Transactions and concurrency control

based on Distributed Systems: Concepts and Design, Edition 5

Ali Fanian

Isfahan University of Technology
www.Fanian.iut.ac.ir

Outline

- What are transactions?
- Concurrency control
- Recoverability from aborts
- Locks
- Optimistic concurrency control
- Timestamp ordering

Transactions

In some situations, clients require a sequence of separate requests to a server to be atomic in the sense that:

- They are free from interference by operations being performed on behalf of other concurrent clients.
- Either all of the operations must be completed successfully or they must have no effect at all in the presence of server crashes.

Transactions

- Example

Transaction T:

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

ACID properties

- Atomicity: a transaction must be all or nothing.
- Consistency: a transaction takes the system from one consistent state to another consistent state.
- Isolation: each transaction must be performed without interference from other transactions.
- Durability: after a transaction has completed successfully, all its effects are saved in permanent storage.

Concurrency control

- Problems of concurrent transactions:
  - **The lost update problem**
  - **Inconsistent retrievals problem**
- We assume throughout that each of the operations *deposit*, *withdraw*, *getBalance* and *setBalance are atomic.*

The lost update problem

Initial balances: A=100$; B=200$; C=300$

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)

T	U	Balance A	Balance B	Balance C
balance =b.getBalance(); 200$		100	200	300
	balance=b.getBalance(); $200	100	200	300
	b.setBalance(balance*1.1);	100	220	300
b.setBalance(balance*1.1);		100	220	300
a.withdraw(balance/10)		80	220	300
	c.withdraw(balance/10)	80	220	280

Inconsistent retrievals problem

Initial balances: A=200$; B=200$

Transaction V	Transaction W
a.withdraw(100)	aBranch.branchTotal()
b.deposit(100)	

Transaction V	Transaction W	Balance A	Balance B
a.withdraw(100);		100	200
	total = a.getBalance( ) $100	100	200
	total += b.getBalance() $300	100	200
b.deposit(100)		100	300

Serial equivalence

- If we have a set of transactions and we don't't have any particular order on them then we could say a correct result is some sequence of them.
- For example consider the set of transaction S, T,U then we could take any order:

S;T;U, S;U;T, T;S;U, T;U;S, U;S;T, U;T;S.

Serial equivalence (continue)

- An interleaving of the operations of transactions in which the combined effect is the same as if the transactions had been performed one at a time in some order is a *serially equivalent* interleaving.
- The goal of concurrency control is to ensure serial equivalence while trying to be as efficient as possible.

Serial equivalence (continue)

- The lost update problem occurs when two transactions read the old value of a variable and then use it to calculate the new value.
- As a serially equivalent interleaving of two transactions produces the same effect as a serial one, we can solve the lost update problem by means of serial equivalence.

Transaction T	Transaction U
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

Serial equivalence (continue)

- The inconsistent retrievals problem can occur when a retrieval transaction runs concurrently with an update transaction.
- It cannot occur if the retrieval transaction is performed before or after the update transaction.

Transaction V	Transaction W
a.withdraw(100); $100	
b.deposit(100) $300	
	total = a.getBalance( ) $100
	total += b.getBalance() $400

Conflicting operations

- When we say that a pair of operations *conflicts* we mean that their combined effect depends on the order in which they are executed.

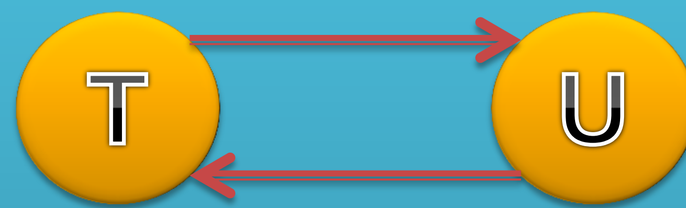
The conflict rules for *read* and *write* operations:

Operations of different transactions		Conflict
Read	Read	No
Read	Write	Yes
Write	Write	Yes

Serial equivalence definition in terms of operation conflicts

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

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



Recoverability from aborts

- This section illustrates two problems associated with aborting transactions:
  - dirty reads
  - Premature writes
- Both of this problems can occur in the presence of serially equivalent executions of transactions.

Dirty reads

- The isolation property of transactions requires that transactions do not see the uncommitted state of other transactions.
- This problem is caused by the interaction between a *read* operation in one transaction and an earlier *write* operation in another transaction on the same object.

Dirty read example

Transaction T	Transaction U
a.getBalance()	a.getBalance()
a.setBalance(balance + 10)	a.setBalance(balance + 20)

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

Recoverability of transactions

- If a transaction has committed after it has seen the effects of a transaction that subsequently aborted, the situation is not recoverable.
- To ensure that such situations will not arise, any transaction that is in danger of having a dirty read delays its commit operation.

Cascading aborts

Transaction T	Transaction U
a.getBalance() $100	
a.setBalance(balance + 10) $110	
	a.getBalance() $110
	a.setBalance(balance + 20) $130
abort transaction	
	abort transaction

- To avoid cascading aborts, transactions are only allowed to read objects that were written by committed transactions.

Premature writes

Transaction T	Transaction U
a.setBalance(105)	a.setBalance(110)

Transaction T	Transaction U
a.setBalance(105) $105	
	a.setBalance(110) $110

To ensure correct results in a recovery scheme that uses before images, *write* operations must be delayed until earlier transactions that updated the same objects have either committed or aborted.

Recoverability from aborts

Strict executions of transactions

- The executions of transactions are called strict if the service delays both read and write operations on an object until all transactions that previously wrote that object have either committed or aborted.
- Enforces isolation

Tentative versions

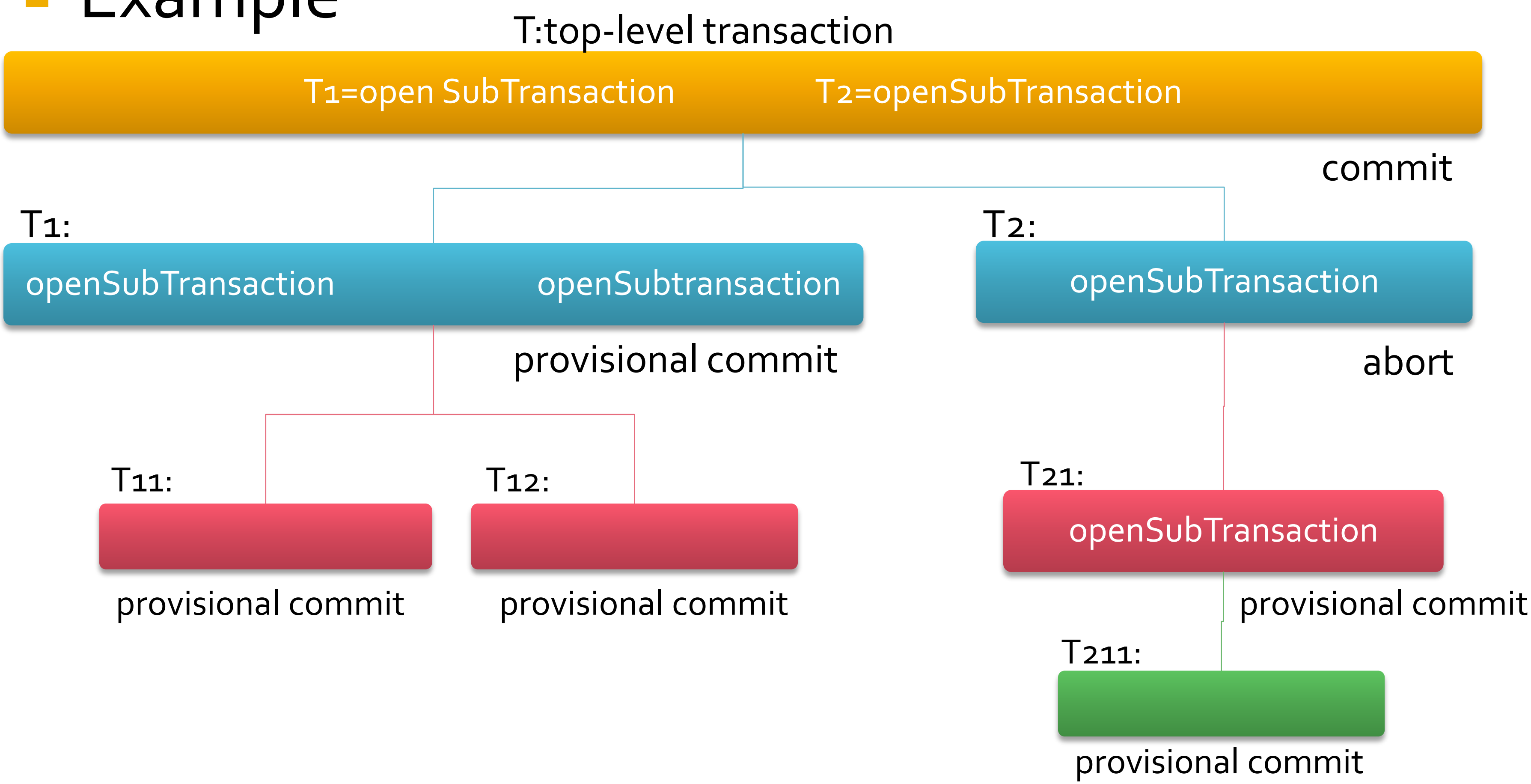
- All of the update operations performed during a transaction are done in tentative versions of objects in volatile memory.
- The tentative versions are transferred to the objects only when a transaction commits.

Nested transactions

- Nested transactions extend the transaction model by allowing transactions to be composed of other transactions.
- The outermost transaction in a set of nested transactions is called the *top-level* transaction.
- Transactions other than the top-level transaction are called *subtransactions*.

Nested transactions

- Example



Nested transactions

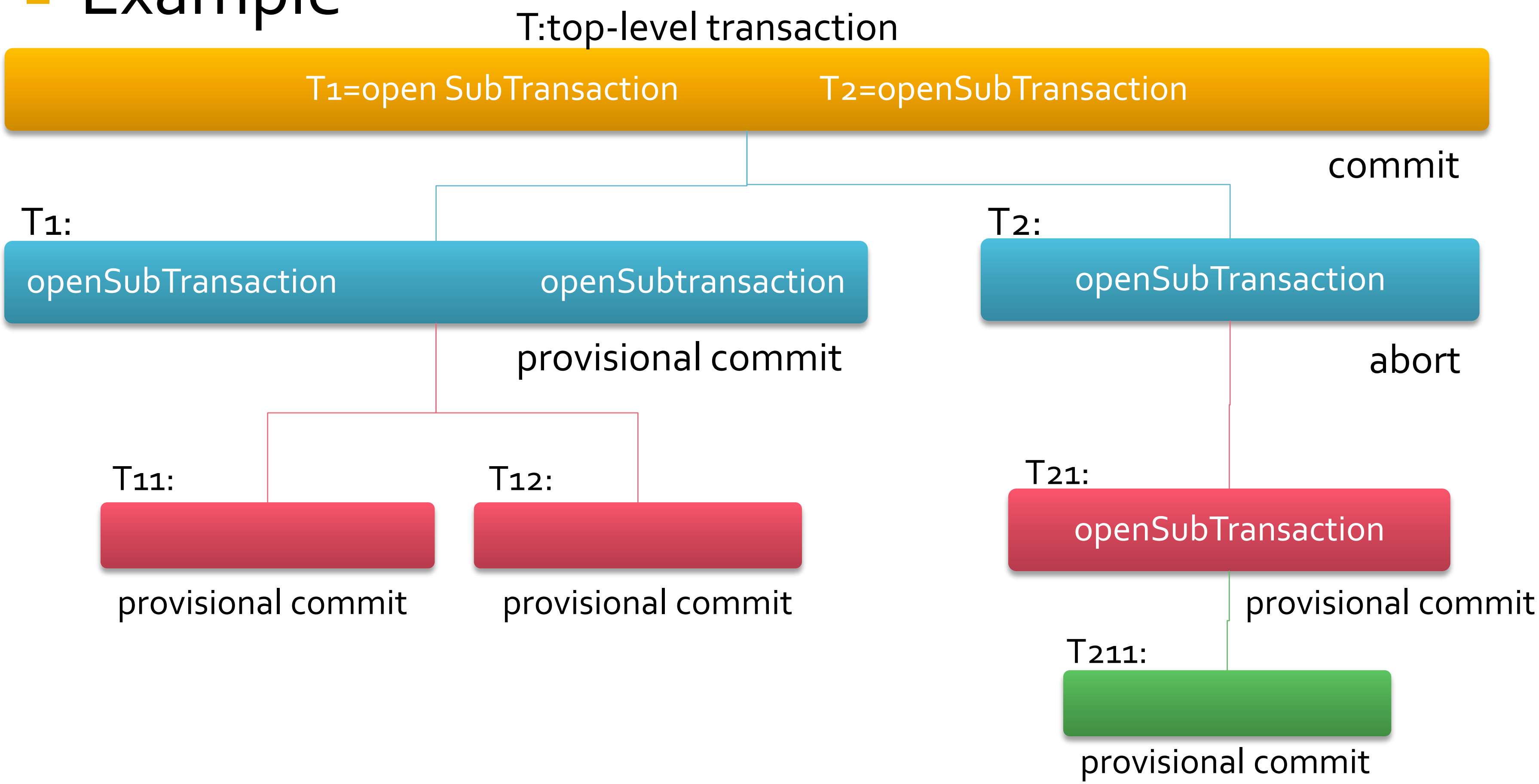
- Main advantages:
  - Additional concurrency in a transaction.
  - Subtransactions can commit or abort independently.

Nested transactions

- The rules for committing of nested transactions:
  - A transaction may commit or abort only after its child transactions have completed.
  - When a subtransaction completes, it makes an independent decision either 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.

Nested transactions

- Example



Locks

- Example

Transaction T		Transaction U	
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

Locks

- Two-phase locking (2PL)
  - All pairs of conflicting operations of two transactions should be executed in the same order.
  - To ensure this, a transaction is not allowed any new locks after it has released a lock.
  - The first phase of each transaction is a ‘growing phase’, and the second phase is a ‘shrinking phase’.
  - This is called two-phase locking.
- Strict two-phase locking (S2PL)
  - Any locks acquired are not given back until the transaction completed or aborts (ensures recoverability).
  - the locks must be held until all the objects it updated have been written to permanent storage.

Locks

- Two types of locks are used:
  - *Read locks .*

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

Locks

- Solving inconsistent retrieval problem
  - Inconsistent retrievals are prevented by performing the retrieval transaction before or after the update transaction.
- Solving lost update problem
  - Lost updates occur when two transactions read a value of 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.

Strict two-phase locking rules

1. When an operation accesses an object within a transaction:

If the object is not locked	locked	Operation proceeds
If the object has a conflicting lock	wait	
If the object has a non-conflicting lock	Share the lock	Operation proceeds
If the object has already been locked in the same transaction	Promote it if necessary	Operation proceeds

2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.

Locks

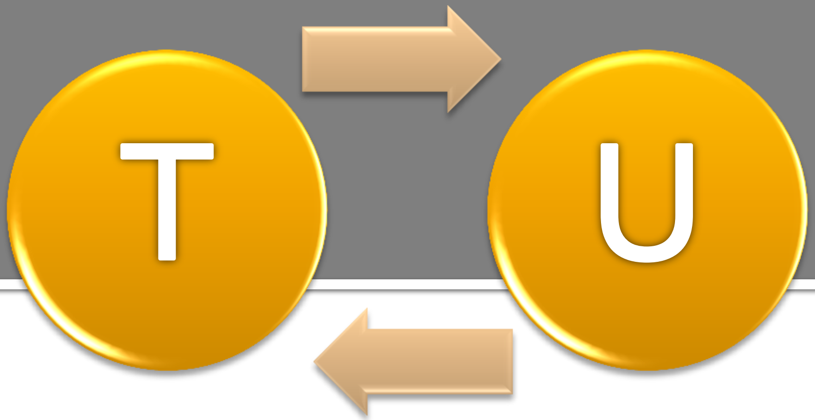
- Lock implementation
  - The granting of locks will be implemented by a separate object in the server that we call the *lock manager*.
  - The lock manager holds a set of locks, for example in a hash table.
  - Each lock is an instance of the class *Lock* and is associated with a particular object.

```
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 conflicting 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();
        }
    }

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

locks



- Deadlocks

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*
•••		•••	
•••		•••	

Locks

- **Deadlock prevention**
  - Lock all of the objects used by a transaction when it starts.
  - This would need to be done as a single atomic step.
- Disadvantages
  - Unnecessarily restricts access to shared resources.
  - It is sometimes impossible to predict at the start of a transaction which objects will be used.

Locks

- **Deadlock detection**
  - Deadlocks may be detected by finding cycles in the wait-for graph.
  - Having detected a deadlock, a transaction must be selected for abortion to break the cycle.

Locks

- **Timeouts**

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*
	(timeout elapses)	•••	
T's lock on *A* becomes vulnerable,			
	unlock *A*, abort T		
		a.withdraw(200);	write lock *A*
			unlock *A*, *B*

Optimistic concurrency control

- Drawbacks of locking:
  - Lock maintenance represents an overhead that is not present in systems
  - The use of locks can result in deadlock.
  - To avoid cascading aborts, locks cannot be released until the end of the transaction. This may reduce significantly the potential for concurrency.

Optimistic concurrency control

Each transaction has the following phases:

- Working phase
- Validation phase
- Update phase

Optimistic concurrency control

- Working phase:
  - Each transaction has a tentative version of each of the objects that it updates.
  - The use of tentative versions allows the transaction to abort with no effect on the objects.
  - *Read* operations are performed immediately.

Optimistic concurrency control

- Working phase:
  - *Write* operations record the new values of the objects as tentative values.
  - When there are several concurrent transactions, several different tentative values of the same object may coexist.
  - All *read* operations are performed on committed versions of the objects (or copies of them), dirty reads cannot occur.

Optimistic concurrency control

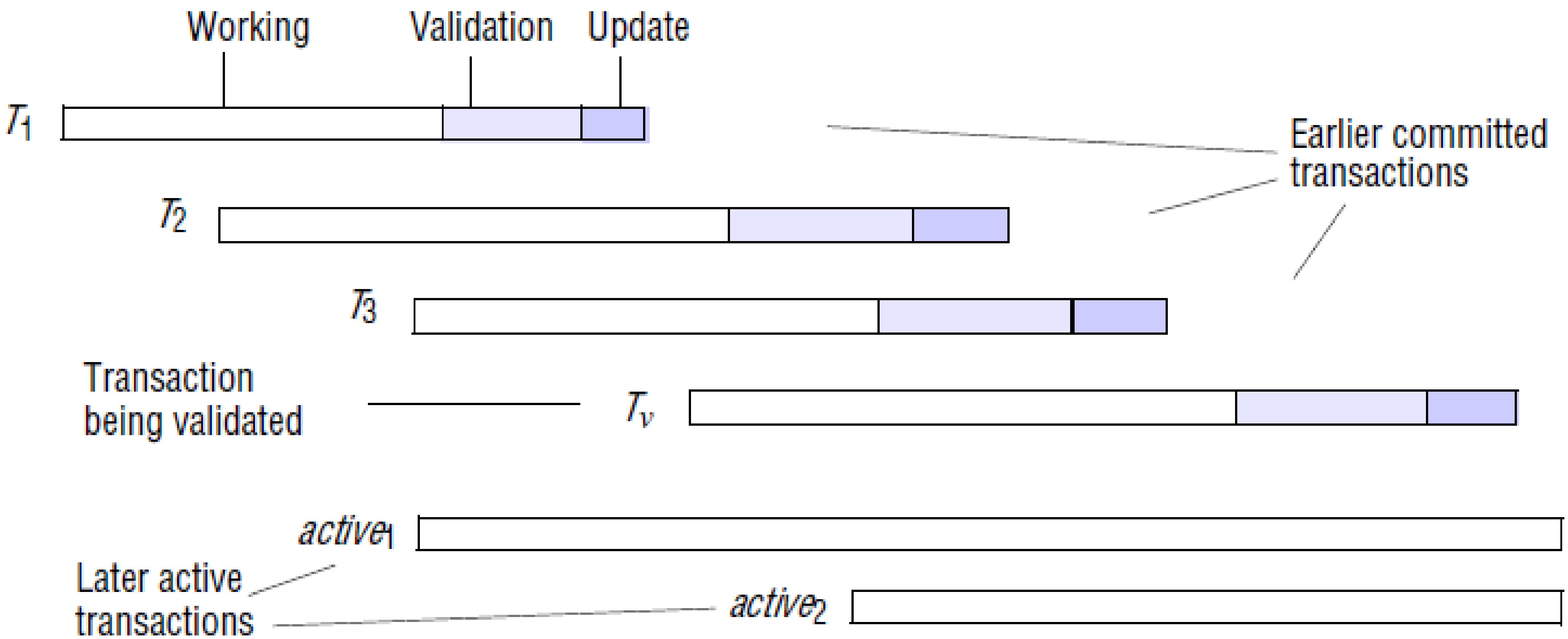
- Validation phase:
  - When the *closeTransaction* request is received, this phase begins.
  - If the validation is successful, then the transaction can commit.
  - If the validation fails, then some form of conflict resolution must be used.

Optimistic concurrency control

- Update phase:
  - If a transaction is validated, all of the changes recorded in its tentative versions are made permanent.
  - Read-only transactions can commit immediately after passing validation.
  - Write transactions are ready to commit once the tentative versions of the objects have been recorded in permanent storage.

Optimistic concurrency control

- Overlapping transactions



Optimistic concurrency control

- Validation of transactions
  - Validation uses the read-write conflict rules to ensure that the scheduling of a particular transaction is serially equivalent with respect to all other *overlapping* transactions.
  - Each transaction is assigned a transaction number when it enters the validation phase.

Optimistic concurrency control

- Validation of transactions

The validation test on transaction *Tv* is based on conflicts between operations in pairs of transactions *Ti* and *Tv*. For a transaction *Tv* to be serializable with respect to an overlapping transaction *Ti*, their operations must conform to the following rules:

Tv	Ti	Rule
write	*read*	1. *Ti* must not read objects written by *Tv*.
read	*write*	2. *Tv* must not read objects written by *Ti*.
write	*write*	3. *Ti* must not write objects written by *Tv* and *Tv* must not write objects written by *Ti*.

Note that this restriction on *write* operations, together with the fact that no dirty reads can occur, produces strict executions.

Optimistic concurrency control

- Backward validation
  - Rule 1 is satisfied, why?

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

Optimistic concurrency control

- Forward validation
  - Rule 2 is satisfied, why?

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

Optimistic concurrency control

- Forward validation
  - Resolving conflicts:
    - Abort all the conflicting active transactions and commit the transaction being validated.
    - Abort the transaction being validated.

Optimistic concurrency control

- Comparison of forward and backward validation
  - Forward validation allows flexibility.
  - Backward validation compares a possibly large read set against the old write sets
  - Backward validation has the overhead of storing old write sets.
- Starvation