Transactions III

Improving Concurrency
Optimistic Concurrency Control
CS3524 Distributed Systems
Lecture 09

Concurrency Control Protocols

- We enforce isolation with concurrency control protocols
- Two kinds of concurrency control behaviour
  - Transactions wait to avoid conflicts (pessimistic concurrency control)
  - Transactions are restarted after conflicts have been detected (optimistic concurrency control)
- Methods
  - Locking
  - Optimistic concurrency control
  - Timestamp ordering

Concurrency Control Protocols

- Locking
  - Most practical systems use locks
  - Locks can lead to deadlocks
- Optimistic Concurrency Control
  - Transaction proceeds until it starts committing
  - During commit, a discovery of conflicts with operations of other transactions has to be performed
    - If conflict then transaction abort and restart by client

Concurrency Control Protocols

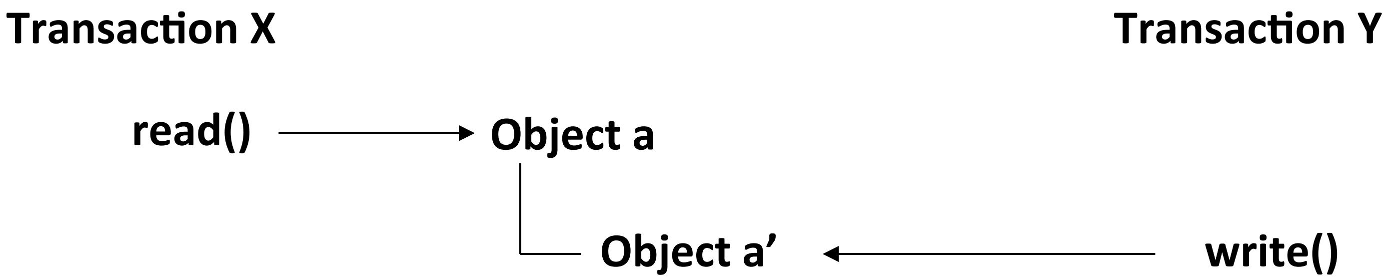
- Timestamp ordering
  - Server records most recent access time of read and write operations by any transaction on an object
  - For each operation performed by a transaction, the timestamp of the transaction is compared with that of the object (most recent access time) to determine whether
    - Operation can be done immediately
    - Has to be delayed (transaction waits)
    - Has to be rejected (transaction is aborted)

Improving Concurrency

Improving Concurrency

- Simple exclusive locks
  - Too restrictive
- Improve Concurrency with shared read locks
  - Read locks can occur concurrently
- Improving concurrency even further: Two-version Locking
  - Allows also shared write locks
  - Checking conflicts between Read / Write and Write / Write operations at transaction commit

Two-Version Locking



- Two-version Locking operates with two versions of the same data object when transactions read / manipulate it
- Transactions operate with a new “tentative” copy of a data object, other transactions read from the original committed and “visible” version

Two-Version Locking

- Two-version Locking improves concurrency:
  - Delays setting exclusive locks (even for write operations) until commit
  - checks for conflicts at commit (with that it can be regarded as an “optimistic concurrency control” scheme) and resolves these conflicts

Two-Version Locking

- Two-version Locking operates with three types of locks:

  Read Lock, Write Lock, Commit Lock

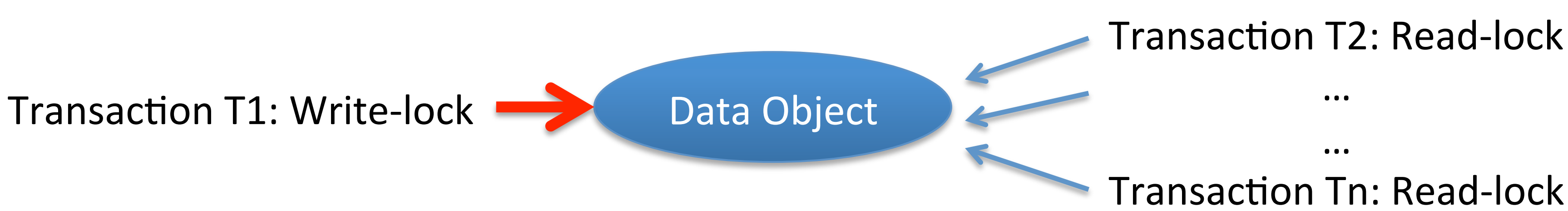
- Read and Write Locks are shared Locks
  - There can be **multiple** shared Read Locks: a transaction can set a read lock, if there are other read locks or one write lock
  - There can be **one** transaction that may hold a shared write lock, if other transactions hold read locks at the same time
- An exclusive Lock is only set at **Commit** – the "**Commit Lock**"
  - Read operations only wait if another transaction is currently committing a **write operation** on the same object and holding a **Commit Lock**

Two-Version Locking

- Three types of locks

Read Lock, Write Lock, Commit Lock

- Read-locks and Write-locks are shared
  - There can be multiple shared Read-locks
  - There can be one transaction holding a Write-lock, while other transactions are holding Read-locks



Two-Version Locking
How to Commit

- How to commit data manipulations?
- Third lock: exclusive **Commit**-lock
  - Exclusive **Commit**-lock must be set during the commit phase for committing write operations
- **Lock Promotion**
  - Transaction that holds a Write-lock has to “promote” Write-lock to exclusive Commit-lock
  - This is only possible, if there are no concurrent shared Read-locks by other transactions (**Commit**-lock is exclusive!)
- Consequence:
  - All transactions with shared Read-locks must finish / commit first and release locks, before the writing transaction can commit
  - Reading transactions may delay the writing transaction !!

Two-Version Locking

		Transaction T2 requests lock		
		read	write	commit
Transaction T1 Lock already set on object	none	OK	OK	OK
	read	OK	OK	wait
	write	OK	wait	---
	commit	wait	wait	---

- Uses three locks:
  - Read, write, commit
- With two-version locking, a transaction can still acquire a write lock for a data object, if there are already read locks
- This has consequences for committing such write operations
  - A transactions cannot commit its write operations as long as other uncompleted transactions have read locks on the same objects – they cannot "promote" their write locks to commit locks

Two-Version Locking

- Conflict Rules
  - A transaction can set a **Read Lock** on an object, if the object already has one or more read locks or one Write Lock
    - If the object has a Commit Lock (exclusive lock) then the transaction has to wait
  - A transaction can set a **Write Lock** on an object, if the object already has one or more Read Locks
    - If there is already a Write lock or a Commit Lock (exclusive lock) on this object then the transaction has to wait
  - A transaction can promote a Write Lock to a **Commit Lock**, if the data object does not have any read locks

Two-Version Locking

- It is an alternative to other locking schemes and based on a trade-off between risk of conflict and overhead of lock management schemes such as two-phase locking
- Advantage
  - Read operations only wait if another transaction is currently committing updates on the same object
- Problem
  - Unfortunately, transactions that perform write operations risk waiting (for read locks to be released) when they attempt to commit or have to wait to gain a write lock (commit lock already set by another transaction)

Improving Concurrency

- Locks serialise access to date objects and guarantee consistency, but transactions have to wait
- This counteracts the goal of fast, concurrent access to date by multiple users
- Alternative
  – Use an optimistic concurrency scheme --> “act first, look later”

Optimistic Concurrency Control

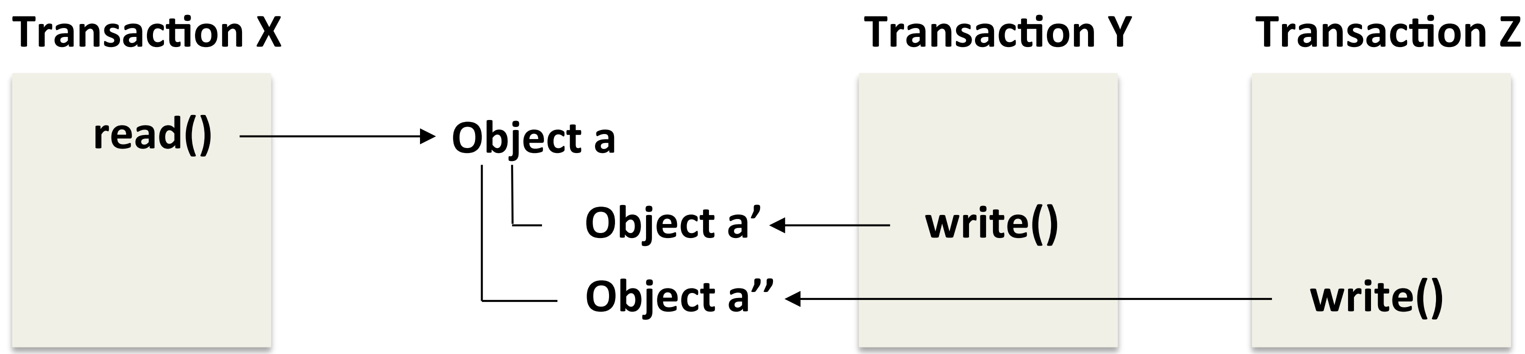
Optimistic Concurrency Control

- Optimistic Concurrency Control tries to overcome disadvantages of locking schemes, such as:
  - Lock maintenance is a computational overhead
  - Also read operations need locking
  - The use of locks can result in deadlocks
    - Deadlock prevention reduces concurrency, therefore deadlock detection or timeouts must be used
  - Locks cannot be released until the end of the transaction – reduces concurrency of transactions

Optimistic Concurrency Control

- Based on the observation that the likelihood of two clients accessing the same object is low
- Uses conflict detection:
  - Transactions proceed as if no conflicts may occur
  - Conflicts between transactions are checked at the end of each transaction
  - In case of conflict, transactions are aborted and have to be restarted by client

Optimistic Concurrency Control



- Each transaction operates on a tentative copy of each of the objects it manipulates (write)
- Transactions have three phases
  - Working phase
  - Validation phase
  - Update phase
- Transactions can be aborted any time, as they operate on tentative copies after write() operations

Transaction
openTransaction()
. . .
read() and write() operations
. . .
closeTransaction()

Work phase
Validation phase
Update phase

Optimistic Concurrency Control

Tentative Copies

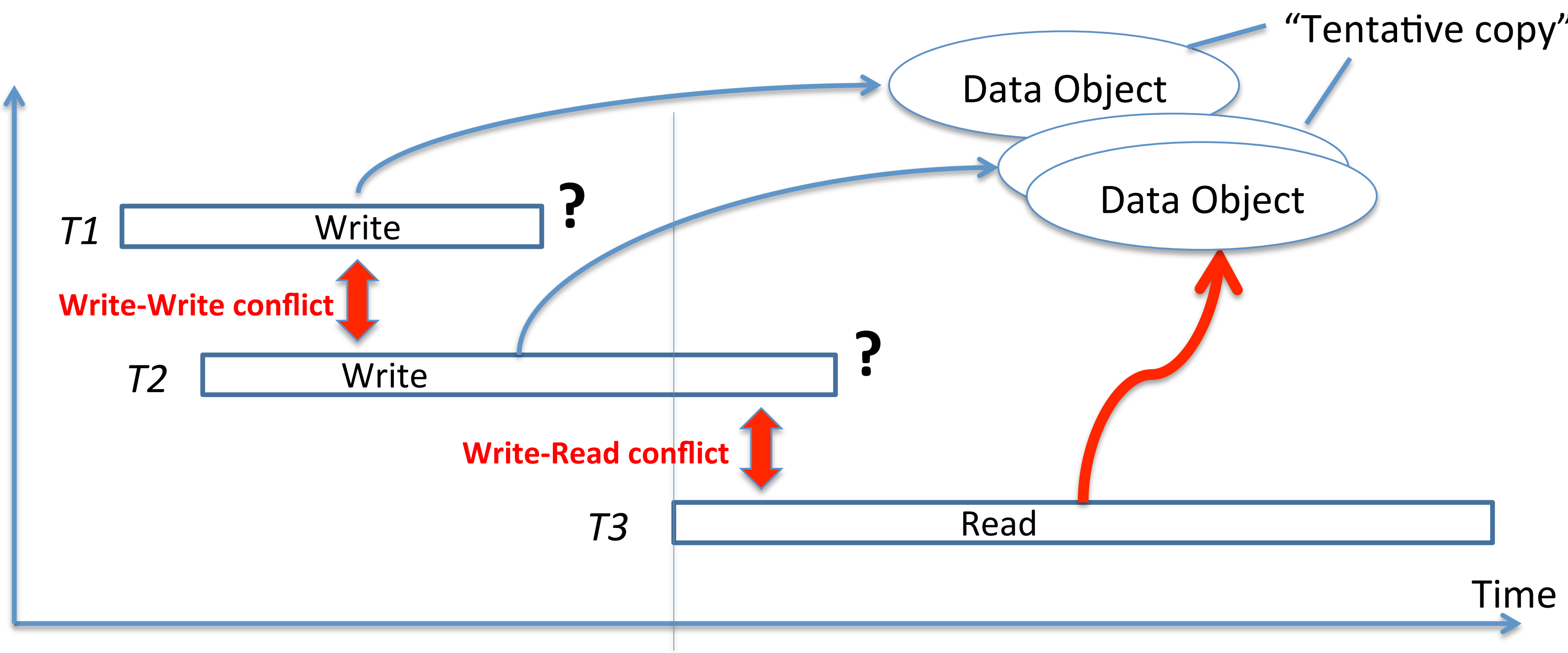
- Introducing tentative copies when write operations occur, improves concurrency
  - Read and write operation can occur concurrently
    - “Reading” transactions still see a committed data object
    - “Writing” transactions are not blocked, but can occur concurrently, because they operate on a copy of the data object
- It allows transactions to abort without affecting any other transaction or the “visible” (latest committed) state of the object itself

Optimistic Concurrency Control

Committing Change

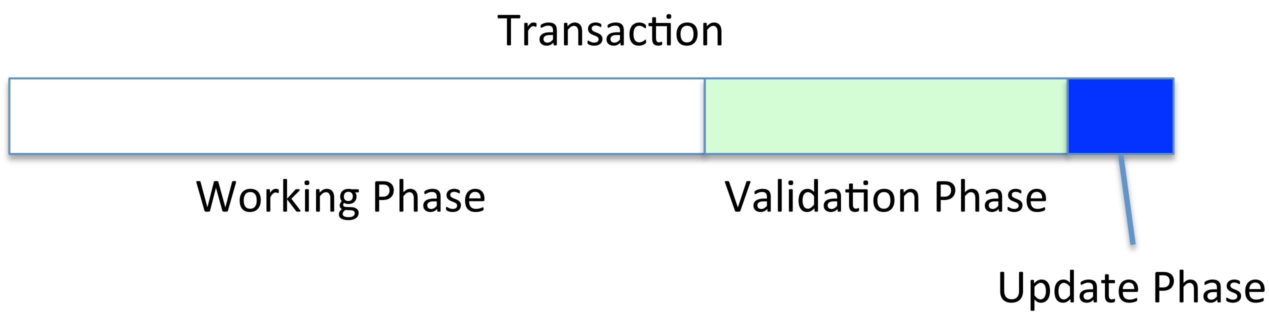
- At transaction commit, a tentative copy is supposed to become the new actual committed version of the data object
- Problem:
  - **There may be Write-Read conflicts between different transactions**
    - Writing transaction commits earlier than reading transaction – inconsistent retrieval
  - **There may be Write-Write conflicts between transactions**
    - Race condition: the transaction that commits its updates later, wins the race condition
- We have to carefully "untangle" all these transactions in order to avoid the classic problems "lost update" or "inconsistent retrievals" - we have to check for serial equivalence!!

When to Commit ?



- Is T2 allowed to commit?
  - There is a conflict with T1: T1 will overwrite update of T2, Race Condition!
- Is T1 allowed to commit?
  - Conflict with T2 (Race Condition)
  - Also conflict with T3: T3 holds a read lock, T1 cannot update current visible version of Data Object (Inconsistent Retrieval)

Optimistic Concurrency Control



- Transaction phases:
  - A transaction has three phases
    - Working phase:
      - Read and write actions happen
    - Validation phase:
      - Check, whether conflicts exist with other transactions and whether to commit or abort
    - Update phase
      - If validation is OK, update

Optimistic Concurrency Control
Working Phase

- When such a transaction is *opened*, the Working phase starts:
  - Write operations:
    - record the updated version of an object as a new tentative copy of the most recently committed "visible" version of the object
    - Tentative copies are invisible to other transactions
  - Read operations:
    - are performed immediately, either on an existing tentative copy or the last original committed version of the object
- A transaction maintains two records:
  - **Read set**: all objects read by the transaction
  - **Write set**: all objects written by the transaction

Optimistic Concurrency Control
Validation Phase

- When a transaction is closed, it enters the Validation phase
  - it has to be validated against all overlapping transactions that access the same shared objects:
    - Check whether there is a **conflict** with operations of other overlapping transactions

If validation is successful, transaction can enter the Update phase and is committed

If validation fails, a conflict exists:
- Either abort the validated transaction OR
- Abort any other conflicting transaction

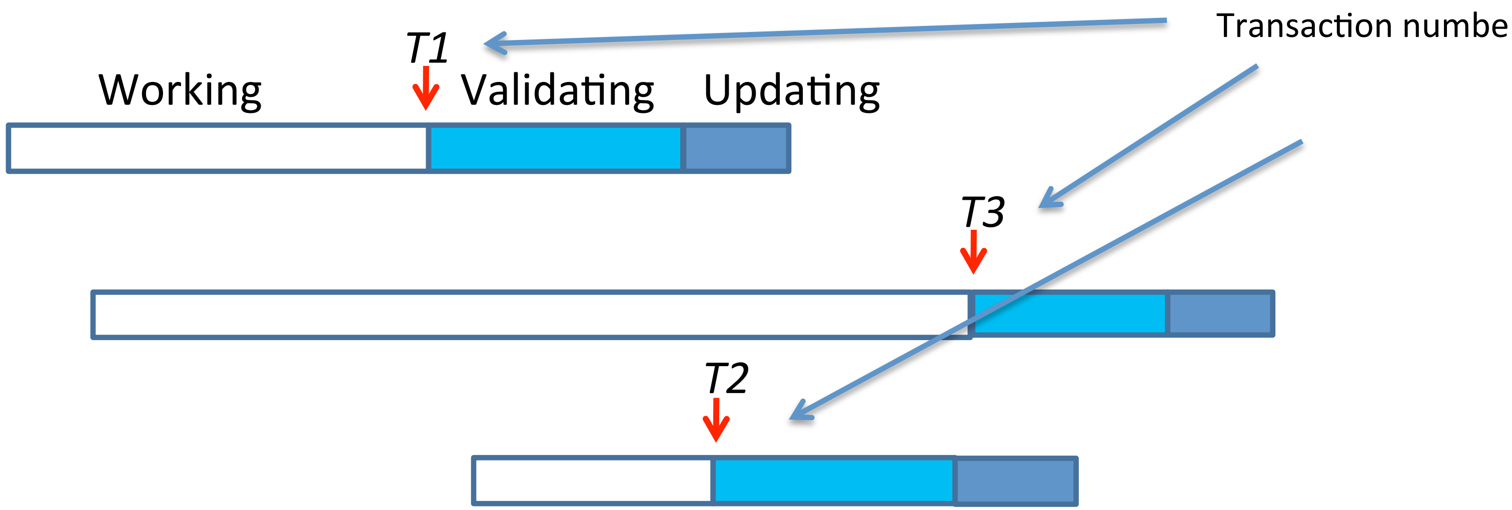
Transaction
openTransaction()
. . .
read() and write() operations
. . .
closeTransaction()

Work phase

Validation phase

Update phase

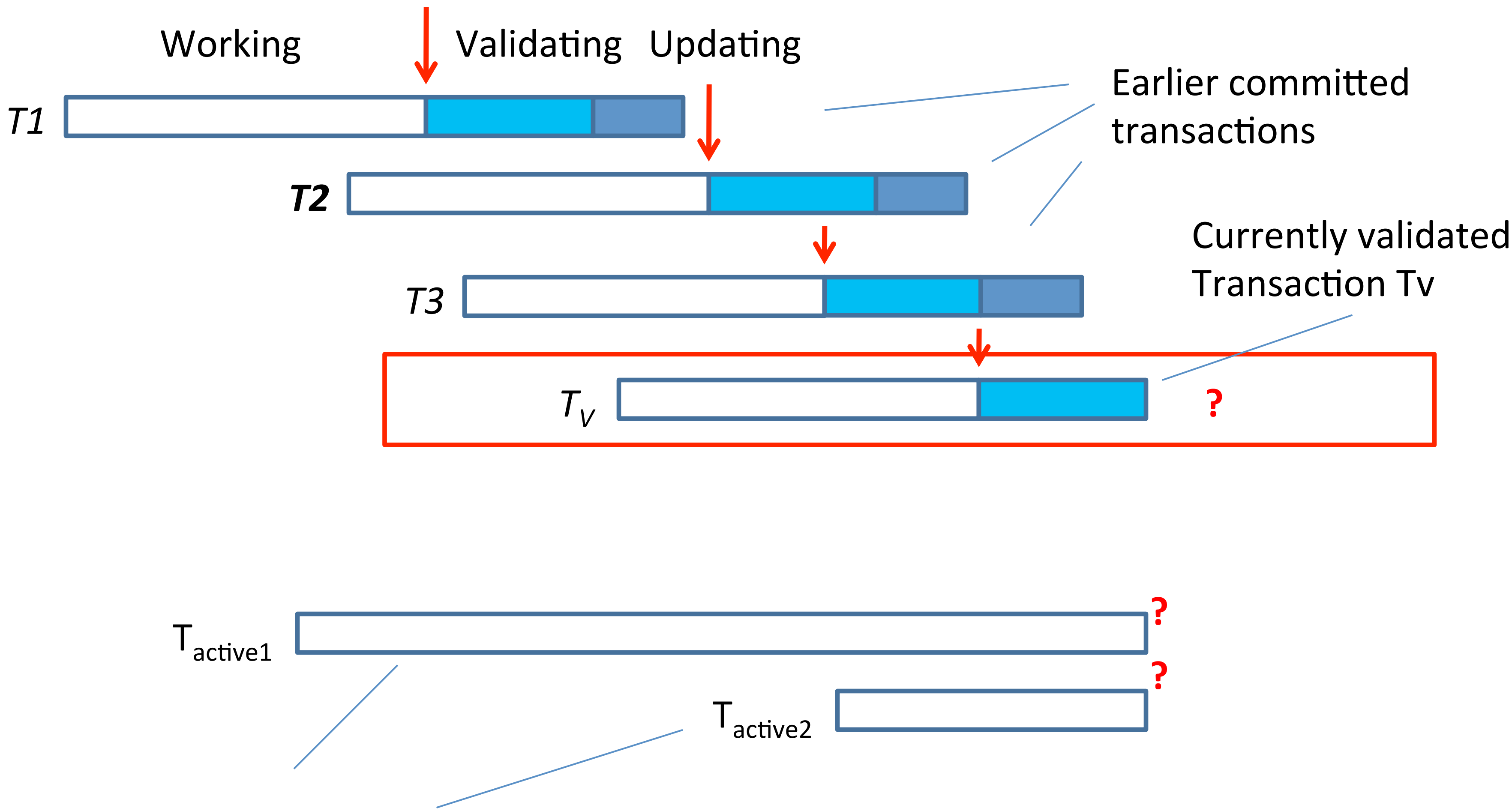
Validation of Transactions



- Each transaction that enters the validation phase is assigned a "transaction number" (may also be called the *validation* number)
  - A transaction number is an integer assigned in ascending sequence
  - The transaction number defines a sequence over validated transactions in time
  - If a transaction is successfully committed, it retains this number, otherwise it is re-assigned
- All transactions still in their working phase are identified with a separate unique transaction ID

Optimistic Concurrency Control

Validation of Transactions



Later, still active Transactions, do not have a transaction number yet, because they did not enter their Validation phase

Validation of Transactions Conflict Rules

- The validated transaction Tv is checked against all overlapping transactions Ti that also access a particular object
  - For each pair (Tv, Ti), the read-write conflict rules are tested whether they indicate a conflict and, therefore, non-serialisability of Tv
- We can say
  - In order for a transaction Tv to be serializable with respect to an overlapping transaction Ti, their operations must conform to the following rules:

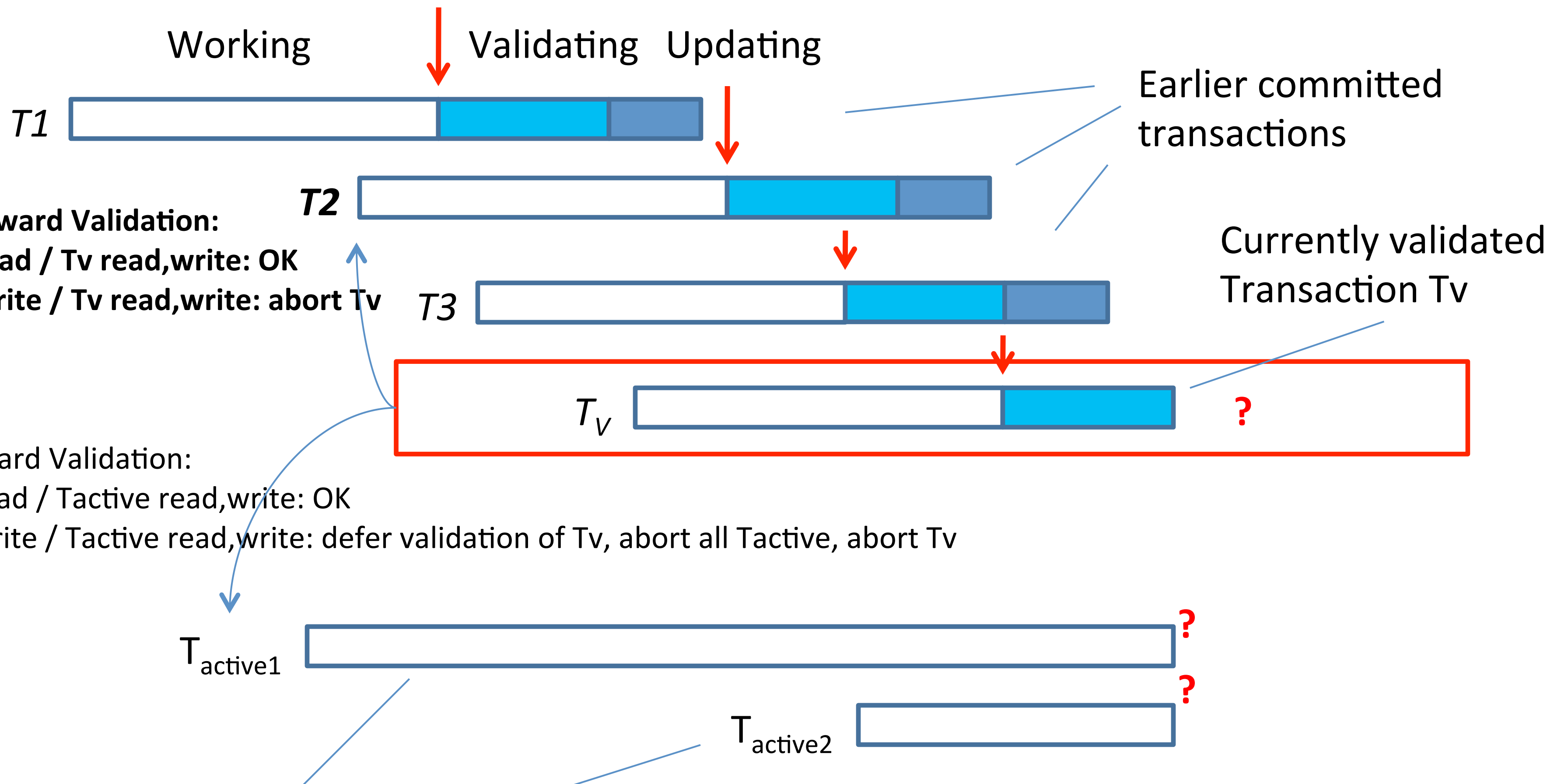
Ti	Tv	Rule
Read	Write	Rule 1: Ti must not read objects written by Tv
Write	Read	Rule 2: Tv must not read objects written by Ti
Write	Write	Rule 3: Ti must not write objects written by Tv and Tv must not write objects written by Ti

Validation of Transactions

- The validated transaction has to be tested against two sets of concurrently executing transactions:
  - Backward Validation
    - Transactions already validated / committed
  - Forward Validation
    - Transactions still active / not validated
- Validation test is based on possible conflicts between operations of pairs of overlapping transactions
  - Conflict between write and read operations:
    - Conflicts occur depending on whether a transaction performs a **read operation** **before or after** another transaction performs a **write operation** on a shared object
  - Conflict between write operations:
    - Write operations of two overlapping transactions are are always in conflict

Optimistic Concurrency Control

Validation of Transactions



Backward Validation:
T2 read / Tv read,write: OK
T2 write / Tv read,write: abort Tv

Forward Validation:
Tv read / Tactive read,write: OK
Tv write / Tactive read,write: defer validation of Tv, abort all Tactive, abort Tv

Later, still active Transactions, do not have a transaction number yet, because they did not enter their Validation phase

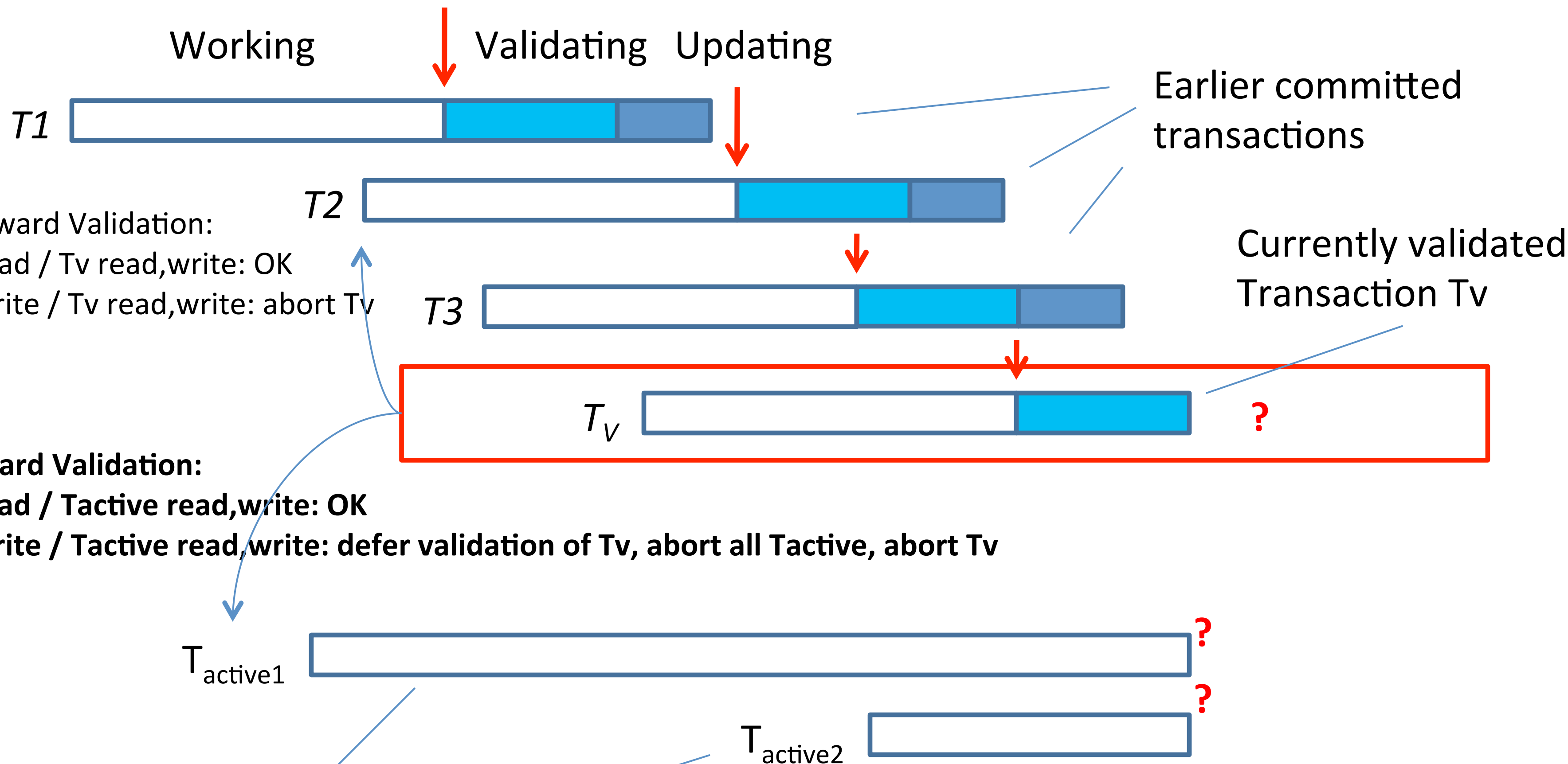
Optimistic Concurrency Control

Backward Validation

- Backward Validation:
  - Write operations of T_v:
    - **no conflict with read operations that were performed by already validated transactions, T_v is the last transaction to be validated and updates the shared data object with the tentative copy**
  - The problem are read operations of T_v:
    - these are in conflict with write operations from earlier transactions T_i
    - Inconsistent Retrieval !
    - Abort the validated transaction T_v

Optimistic Concurrency Control

Validation of Transactions



Backward Validation:
T2 read / Tv read,write: OK
T2 write / Tv read,write: abort Tv

Forward Validation:
Tv read / Tactive read,write: OK
Tv write / Tactive read,write: defer validation of Tv, abort all Tactive, abort Tv

Later, still active Transactions, do not have a transaction number yet, because they did not enter their Validation phase

Optimistic Concurrency Control

Forward Validation

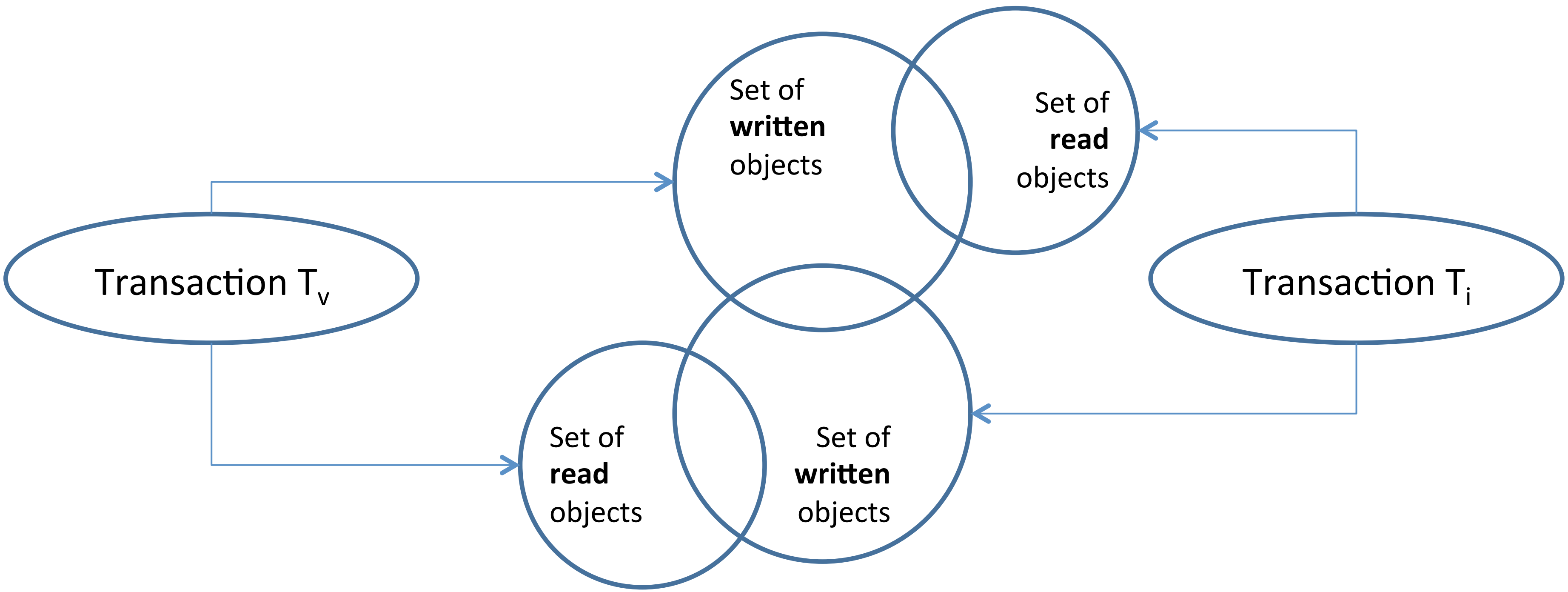
- Forward Validation
  - Read operations of T_v:
    - **no conflict with write operations performed by these unfinished concurrent transactions, because T_v is already in the validation phase and will finish before these concurrent transaction commit their write operations**
  - The problem are write operations of T_v:
    - these are in conflict with read operations from later transactions T_i
    - Strategies
      - Defer validation of T_v, until conflicting transactions have finished – this means that T_v will perform updates on objects after these conflicting transactions finished
      - Abort all the conflicting active transactions and commit T_v
      - Abort the validated transaction T_v

Optimistic Concurrency Control

Validation of Transactions

- How to test for conflict?
  - For each transaction, we record
    - a **read set of objects that have been read** and
    - a **write set of objects that have been written**
  - We test whether there is an intersection of these sets
    - we have check the **intersection** of the read set of one transaction with the write set of a conflicting transaction
    - We have to check the intersection of both write sets
    - if there is at least one object that is in both sets then there is a **conflict**

Validation of Transactions
Testing the Conflict Rules



- Investigate the ***intersection*** of read and write sets of the transactions

Optimistic Concurrency Control

Backward Validation

- Backward Validation:
  - check T_v against concurrent transactions T_i that are already committed and still relevant to T_v
  - Transaction numbers help us to select the **relevant set** of concurrent transactions T_i:
    - Is the set of committed transactions whose working phases overlapped with the working phase of T_v and were committed **after** T_v started and **before** T_v entered validation phase
  - Check the **read set** of T_v against the **write set** of each Ti and test whether there is an intersection

```
beforeRelevant ...  Last recorded transaction number
                    before Tv started
boolean valid = true ;
for ( int ti = beforeRelevant + 1; ti < tv; ti ++ ) {
       if ( read-set of tv intersects with write set of ti )
            valid = false ;
}
```

Optimistic Concurrency Control

Forward Validation

- Forward Validation
  - check T_v against concurrent transactions T_{id} that are still active and in their working phase
  - Check the **write set** of T_v against the **read set** of each T_{id} and test whether there is an intersection

```
boolean valid = true ;
for ( int tid = active1; tid <= activeN; tid ++ ) {
       if ( write-set of tv intersects with read-set of tid )
            valid = false ;
}
```

Optimistic Concurrency Control

Closing the Transaction – Validation and Update

- Update Phase
  - If a transaction is validated successfully, all of the changes recorded in its tentative versions are made permanent.
  - Read-only transactions (contains only read() operations) can commit immediately after passing validation
  - Transactions with write() operations are ready to commit once the tentative versions of the objects have been recorded in permanent storage

Timestamp Ordering

- Assign a unique timestamp to a transaction at its start (transaction id that is incremented for each transaction started)
- Each object receives a read and write timestamp from committed transactions
  - Which committed transaction last read the object
  - Which committed transaction last wrote the object
- Correct ordering:
  - If a transaction wants to read an object then its own timestamp must be younger than the object's own write time stamp
  - If a transaction wants to write an object then its own timestamp must be younger than the object's own read / write time stamps
- Improper ordering: abort and restart transaction