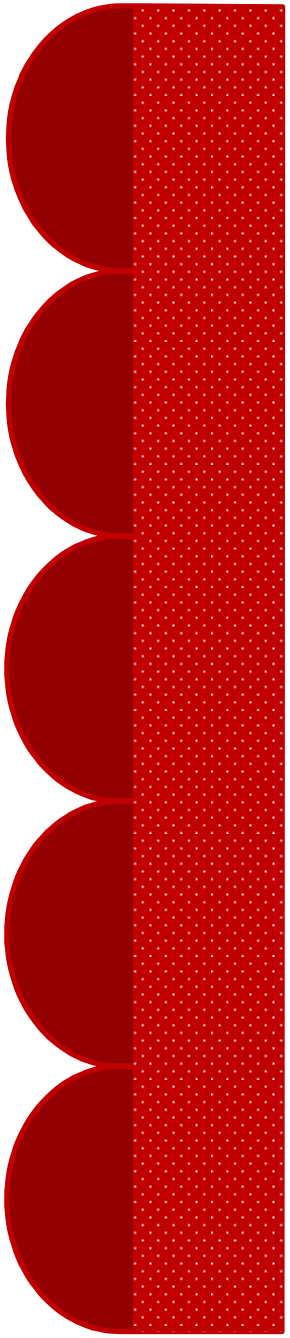
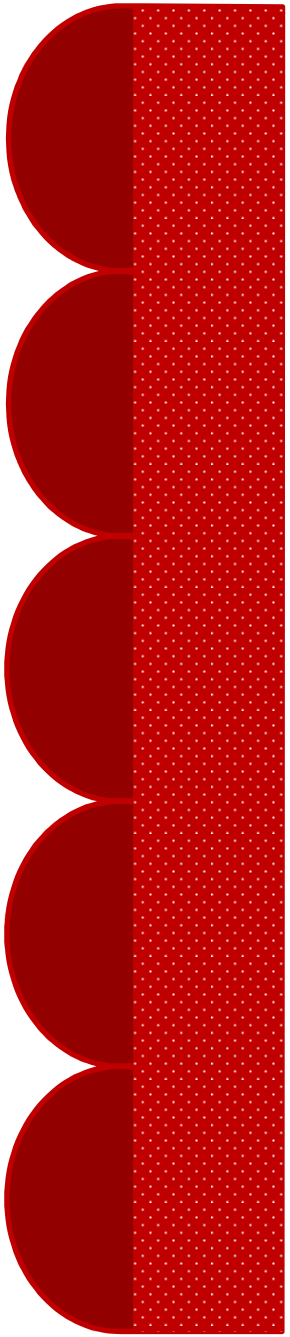
Optimistic Concurrency Control & Timestamp Ordering

George Coulouris, Jean Dollimore and Tim Kindberg, "Distributed Systems Concepts and Design", Fifth Edition, Pearson Education, 2012



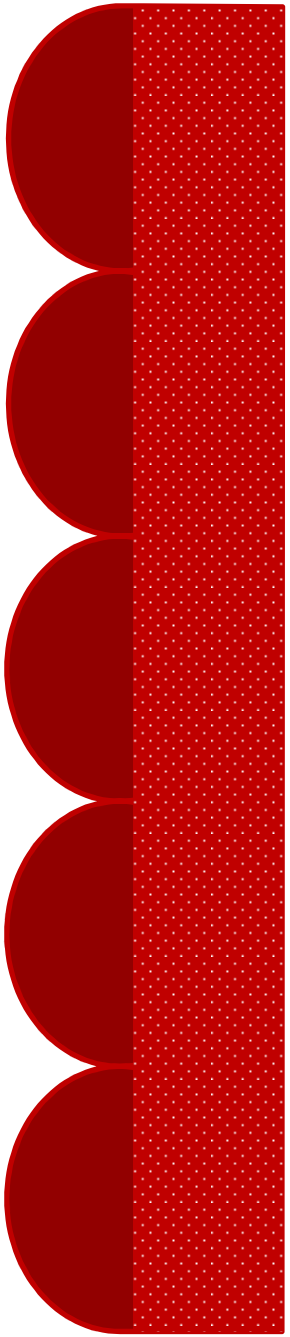
Overview

- Optimistic Concurrency Control
  - Backward Validation
  - Forward Validation
- Timestamp Ordering



Problems with Locks

- **Lock** maintenance represents an **overhead**.
- Use of locks can result in **deadlock**
- To avoid **cascading aborts**, **locks cannot** be **released until** the **end** of the **transaction**.



Optimistic Approach

Optimistic Concurrency Control

- Transactions are **allowed to proceed** as though there were **no possibility of conflict** with other transactions **until** the client **completes** its task and issues a **closeTransaction** request.

- When a **conflict arises**, some transaction is generally **aborted** and will need to be **restarted** by the client.

Optimistic Concurrency Control

- **Working phase:**
  - During the working phase, each transaction has a **tentative** version of each of the **objects** that it updates.
  - **Write** operation happens on **Tentative** versions of object
  - **Read** operation happens from **committed** version of object.
  - Every **transaction** will have a set of **read set** and **write set** of **objects**.
- **Validation Phase**
  - When the **closeTransaction** request is received, the transaction is **validated** to establish **whether** or **not** its **operations** on **objects conflict** with **operations** of **other transactions** on the **same objects**. Commit- Validation Success, conflict occurs- Validation Fails.
- **Update Phase**
  - If a is validated, all of the **changes** recorded in its **tentative versions** are made **permanent**

Optimistic Concurrency Control

- **Validation of Transactions:**
- Each **transaction** is **assigned** a **transaction number** when it **enters** the **validation** phase (that is, when the client issues a closeTransaction).
- If the transaction is **validated** and **completes** successfully, it **retains** this number;
- If it **fails** the validation checks and is **aborted**, or if the transaction is **read only**, the number is **released** for **reassignment**.
- Transaction numbers are **integers** assigned in **ascending sequence**

Optimistic Concurrency Control

Validation of Transactions:

- Consider two transactions: Tv and Ti.
- Tv is to be validated

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.

Optimistic Concurrency Control

- **Validation of Transactions:**

Only one transaction may be in the validation and update phase at one time.

- To prevent overlapping, the entire validation and update phases can be implemented as a critical section
- Assignment of transaction numbers is performed sequentially.
- A transaction number is like a pseudo-clock that ticks whenever a transaction completes successfully

Types of Validations

- **Backward validation** checks the **transaction** undergoing **validation** with other **preceding overlapping transactions** – those that **entered** the **validation** phase **before** it.

- **Forward validation** checks the **transaction** undergoing **validation** with other **later transactions**, which are **still active**.

Types of Validations

- **Backward validation** As **all** the **read operations** of **earlier** overlapping **transactions** were performed before the **validation** of **Tv started**, they cannot be affected by the **writes** of the **current transaction**

Tv checks whether its **read set** (the objects affected by the read operations of Tv) **overlaps** with any of the **write sets** of **earlier** overlapping **transactions**, **Ti**

Types of Validations

- **Forward validation**
- Forward validation of the transaction Tv, the **write set** of **Tv** is **compared** with the **read sets** of **all** overlapping **active transactions**

Validation

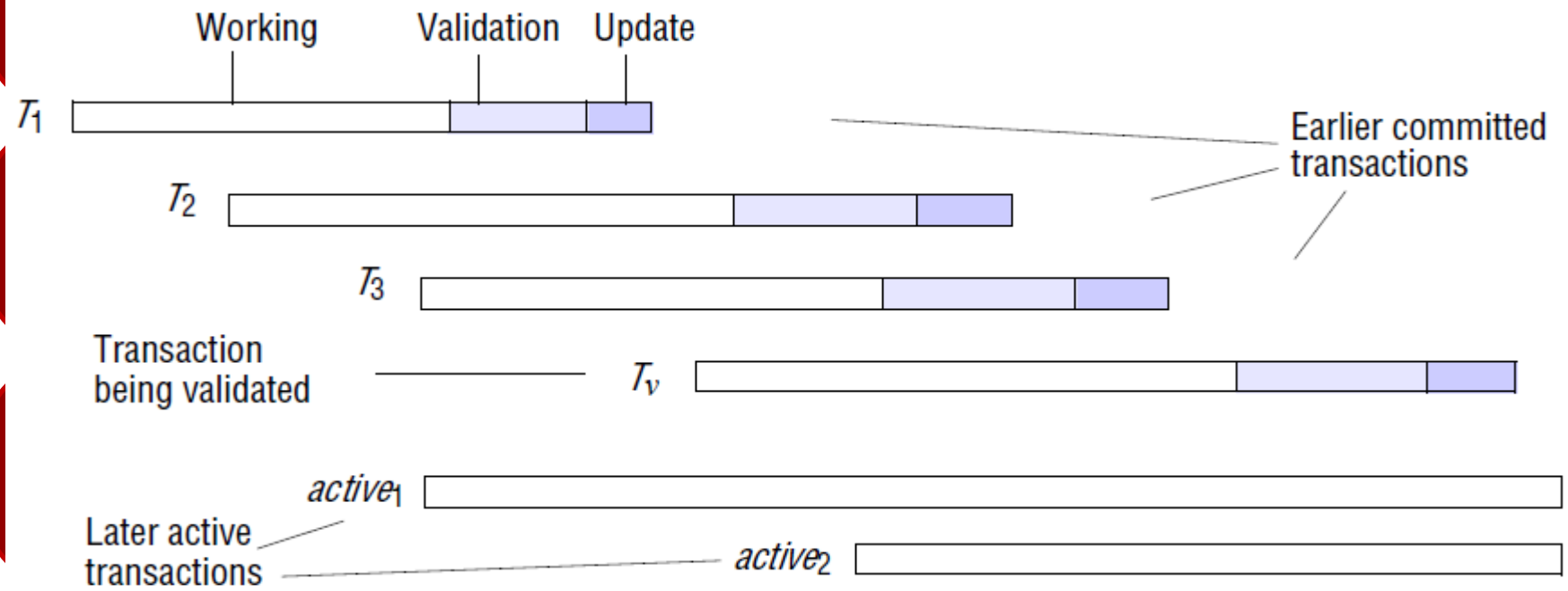
- **startTn** be the **biggest transaction number** assigned (to some other committed transaction) at the **time** when **transaction Tv started** its **working** phase

- **finishTn** be the **biggest transaction number** assigned at the **time** when **Tv entered** the **validation** phase

Backward Validation

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;
}
```



Forward Validation

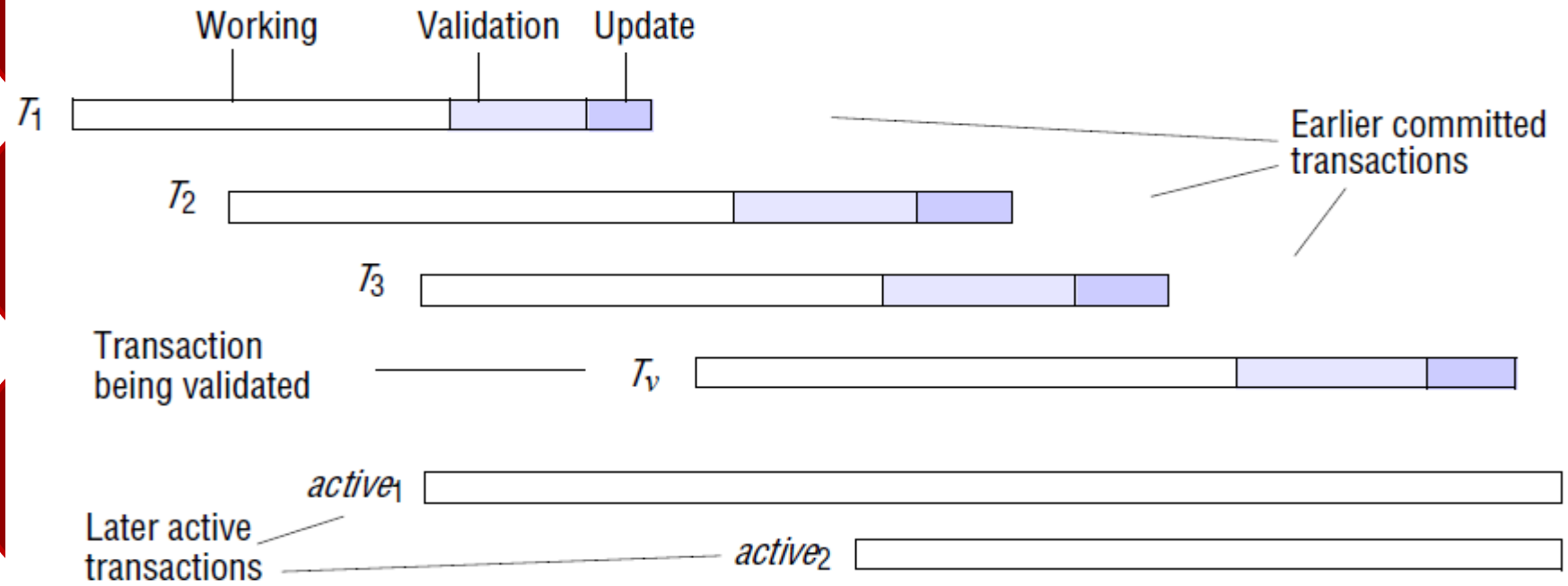
Forward validation of transaction T_v

boolean valid = true;

for (int T_{id} = active1; T_{id} <= activeN; T_{id}++){

if (write set of T_v intersects read set of T_{id}) valid = false;

}

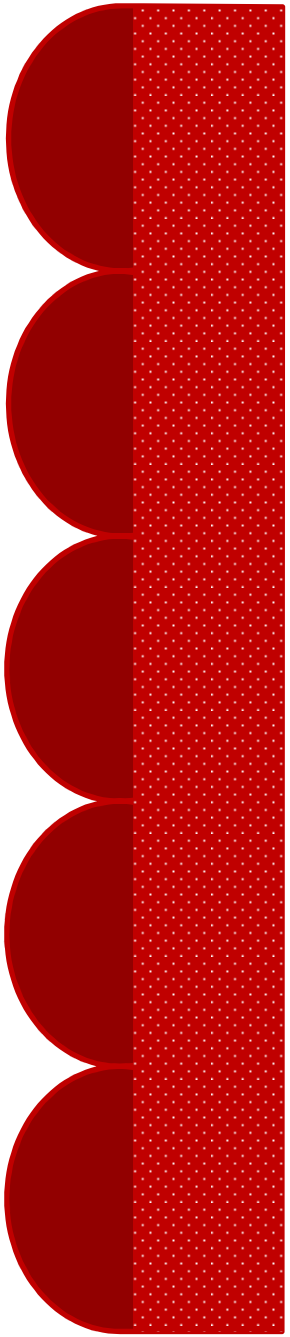


Alternatives of Forward validation

- **Defer** the **validation until** a **later** time when the **conflicting transactions** have **finished**.
- **Abort all** the **conflicting active transactions** and **commit** the **transaction** being **Validated**
- **Abort** the **transaction** being **validated**.
- **Future conflicting transactions** may be going to **abort**, the **transaction** under **validation** has **aborted unnecessarily**.
- **Validation may lead to starvation.**
- **Set maximum limit on number of times a transaction can be aborted.**
- **Server must keep track of number of times a transaction is aborted due to validation. Allow transaction to proceed, if it exceeds max limit.**

Forward vs. Backward Validation

Sno	Forward Validation	Backward Validation
1	Allows flexibility in the resolution of conflicts	Allows only one choice - To abort the transaction being validated
2	Checks a small write set against the read sets of active transactions	Compares a possibly large read set against the old write sets
3	Allow for new transactions starting during the validation process.	Overhead of storing old write sets until they are no longer needed.



Timestamp Ordering

Timestamp Ordering

- Each operation in a transaction is validated when it is carried out.
- If the operation cannot be validated, the transaction is aborted immediately and can then be restarted by the client.
- **Write: A transaction's request to write an object is valid only if that object was last read and written by earlier transactions.**
- **Read: A transaction's request to read an object is valid only if that object was last written by an earlier transaction**

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 $T_c >$ the write timestamp of the committed object.

Write Operation

if ($T_c \geq$ **maximum read timestamp** on D &&

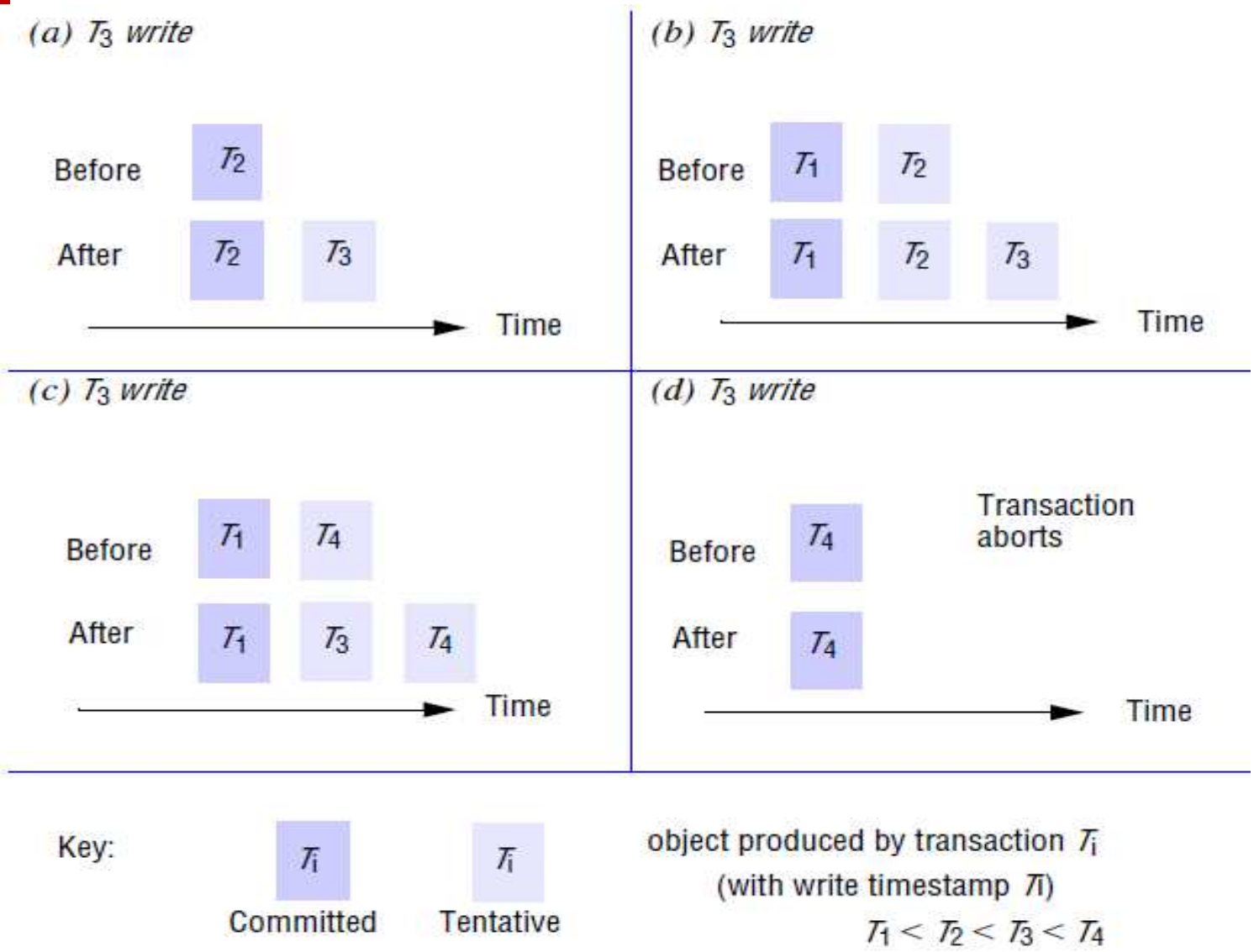
$T_c >$ **write timestamp** on committed version of D)

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

else /* write is too late */

Abort transaction T_c

Write Operation



Read Operation

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 ($\mathbf{D_{selected}}$ is **committed**)

perform ***read*** operation on the version D_{selected}

else

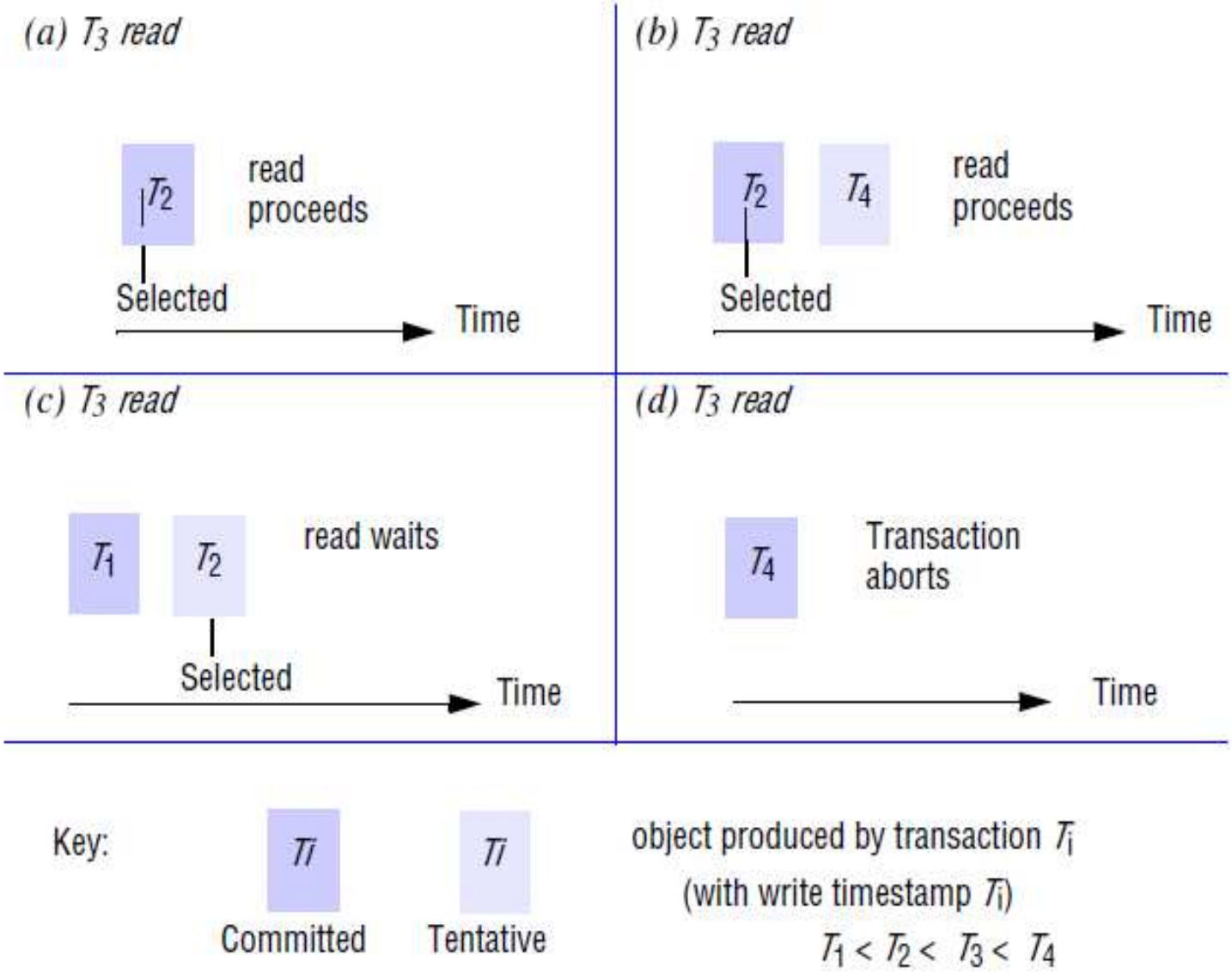
Wait **until** the **transaction** that made version D_{selected} **commits** or **aborts**

then **reapply** the ***read*** rule

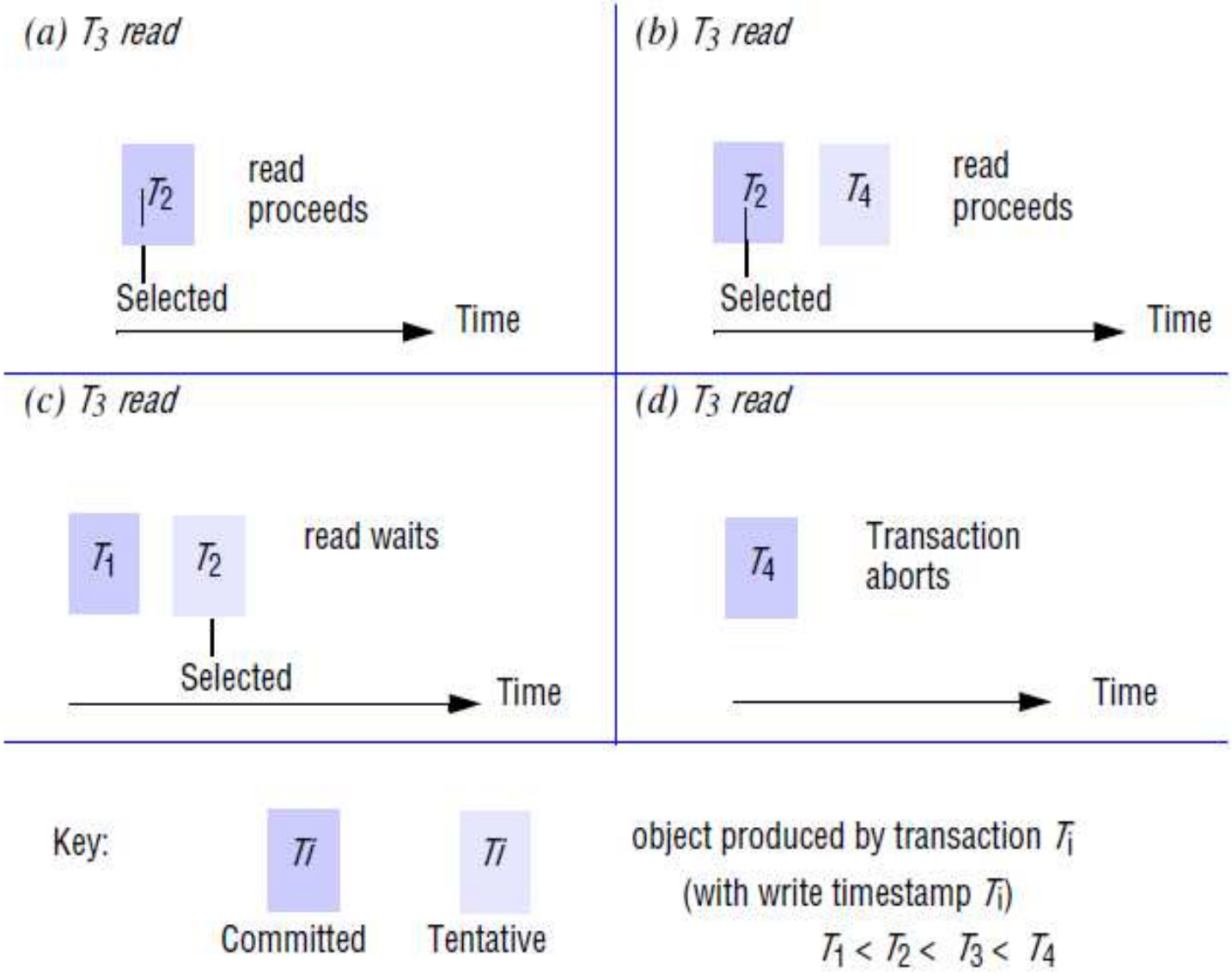
} else

Abort transaction T_c

Read Operation



Read Operation



Timestamps in transactions T and U

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

S<T<U

Timestamps in transactions T and U

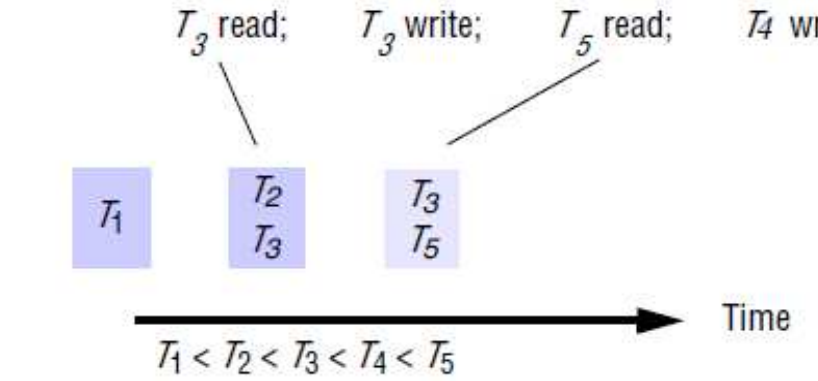
		Timestamps and versions of objects					
T	U	A		B		C	
		RTS	WTS	RTS	WTS	RTS	WTS
		{}	S	{}	S	{}	S
openTransaction							
bal = b.getBalance()				{T}			
	openTransaction						
b.setBalance(bal*1.1)					S, T		
	bal = b.getBalance()						
	wait for T						
a.withdraw(bal/10)	•••		S, T				
commit	•••		T		T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

S<T<U

Multi-version Timestamp Ordering

- A **list** of **old committed versions** as well as **tentative versions** is **kept** for each **object**.
- This **list** represents the **history** of the **values** of the **object**.
- The **benefit** of using **multiple versions** is that **read** operations that **arrive too late need not** be **rejected**.
- Write that arrive too late may invalidate read. Such write needs to be aborted.

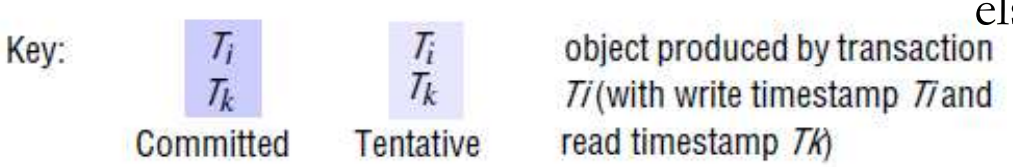
Late write would invalidate read



if (read timestamp of DmaxEarlier <= Tc)

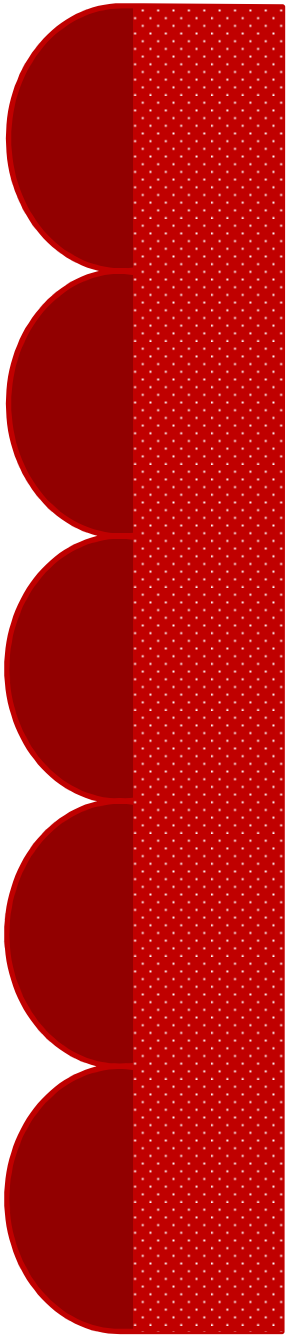
perform write operation on a tentative version of D with write timestamp Tc

else abort transaction Tc



T_3 read; T_3 write; T_5 read; T_4 write.

1. T_3 requests a *read* operation, which puts a read timestamp T_3 on T_2's version.
2. T_3 requests a *write* operation, which makes a new tentative version with write timestamp T_3.
3. T_5 requests a *read* operation, which uses the version with write timestamp T_3 (the highest timestamp that is less than T_5).
4. T_4 requests a *write* operation, which is rejected because the read timestamp T_5 of the version with write timestamp T_3 is bigger than T_4. (If it were permitted, the write timestamp of the new version would be T_4. If such a version were allowed, then it would invalidate T_5's *read* operation, which should have used the version with timestamp T_4.)



Summary

- Optimistic Concurrency Control
  - Backward Validation
  - Forward Validation
- Timestamp Ordering