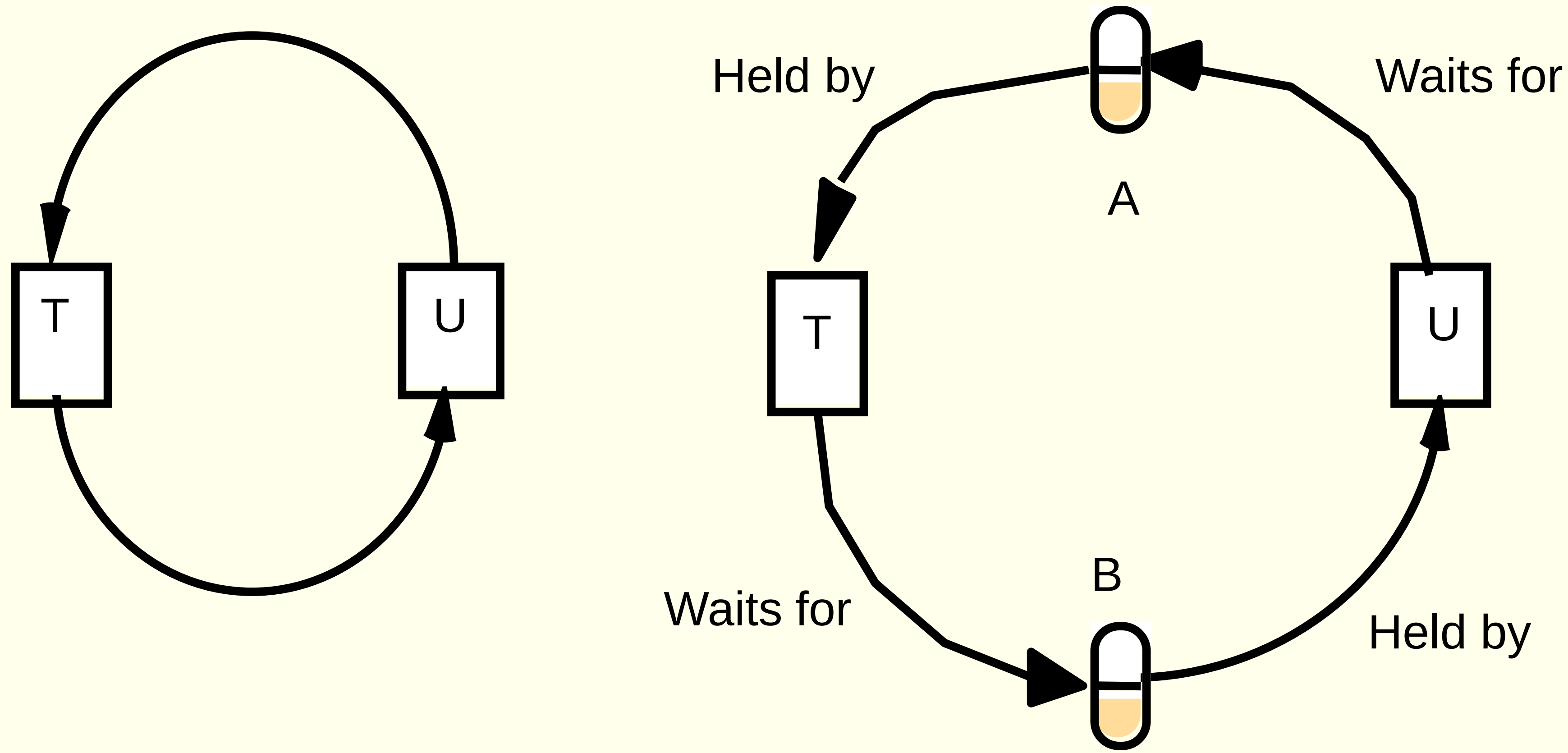
```
public class LockManager {
   private Hashtable theLocks;
   public  void setLock(Object object, TransID trans,  LockType
lockType){
         Lock foundLock;
         synchronized(this){
                  // find the lock associated with object
                  // if there isn't one, create it and 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);
      }
   }
}
```

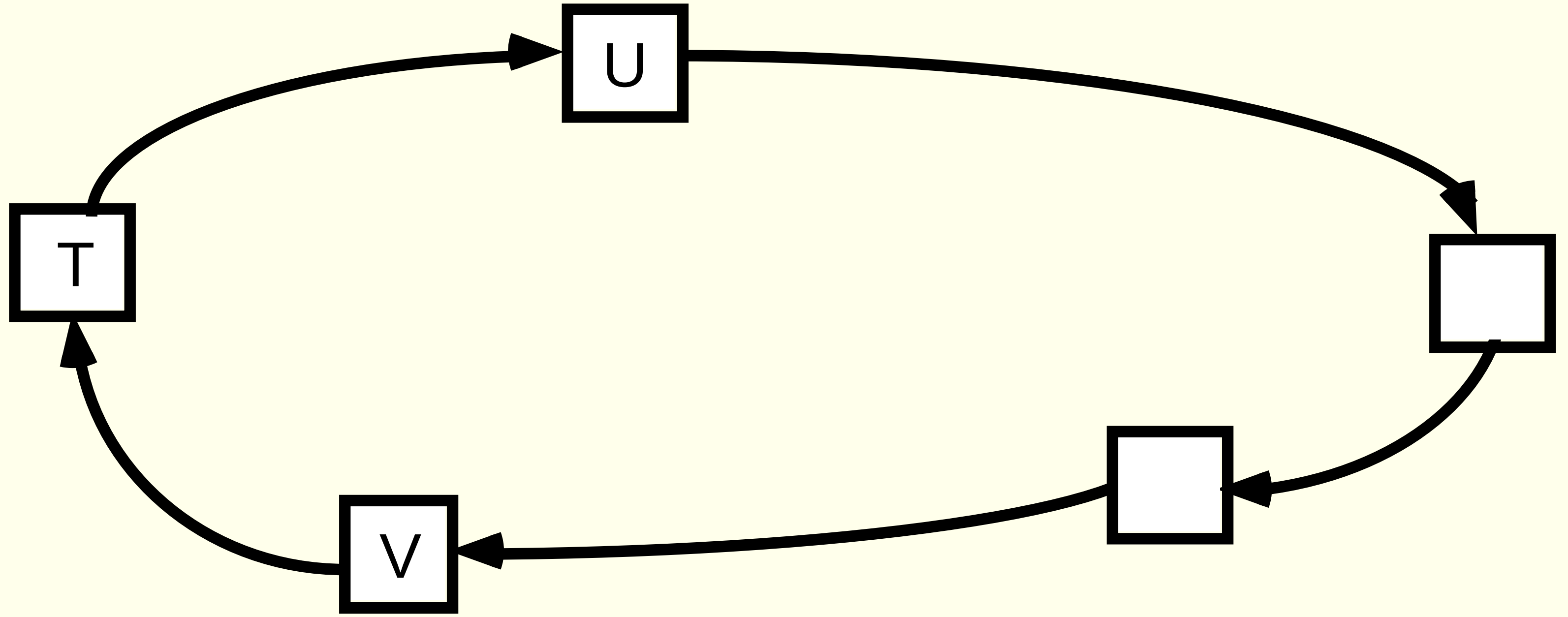
Dead lock with write locks

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*		
• • •	lock on *B*	*a.withdraw(200);*	waits for *T's*
		• • •	lock on *A*
• • •		• • •	
• • •		• • •	

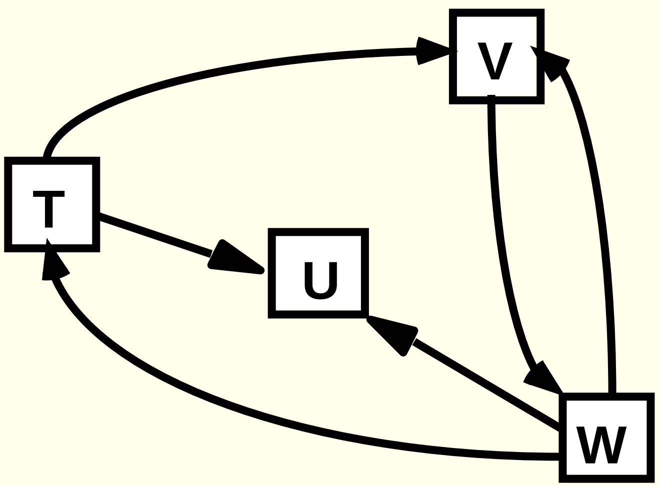
The wait-for graph

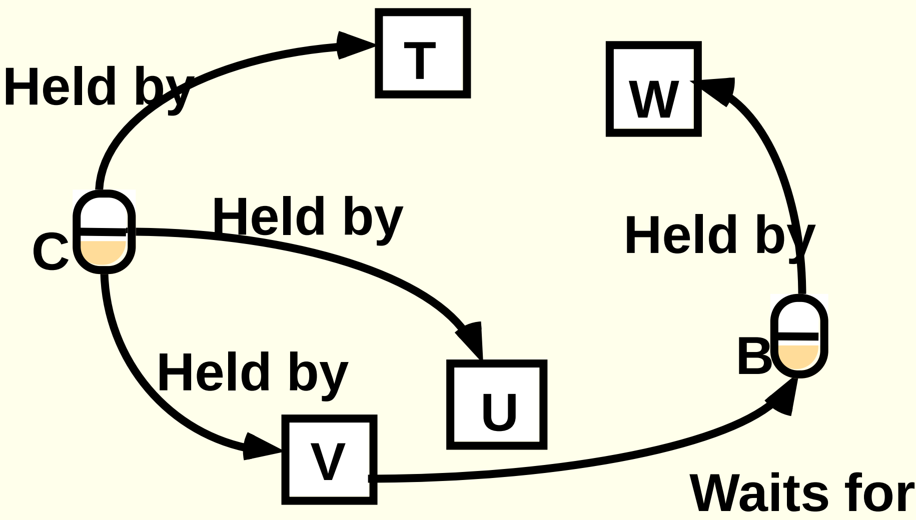


A cycle in wait-for graph



Another wait-for graph





T, U and V share a read lock on object C

W holds a write lock on object B

Dead lock happens when T and W request to hold write lock on object C

Resolution of the deadlock

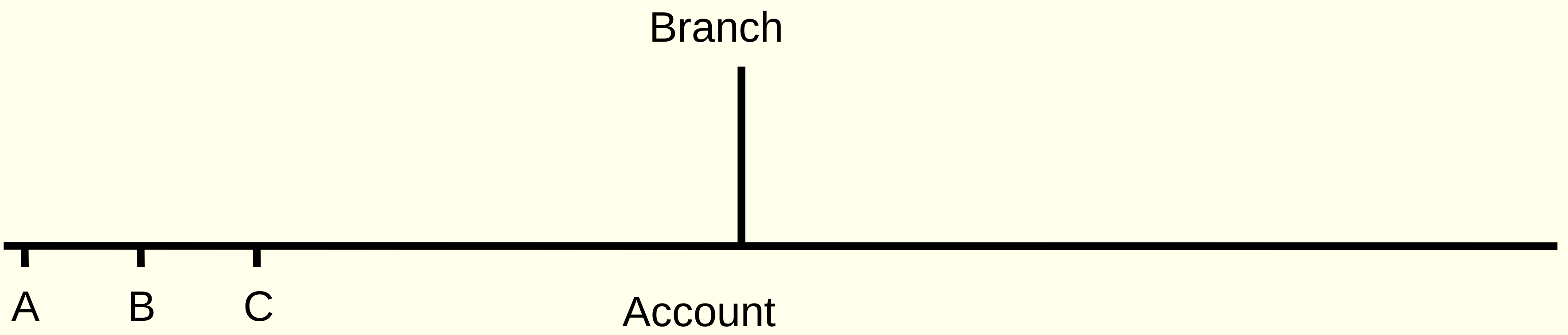
Transaction T		Transaction U	
Operations	Locks	Operations	Locks
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock B
b.withdraw(100) •••	waits for U's lock on B	*a.withdraw(200);* •••	waits for T's lock on A
	(timeout elapses)	•••	
T's lock on becomes vulnerable, unlock A , abort T			
		a.withdraw(200);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

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

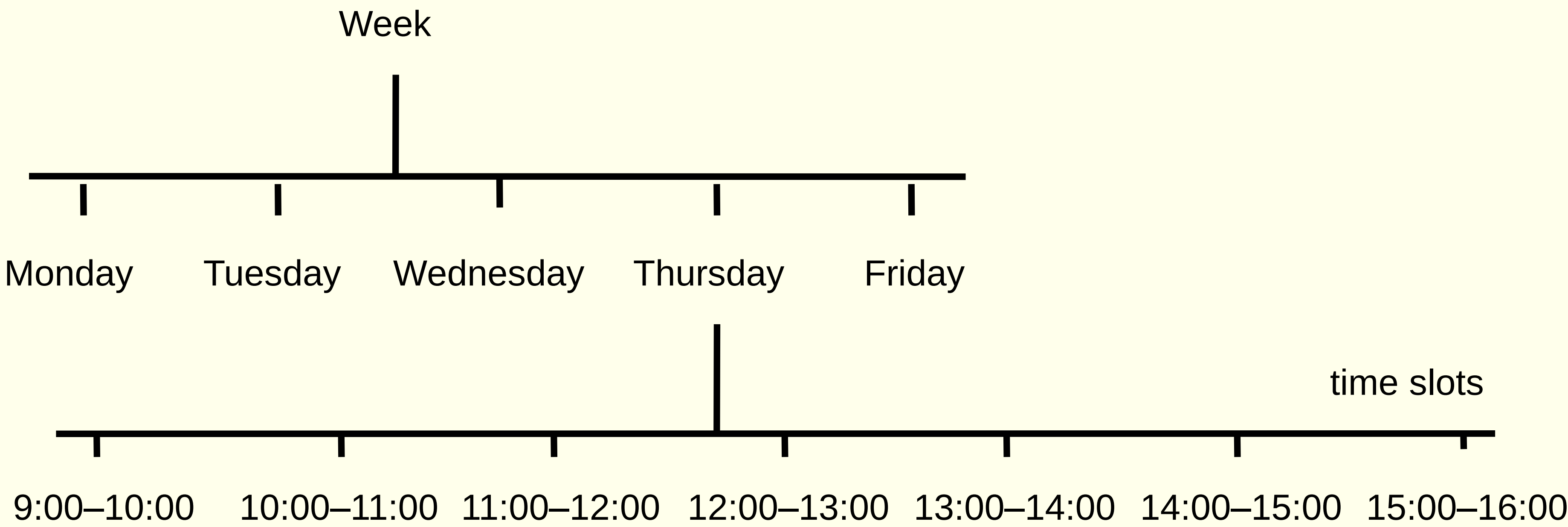
For one object		*Lock to be set*		
		read	*write*	*commit*
Lock already set	*none*	OK	OK	OK
	read	OK	OK	wait
	write	OK	wait	
	commit	wait	wait	

To tentative version

Lock hierarchy for the banking example

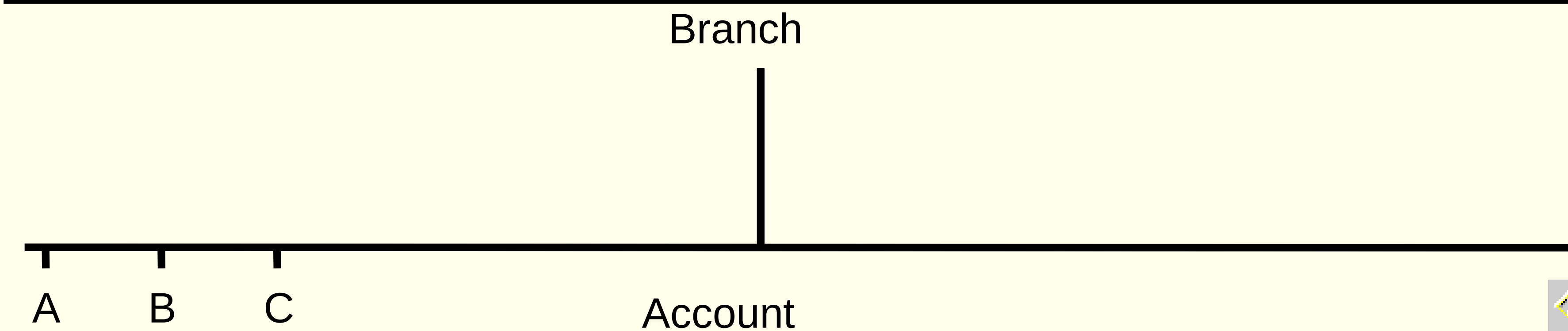


Lock hierarchy for a diary



Lock compatibility table for hierarchic locks

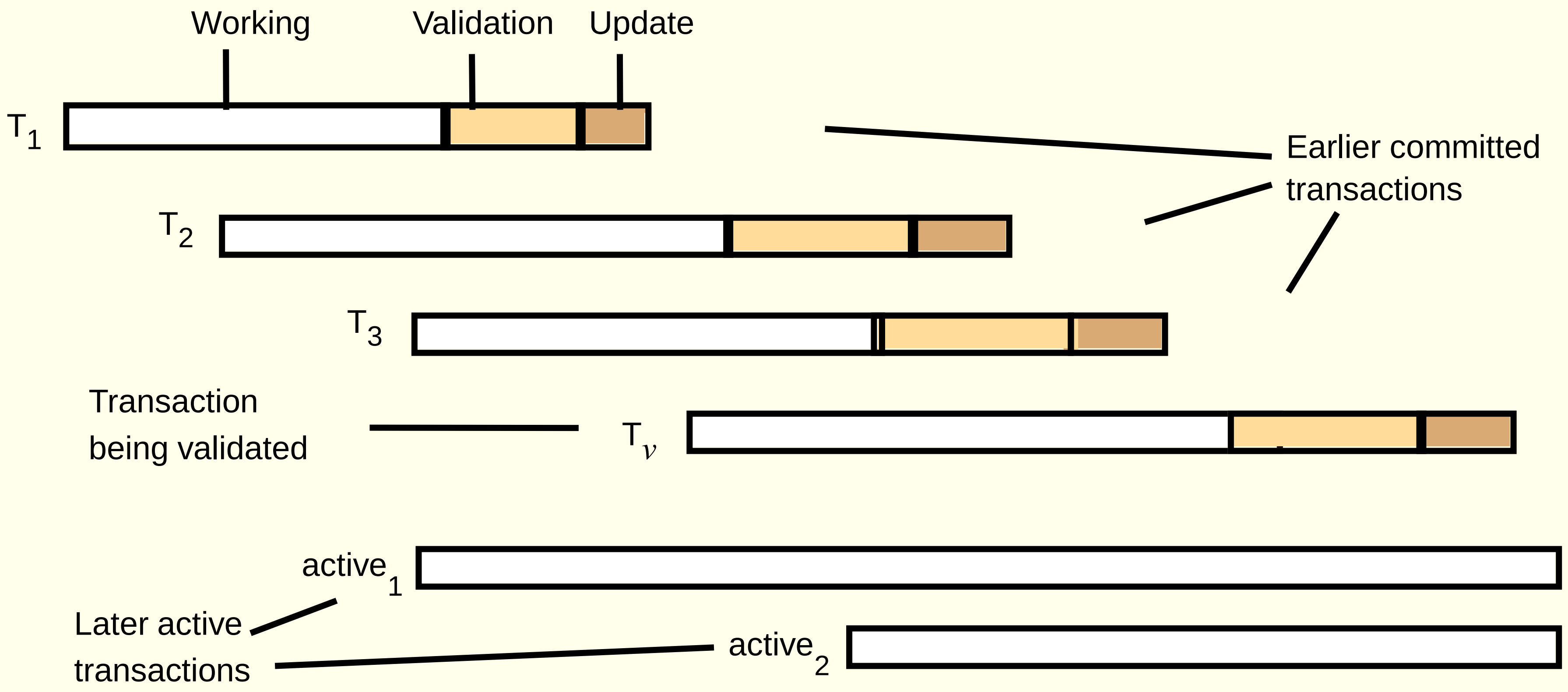
For one object		*Lock to be set*			
		read	*write*	*I-read*	*I-write*
Lock already set	*none*	OK	OK	OK	OK
	read	OK	wait	OK	wait
	write	wait	wait	wait	wait
	I-read	OK	wait	OK	OK
	I-write	wait	wait	OK	OK



Serializability of transaction T_v with respect to transaction $T_i (i<v)$

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

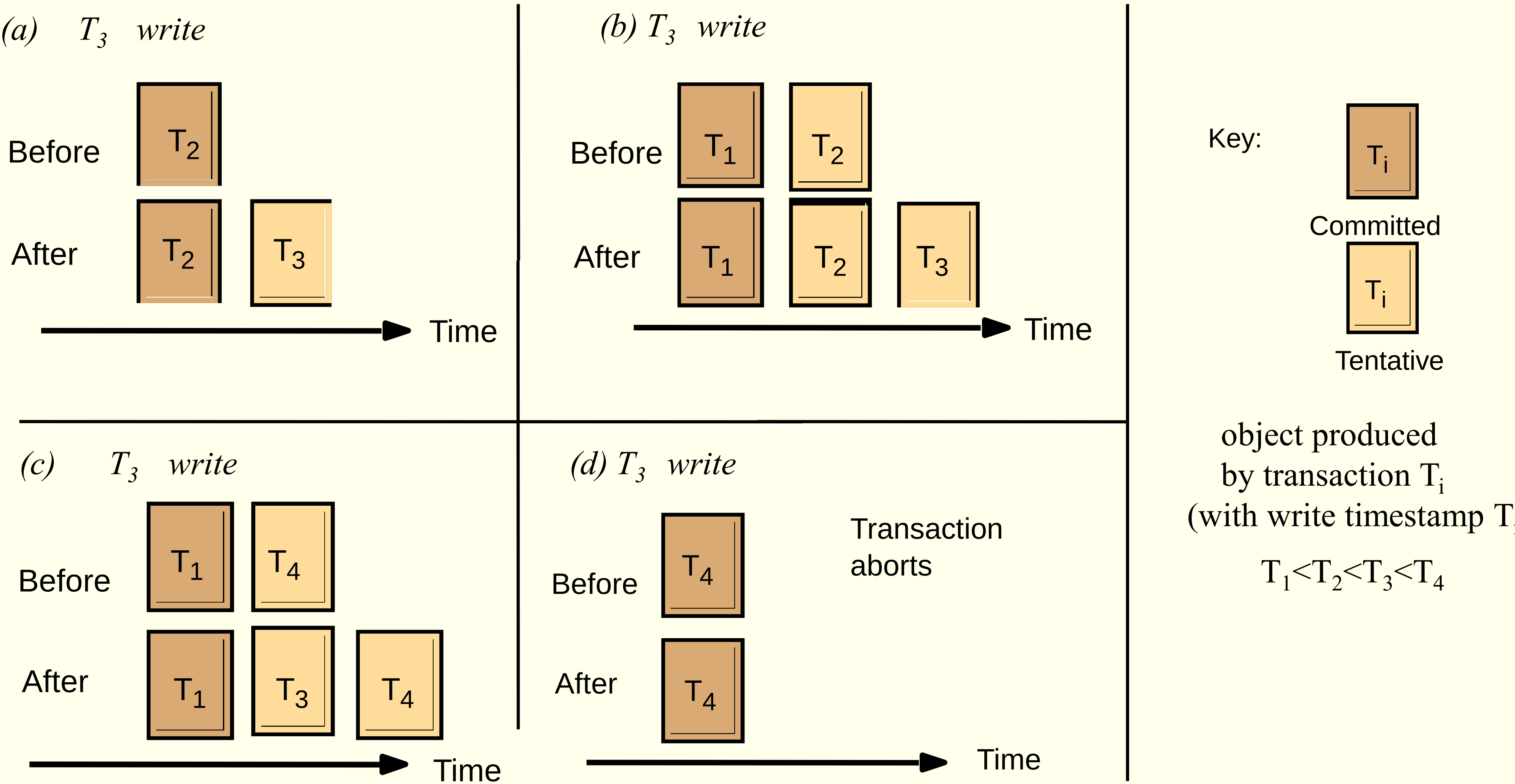
Validation of transactions



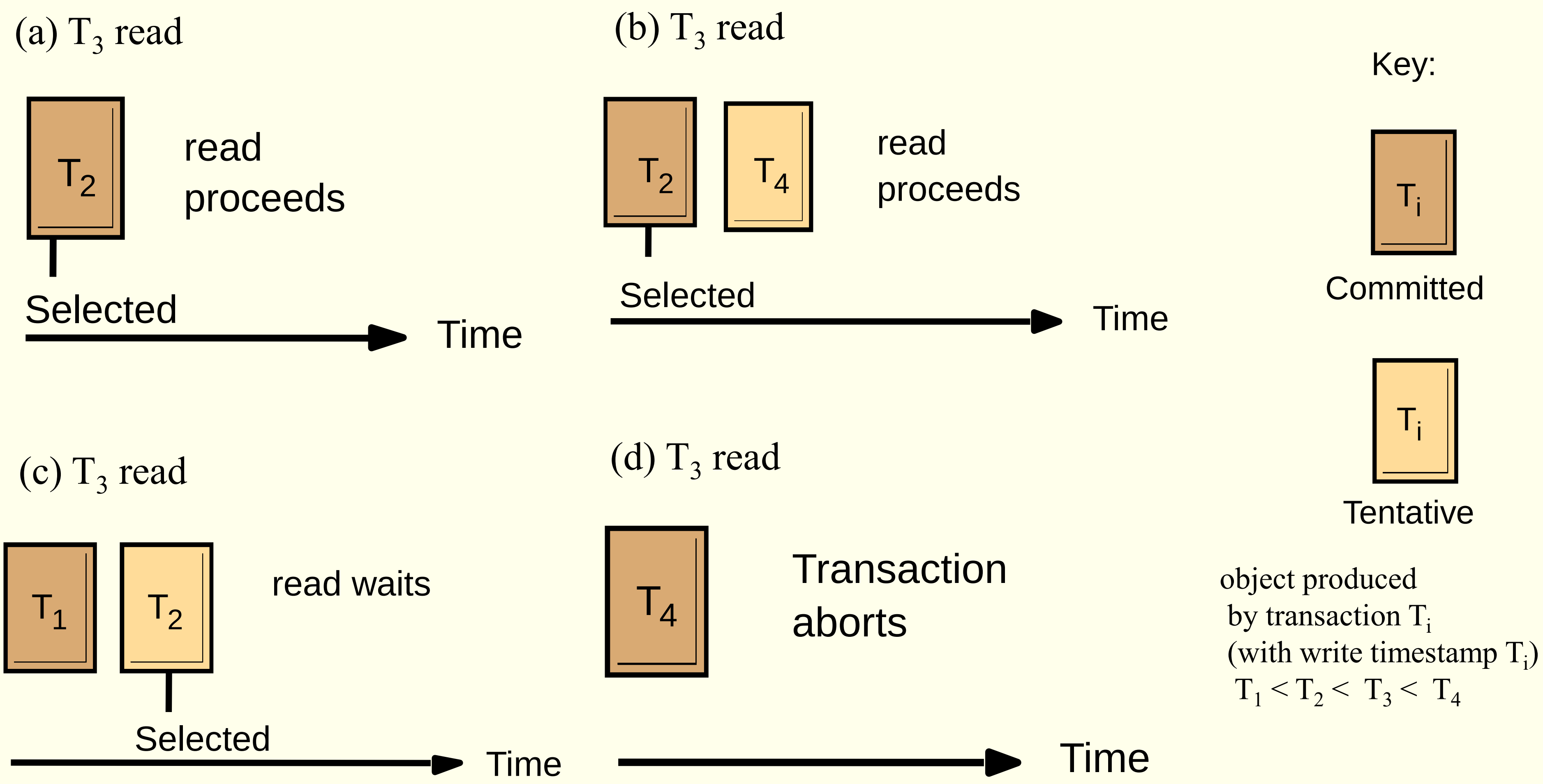
Operation conflicts for timestamp ordering

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 $T_c >$ the 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.

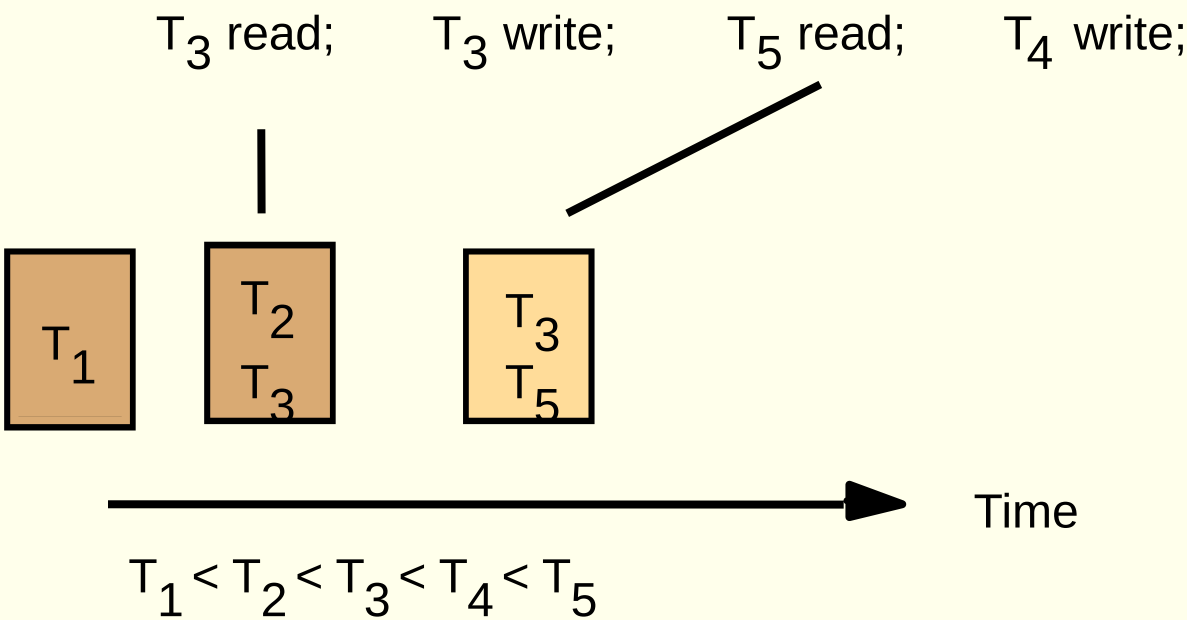
Write operations and timestamps



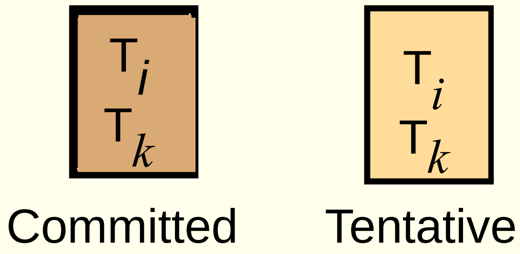
Read operations and timestamps



Late *write* operation would invalidate a *read*



Key:



object produced by transaction T_i (with write timestamp T_i and read timestamp T_k)