

Distributed Relational Databases

Thomas Schwarz, SJ

Why?

- Parallelism is a simple way to scale up performance
 - Example: Google web searches implemented via the Google File System
 - Every file is stored at at least three servers (~2010)
 - Unlikely to need to access an overloaded server
 - Capacity of each server only used a little
 - Because bandwidth is more important than storage capacity

Why?

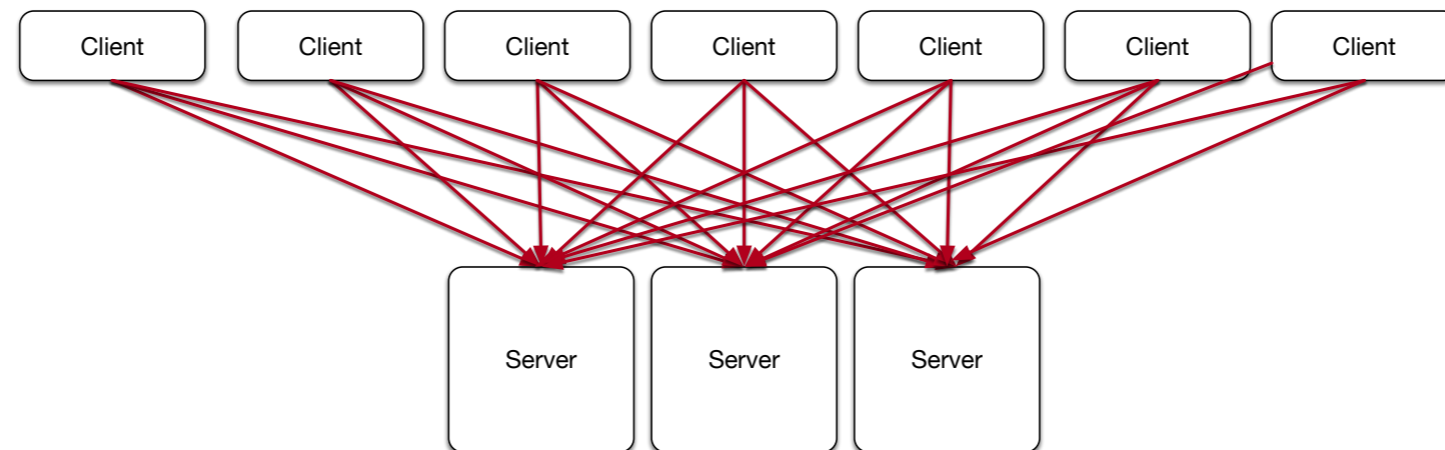
- Local accesses
 - A global enterprise has users around the world
 - To save latencies, might create data centers around the world as well
 - Central databases are then replicated in those data centers

Why?

- Heterogeneity
 - Your company buys a competitor
 - Now you have two customer databases
 - They contain different types of information
 - Differences in the details, at the macro level same functionality
 - Difficult to combine the databases
 - Easier:
 - "Federate them"

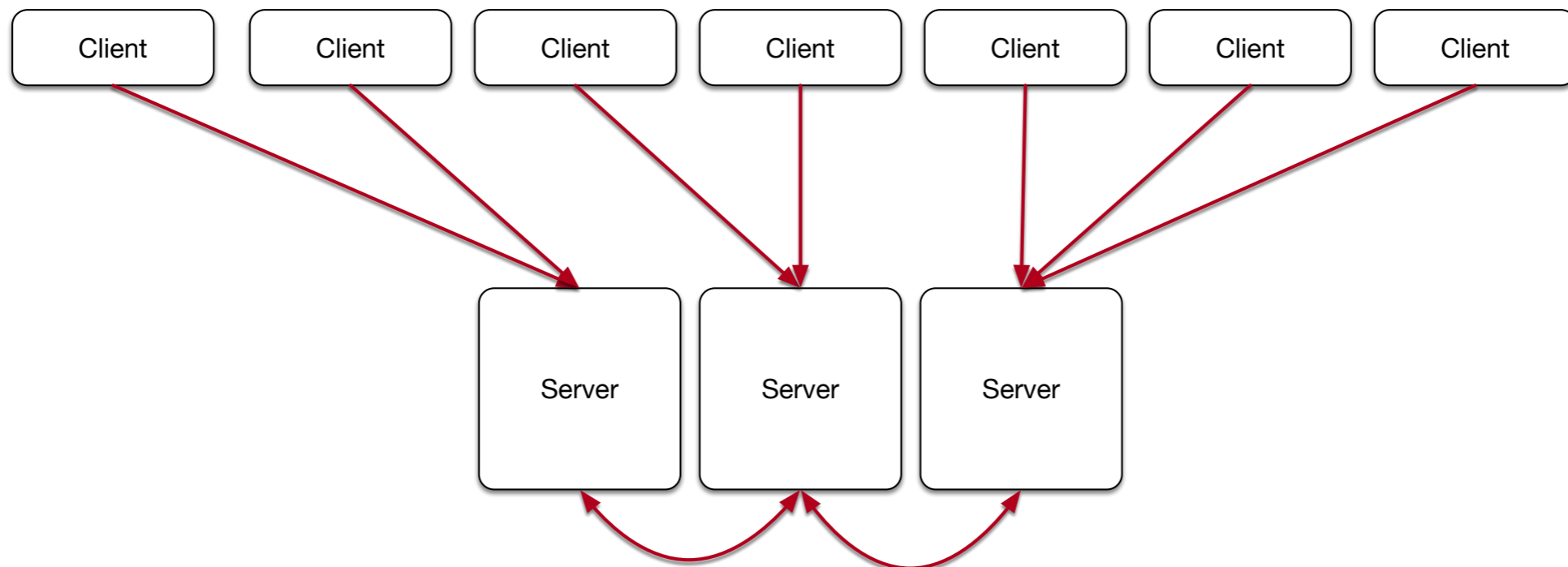
Client Server Systems

- Two-tier architecture
 - Client has client DBMS
 - Can communicate by issuing SQL commands
 - Servers have server DBMS



Client Server Systems

- Two architectures:
 - Every client can demand services from every server
 - Every client has a home server that can forward demands to another server



P2P Systems

- Server components are distributed over clients
 - Might or might not have a global directory replicated at all clients

What Makes Distributed Databases Difficult

- Finding information
- Coordination
 - Updating the database should leave it into a coherent state
- Example: Global email system
 - Use a VPN to go to Mumbai and send an email
 - Disconnect the VPN and find out that you did not yet send the email

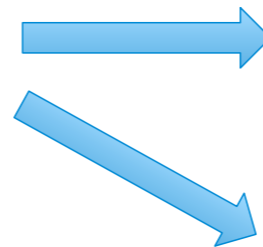
Distributing Data

- Replication
 - Motivation: Speed of access to data
- Fragmentation
 - Break relations up

Distributing Data

- Horizontal fragmentation
 - Divide the rows into different fragments
 - Example: Employees Database
 - Primary decomposition of Employees Table based on emp_no

100254	José Miguel	Ocampo	M	1967-05-08
100256	Angh	Pham	F	1959-12-04
100257	Joseph	Smith	M	1947-08-28
100258	Witold	Litin	M	1998-02-27
100259	Alethea	Parker	F	2001-05-12
100260	Deborah	Rabin	F	1993-08-19
100261	Fulton	Dagger	M	1982-06-21



100254	José Miguel	Ocampo	M	1967-05-08
100256	Angh	Pham	F	1959-12-04
100258	Witold	Litin	M	1998-02-27
100260	Deborah	Rabin	F	1993-08-19

100257	Joseph	Smith	M	1947-08-28
100259	Alethea	Parker	F	2001-05-12
100261	Fulton	Dagger	M	1982-06-21

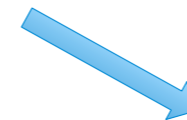
Distributing Data

- Example: Employees Database
 - Since we frequently join on emp_no, we should now do secondary fragmentation on emp_no of salary

100254	1985-03-19	1985-12-31	39,000.00
100254	1986-01-01	1986-12-31	39,500.00
100254	1987-01-01	1987-12-31	40,500.00
100254	1988-01-01	1988-12-31	41,700.00
100256	1982-07-01	1982-12-31	58,750.00
100256	1983-01-01	1983-12-31	60,200.00
100256	1984-01-01	1985-12-31	62,000.00
100256	1985-01-01	1985-12-31	62,100.00
100256	1986-01-01	1986-12-31	63,000.00
100256	1987-01-01	1987-12-31	65,500.00
100256	1988-01-01	1988-12-31	69,000.00
100257	1988-09-01	1988-12-31	89,000.00
100258	1983-11-01	1983-12-31	44,200.00
100258	1984-01-01	1984-12-31	44,500.00
100258	1985-01-01	1985-12-31	44,500.00
100258	1986-01-01	1986-12-31	63,000.00
100258	1987-01-01	1987-12-31	65,500.00
100258	1988-01-01	1988-12-31	69,000.00
100259	1986-03-01	1986-12-31	89,000.00
100259	1987-01-01	1987-12-31	98,000.00
100259	1988-01-01	1988-05-31	99,900.00



100254	1985-03-19	1985-12-31	39,000.00
100254	1986-01-01	1986-12-31	39,500.00
100254	1987-01-01	1987-12-31	40,500.00
100254	1988-01-01	1988-12-31	41,700.00
100256	1982-07-01	1982-12-31	58,750.00
100256	1983-01-01	1983-12-31	60,200.00
100256	1984-01-01	1985-12-31	62,000.00
100256	1985-01-01	1985-12-31	62,100.00
100256	1986-01-01	1986-12-31	63,000.00
100256	1987-01-01	1987-12-31	65,500.00
100256	1988-01-01	1988-12-31	69,000.00
100258	1983-11-01	1983-12-31	44,200.00
100258	1984-01-01	1984-12-31	44,500.00
100258	1985-01-01	1985-12-31	44,500.00
100258	1986-01-01	1986-12-31	63,000.00
100258	1987-01-01	1987-12-31	65,500.00
100258	1988-01-01	1988-12-31	69,000.00



100257	1988-09-01	1988-12-31	89,000.00
100259	1986-03-01	1986-12-31	89,000.00
100259	1987-01-01	1987-12-31	98,000.00
100259	1988-01-01	1988-05-31	99,900.00

Distributing Data

- Horizontal fragmentation
 - Use a (simple) predicate on a primary code on the principal relation
 - In our example: $\text{emp_no} \% 2 == 0$, $\text{emp_no} \% 2 == 1$
 - Use derived fragmentation for other tables

Distributing Data

- Vertical fragmentation
 - Break table into different attributes
 - Need to repeat primary keys
 - (Or introduce a new artificial tuple identifier functioning as a primary key)
 - Keep attributes together that appear in many projections and selections together

Distributing Data

- Example: Employees Table

100254	José Miguel	Ocampo	M	1967-05-08
100256	Angh	Pham	F	1959-12-04
100257	Joseph	Smith	M	1947-08-28
100258	Witold	Litin	M	1998-02-27
100259	Alethea	Parker	F	2001-05-12
100260	Deborah	Rabin	F	1993-08-19
100261	Fulton	Dagger	M	1982-06-21



100254	José Miguel	Ocampo
100256	Angh	Pham
100257	Joseph	Smith
100258	Witold	Litin
100259	Alethea	Parker
100260	Deborah	Rabin
100261	Fulton	Dagger

100254	M	1967-05-08
100256	F	1959-12-04
100257	M	1947-08-28
100258	M	1998-02-27
100259	F	2001-05-12
100260	F	1993-08-19
100261	M	1982-06-21

Distributing Data

- Given statistics on queries, use a clustering algorithm to decide the break-up

Distributing Data

- It is possible to combine vertical and horizontal fragmentation

Query Processing

- Query plans now need to include the costs of sending data over a network
 - Usually, have different costs for what was previously the same query plan

Transaction Processing

Transaction Concept

- Transactions:
 - Transparent concurrency
 - Transparent recovery
- **Atomic**
 - Transaction is completely executed or not at all
- **Consistency**
 - Consistency constraints are preserved by a transaction
- **Isolation**
 - Each transaction behaves as if it were operating alone
- **Durability**
 - Updates made by a committed transaction are durable

Transaction Concept

- Computational model
 - Elementary operations on data objects
 - Transactions are sequences of these operations
 - The execution of several transactions is described by a schedule or history
 - Histories where ACID properties are guaranteed are correct
 - Generate algorithms / protocols

Transaction Concept

- Computational model: Page model
 - Transaction is a sequence of elementary operations
 - read, write
 - on a single page
 - Can talk about the value of a page at each level in the history

Transaction Concept

- Computational model
 - Example transaction:
 - $t = r(x)r(y)r(z)w(u)w(x)$
 - Transaction reads three values
 - Can use these values to write new values
 - One value has potentially changed
 -

Transaction Concept

- Not necessary to assume that the steps in a transaction are sequential
- Just need a partial order between the steps

Concurrency Control

- Canonical Concurrency Problems
 - Lost update problem

t1:

$r(x)$

$x += 30$

$w(x)$

t2:

$r(x)$

$x += 120$

$w(x)$

Concurrency Control

- Inconsistent read problem
 - Assume the constraint $x+y==0$

t1:

r (x)
r (y)

t2:

r (x)
x -= 10
w (x)
r (y)
y += 10

w (y)

Concurrency Control

- Dirty read problem (Reading uncommitted data)

t1:

r (x)

x+=100

w (x)

rollback

t2:

r (x)

x-=100

w (x)

Concurrency Control

- History
 - The sequence of all elementary operations of all transactions
- Schedule
 - A prefix of a history
- Serial History
 - A history where all elementary operations of one transaction are done before those of another transaction or where all elementary operations of one transaction are done after those of another transaction

Concurrency Control

- Herbrand Semantics
 - A notion to make precise what is the result of a history without specifying values
 - Because we could change a non-serial history to a serial one by altering values written
 - For example, set all transfer, withdrawal, and deposit amounts to 0 to make any history in a bank database serial

Concurrency Control

- Herbrand value of a read is the Herbrand value of the last write
- Herbrand value of a write is a generic functions of all reads of the executor of the write

Final state serializability

- Final state serializability
 - Final state equivalence
 - Two histories are final state equivalent if
 - They contain the same operations
 - They have the same Herbrand semantics

Final state serializability

$$s_1 = r_1(x)r_2(y)w_1(y)r_3(z)w_3(z)r_2(x)w_2(z)w_1(x)$$

$$s_2 = r_3(z)w_3(z)r_2(y)r_2(x)w_2(z)r_1(x)w_1(y)w_1(x)$$

- Final state equivalency example

Final state serializability

- Final state equivalency example

$$s_1 = r_1(x)r_2(y)w_1(y)r_3(z)w_3(z)r_2(x)w_2(z)w_1(x)$$

$$s_2 = r_3(z)w_3(z)r_2(y)r_2(x)w_2(z)r_1(x)w_1(y)w_1(x)$$

Herbrand value of x depends on what 1 saw, which is x unaltered in both schedules

Herbrand value of y depends on what 1 saw

Herbrand value of z depends on what 2 saw, which is x and y unaltered

Final state serializability

- Finite state equivalency example:

$$s_a = r_1(x)r_2(y)w_1(y)w_2(y)$$

$$s_b = r_1(x)w_1(y)r_2(y)w_2(y)$$

Final state serializability

- Finite state equivalency example:

$$s_a = r_1(x)r_2(y)w_1(y)w_2(y)$$

$$s_b = r_1(x)w_1(y)r_2(y)w_2(y)$$

The Herbrand value of y is the original value of y in the first schedule and depends on the write by 1 in the second schedule.

Final state serializability

- We can decide final state serializability with the Life-Reads-From relation
 - Reads-from
 - A transaction reads a value after it has been last written by another transaction
 - Alive
 - Final value depends on the transaction
- Two schedules are final state serializable iff
 - they consist of the same operations
 - they have the same life-reads-from relation

Final state serializability

- A schedule is final-state serializable
 - IFF it is final state equivalent to a serial schedule
 - IFF it has the same life-read-from relation as a serial schedule

View Serializability

- Final state serializability is still insufficient
 - Lost update anomaly is detected (good)

$$r_1(x)r_2(x)w_1(x)w_2(x)$$

- Inconsistent read is still allowed by final state serializability (bad)

$$r_2(x)w_2(x)r_1(x)r_1(y)r_2(y)w_2(y)$$

Transaction 1 makes an inconsistent read, but the final state ignores it.

View Serializability

- View equivalence
 - Two schedules are view equivalent if
 - They have the same set of operations
 - The Herbrand values of their schedules are equal
 - The Herbrand values at each read or write step are equivalent

View Serializability

- View equivalence
 - Two schedules are view equivalent
 - IFF they have the same read-from relation

Conflict Serializability

- Easier to test than view serializability
- Conflict relation:
 - Two operations are in conflict
 - If they access the same data item
 - At least one of them is a write
 - Conflict relation: transitive closure
- Conflict equivalence
 - Two schedules are conflict equivalent
 - They have the same set of operations
 - Their conflict set is the same

Conflict Serializability

$w_1(x)r_2(x)w_2(y)r_1(y)w_1(y)w_3(x)w_3(y)$

Conflict Serializability

$w_1(x)r_2(x)w_2(y)r_1(y)w_1(y)w_3(x)w_3(y)$

$\{(w_1(x), r_2(x)), (w_1(x), w_3(x)), (r_2(x), w_3(x)), (w_2(y), r_1(y)), (w_2(y), w_1(y)), (w_1(y), w_3(y))\}$

Conflict Serializability

- A schedule is conflict serializable
 - IFF it is conflict equivalent to a serial schedule

Conflict Serializability

- Algebraic notation:
 - C1 $r_i(x)r_j(y) \sim r_j(y)r_i(x)$ if $i \neq j$
 - C2 $r_i(x)w_j(y) \sim w_j(y)r_i(x)$ if $i \neq j$ and $x \neq y$
 - C3 $w_i(x)w_j(y) \sim w_j(y)w_i(x)$ if $i \neq j$ and $x \neq y$

Conflict Serializability

- Two schedules with the same operations are equivalent
 - Iff one can be transformed to the other using the commutativity rules

Commit Serializability

- Conflict serializability does not detect the dirty read problem
 - Since it does not pay attention to commit and abort operations
- Correctness criterion should only take committed transactions into account
- Since systems can crash
 - All prefixes of a schedule have to be correct

Commit Serializability

- A schedule is commit conflict serializable iff
 - Projection on committed transactions is conflict serializable

Concurrency Control Algorithms

- How to deal with bad situations:
 - Strategy 1: Never get into a bad situation
 - Strategy 2: Know how to get out of a bad situation (rollback)

Concurrency Control Algorithms

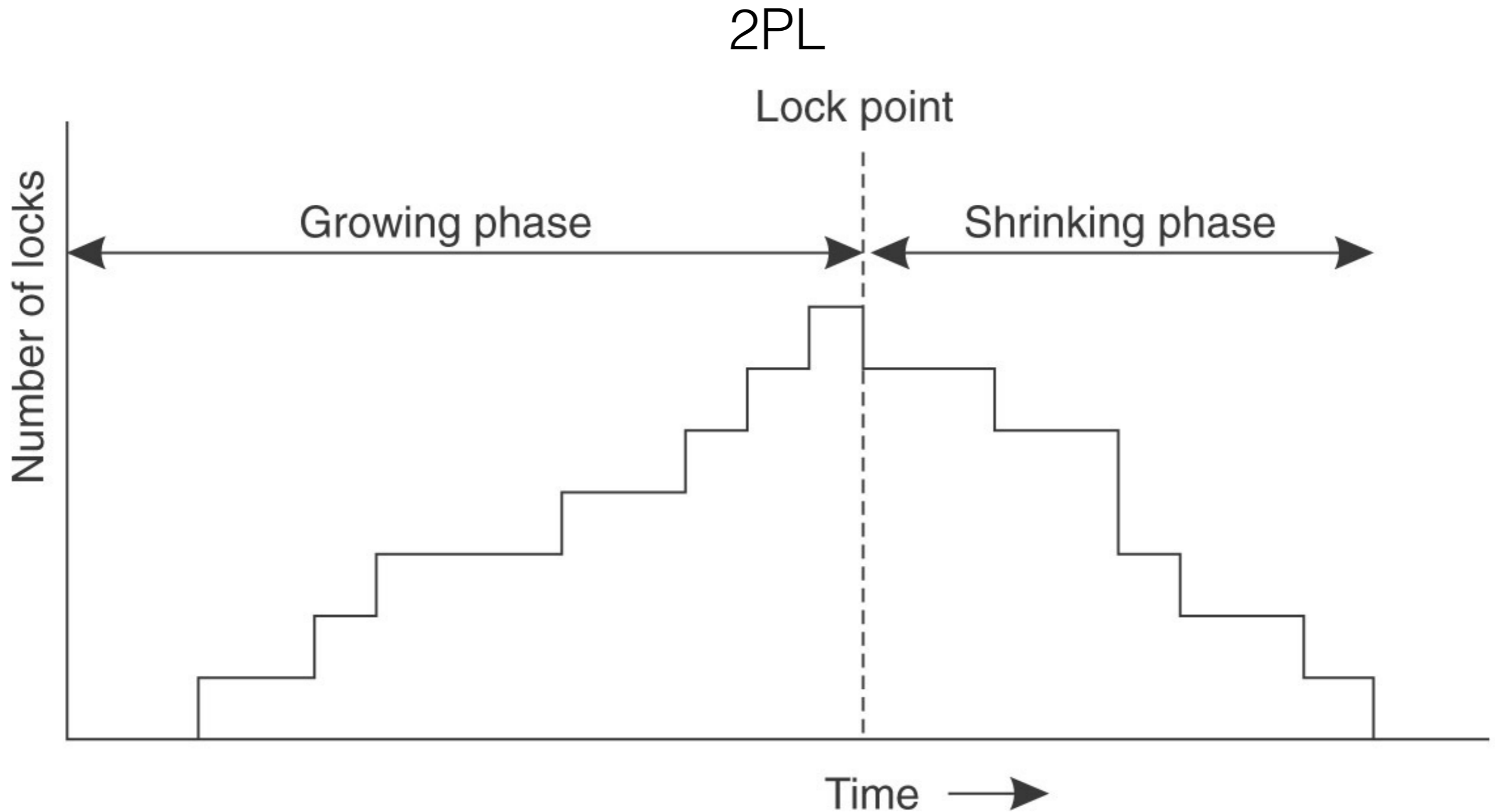
- Locking scheduler: Never get into a bad situation
 - Use locks on data items
 - Shared / Read Locks
 - Exclusive / Write Locks
- Locking problems:
 - Need to make sure that transactions release locks
 - Kill all Zombies!!!!
 - Need to avoid deadlock
 - Need to avoid life lock



Locking Algorithms

- No restrictions on locking are hard to get right
- ***Two phase locking*** (2PL)
 - All transactions pass first through a phase
 - where they only acquire locks
 - Then through a phase
 - where they only release locks
- A schedule with 2PL is conflict serializable
 - But not every conflict serializable schedule can be created with 2PL

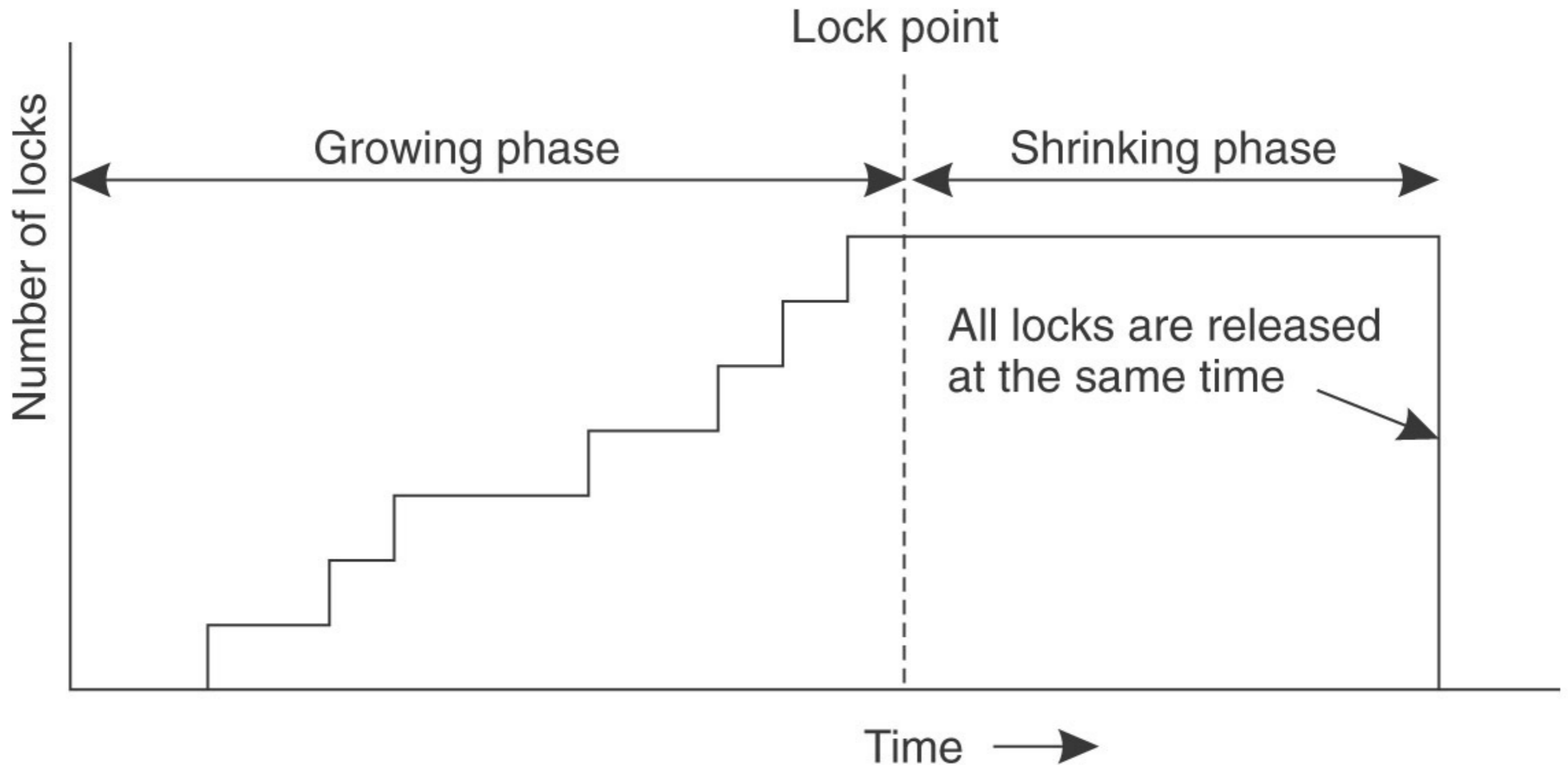
Locking Algorithms



Locking Algorithms

- 2PL
 - Dirty reads:
 - Allows transactions to read from transaction that are later aborted
- Strict 2PL
 - Release locks only at the end of a transaction
 - Allows easy automatization
 - Application wants to read an item
 - Automatically request lock
 - When committing, automatically release all locks

Locking Algorithms



Locking Algorithms

- Deadlock Handling
 - Normal lock requests can lead to deadlock

$$r_1(x)w_2(y)w_2(x)c_2w_1(y)c_1$$

- Yields

$$l_1(x)r_1(x)L_2(y)w_2(y) \dots$$

- / read-lock
- L write-lock

Locking Algorithms

- Deadlock handling
 - Lock conversion can also lead to deadlock

t_1 : $r_1(x)w_1(x)$

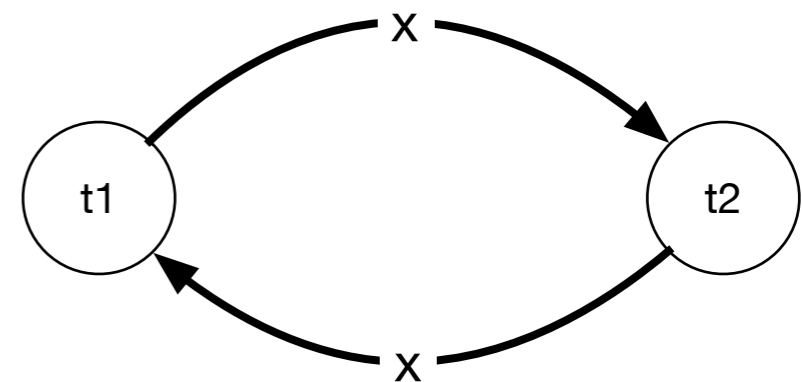
t_2 : $r_2(x)w_2(x)$

$l_1(x)r_1(x)l_2(x)r_2(x)????$

Locking Algorithms

- Deadlock Detection
 - Wait for graph
 - Nodes are transactions
 - Edges represent “waiting for”

t_1 : $r_1(x)w_1(x)$
 t_2 : $r_2(x)w_2(x)$
 $l_1(x)r_1(x)l_2(x)r_2(x)????$



Locking Algorithms

- Deadlock detection
 - Continuous detection
 - WFT is always kept cycle free
 - Periodic detection
 - Check WFT for cycles periodically

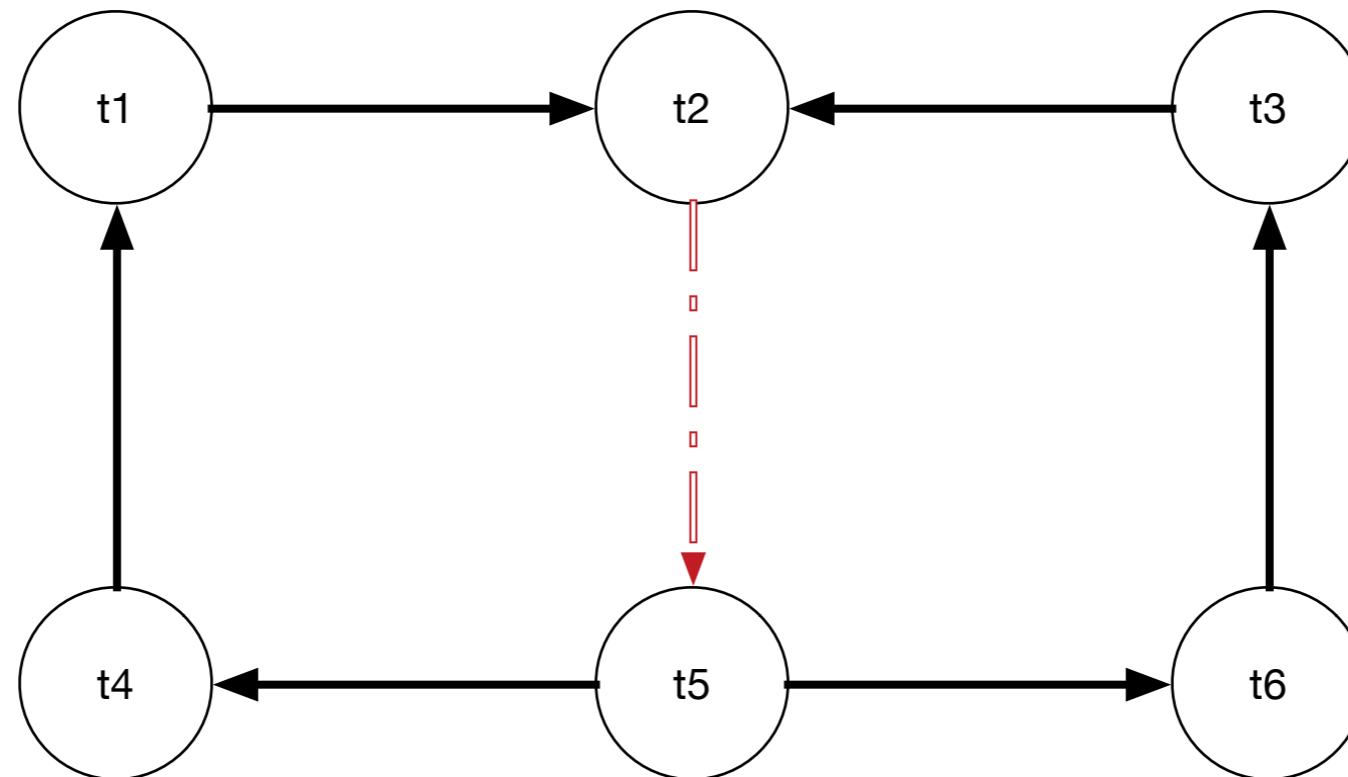
Locking Algorithms

- Deadlock detection
 - If a scheduler detects a cycle:
 - Aborts a *victim* transaction
 - according to heuristics
 - Last blocked
 - Random
 - Youngest
 - Minimum locks
 - Minimum work
 - Most cycles
 - Most edges

Locking Algorithms

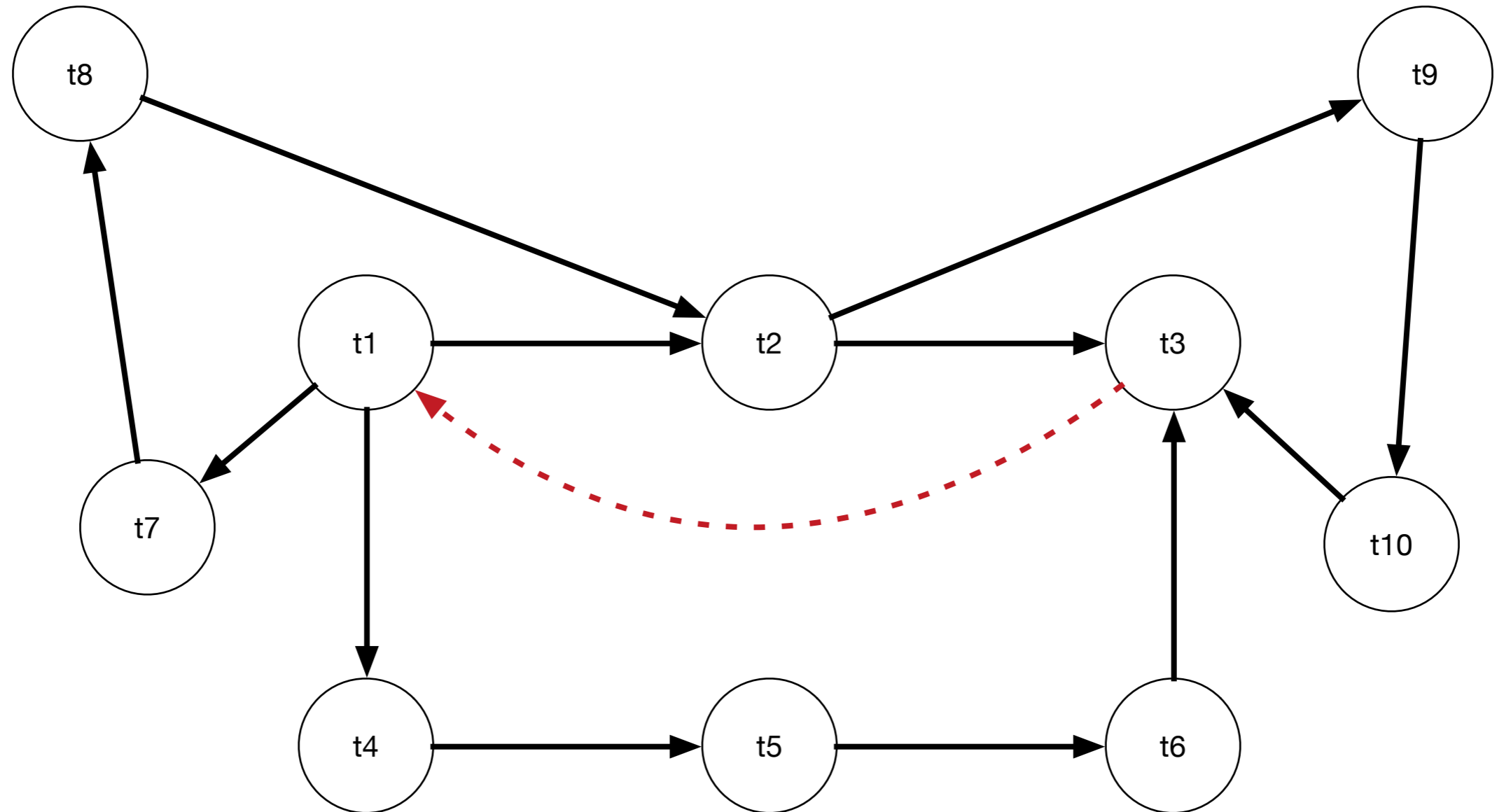
- Life-lock
 - All victim selection mechanism can create live-lock
 - Incarnations of the same transaction are always chosen
 - Various heuristics to avoid life-lock

Locking Algorithms



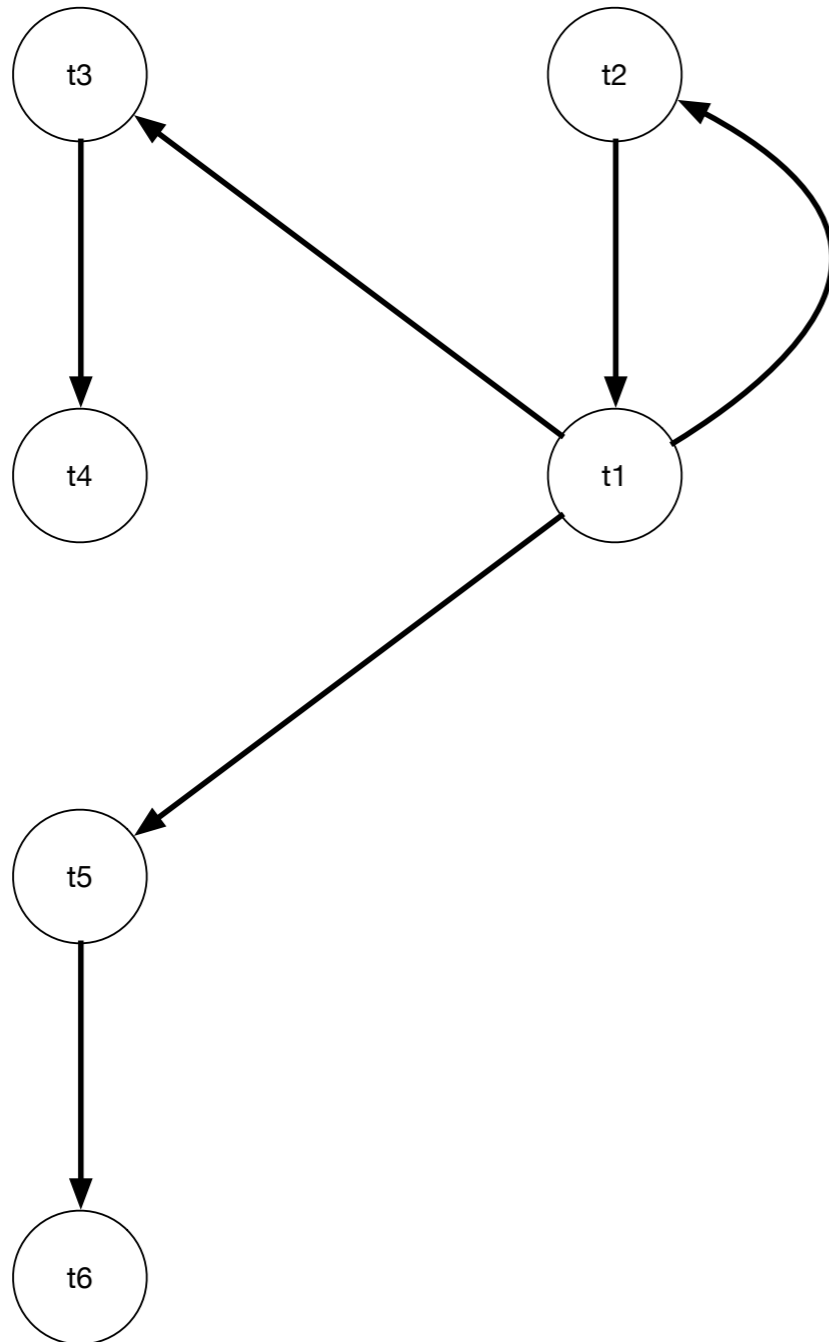
Most cycle heuristics:
Remove dashed edge by aborting
t2 or t5

Locking Algorithms



Dashed cycle breaks the most cycles

Locking Algorithms



WFG with cycle and two candidate victims (t1, t2)

Most edges option:

Abort t1: Two edges remain

Abort t2: Four edges remain

Hence: Abort t1

Locking Algorithms

- Deadlock prevention
 - Abort transaction whose lock request would create a cycle

Non-Locking Algorithms

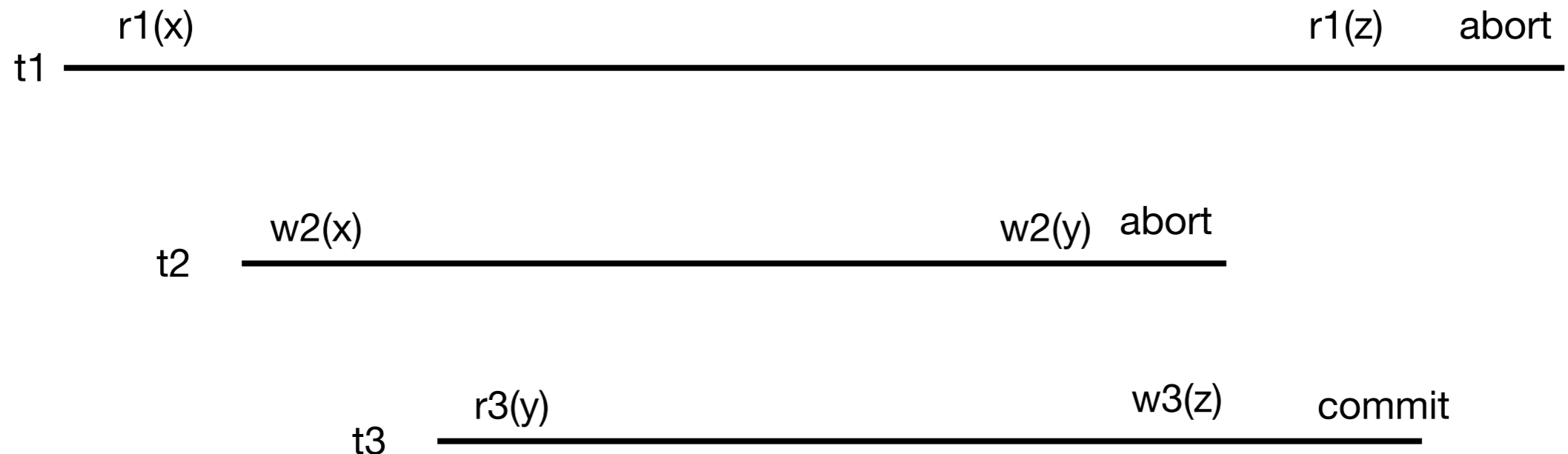
- Timestamp ordering
 - Each transactions gets a unique timestamp
 - Timestamp Ordering rule
 - All operations inherit their timestamp from the transaction
 - If two operations are in conflict, then the one with the smaller timestamp has to be done first
 - If this impossible, abort the offending transaction

Non-Locking Algorithms

- Basic time-stamp ordering (BTO)
 - For all data items, maintain the largest timestamp for
 - a read operation: $\text{max-r-scheduled}(x)$
 - a write operation: $\text{max-w-scheduled}(x)$

Non-Locking Algorithms

- BTO: Transactions can be too late and are aborted
 - $w_2(x)$ succeeds since $t_1 < t_2$
 - $r_3(y)$ succeeds since $t_3 < t_2$
 - $w_2(y)$ fails since $t_2 < t_3$, t_2 is aborted
 - $r_1(z)$ fails since $t_1 < t_3$, t_1 is aborted



Non-Locking Algorithms

- Optimistic protocols
 - Assume that conflicts are reasonably rare
 - Let transactions go ahead
 - But *validate* their serializability

Non-Locking Algorithms

- Optimistic Protocols
 - Read phase
 - Transaction is executed, but all writes are not committed
 - Write “private items”
 - Validation phase
 - If a transaction is ready to commit, check whether its execution has been correct
 - Write phase
 - Write private items to database

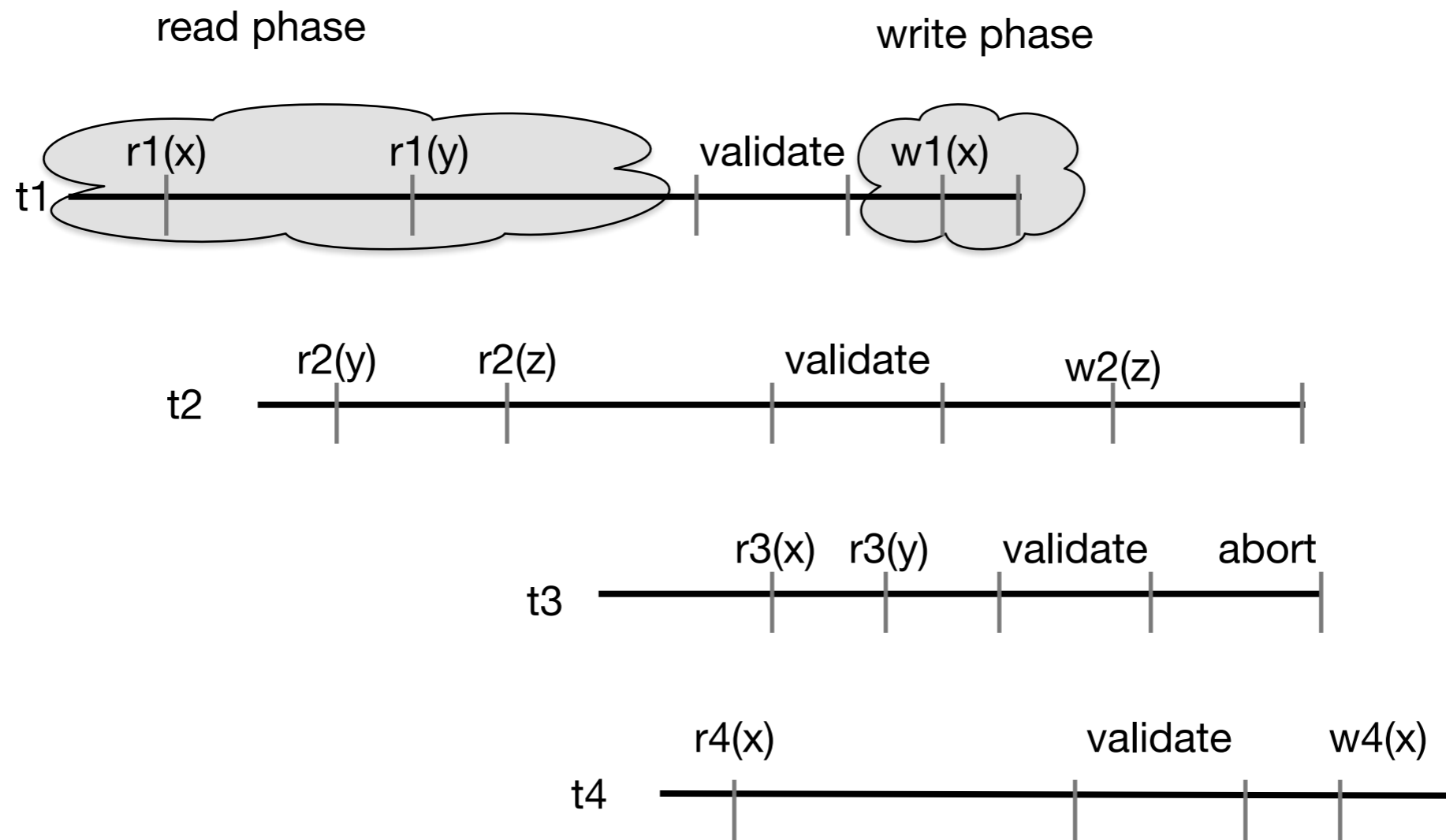
Non-Locking Algorithms

- Backward oriented optimistic concurrency control (BOCC)
 - Validate a transaction against those transactions already committed
- Forward oriented optimistic concurrency control (FOCC)
 - Validate a transaction against those transactions that are in their read phase

Non-Locking Algorithms

- BOCC:
 - The validate-write phase needs to be atomic
 - Transaction j is validated if for every committed transaction i :
 - i has ended before j started
 - or
 - The pages touched by j have not been written by i
 - Thus, j had no chance to read from i

Non-Locking Algorithms



t3 is aborted because its read set {x,y} overlaps with the write set {x} of t1

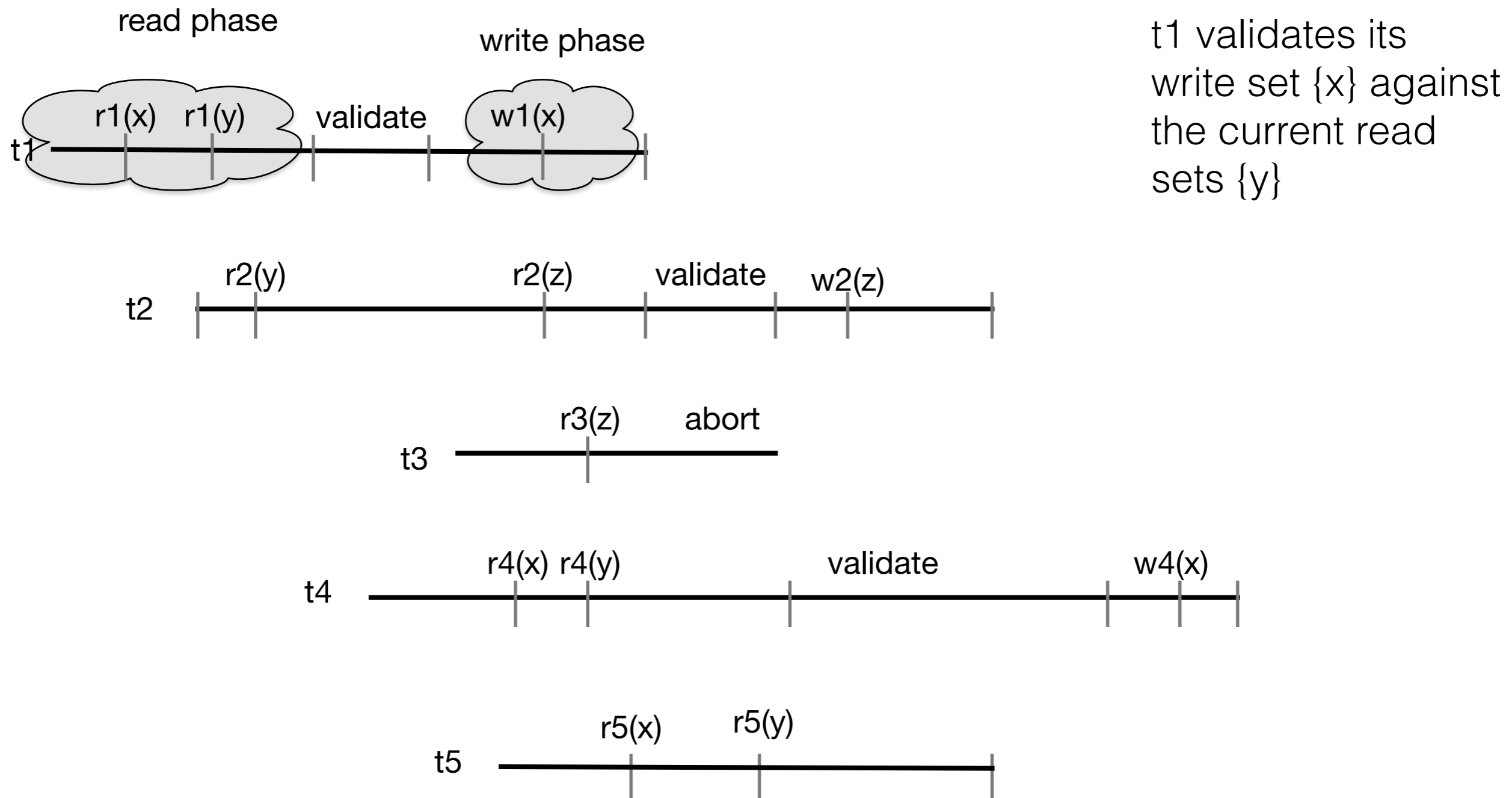
Non-Locking Algorithms

- Schedule
 - ... $r_2(x)$... $w_1(x)$... validate1 ... validate2 ...
 - t2 will get aborted as its read set overlaps with the write set of t1.
 - This is clear once t1 writes
- Therefore Forward-oriented optimistic Concurrency Control (FOCC)

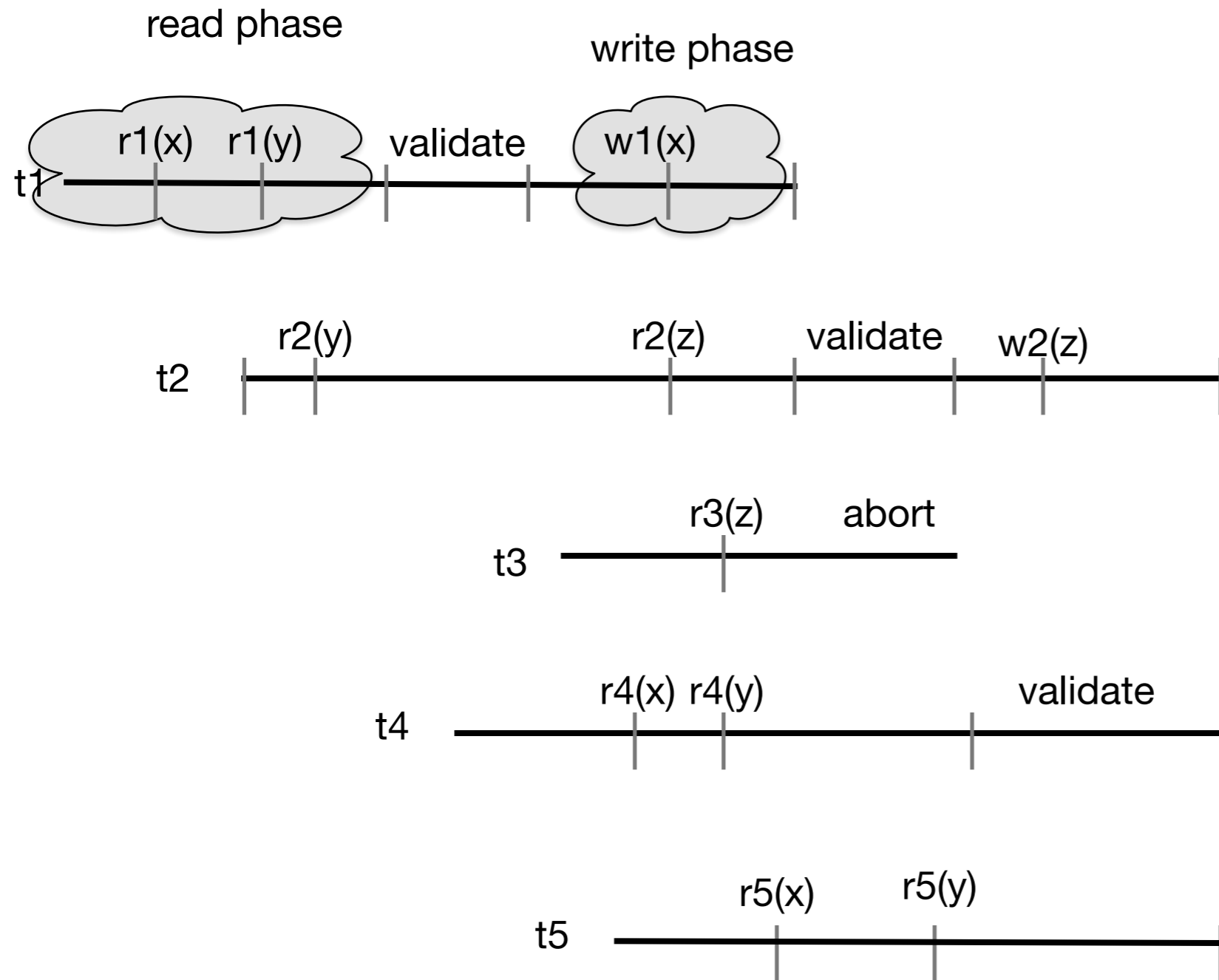
Non-Locking Algorithms

- Forward-oriented concurrency control
 - Accept a transaction t_j if for all transactions t_i that are currently reading
 - $Writeset(t_j)$ is disjoint from $Readset(t_i)$ at current time
- Example: FOCC accepts all read-only transactions

Non-Locking Algorithms



Non-Locking Algorithms

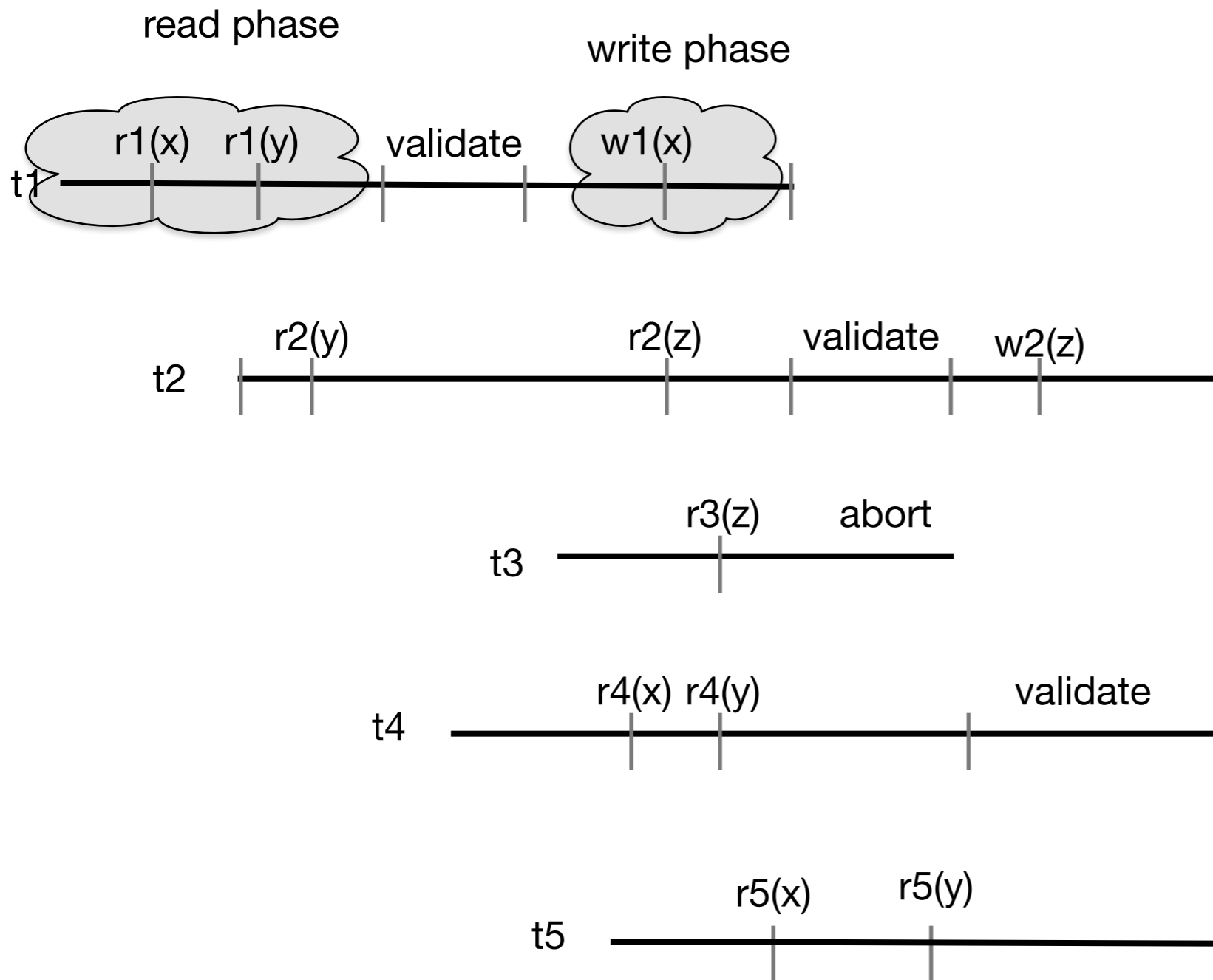


t2 validates its write set {z} against the current read sets {z, x, y} and discovers that they are not disjoint.

FOCC allows:
t2 could abort
t3 could abort

Chooses to abort t3

Non-Locking Algorithms



t4 validates its write set against current read sets {x, y, z} because of t5.

Instead of aborting, we can have t4 wait until t4 terminates

Then t4 validates

Multi-version Concurrency Control

- Allow a single value to have multiple versions
- Multi-version schedule
 - Note values read
 - Example
 - $r_1(x_0)w_1(x_1)r_2(x_1)w_2(y_2)r_1(y_0)w_1(z_1)c_1c_2$
 - Transaction 1 reads an earlier version than the value written by transaction 2

Multi-version Concurrency Control

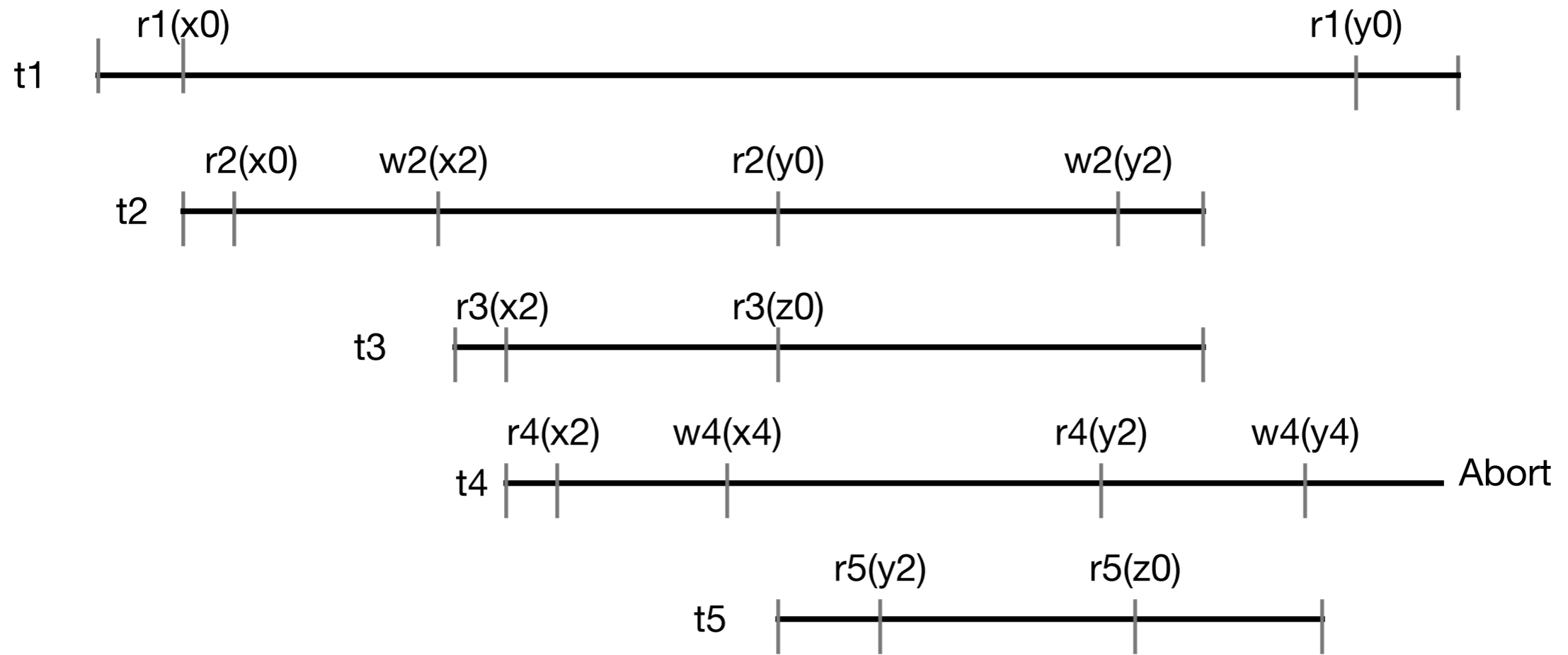
- Multi-version timestamp ordering (MVTO)
 - Each version carries the timestamp of the transaction that created it
 - Each read reads the last version that was written before the timestamp of the transaction
- Writes:
 - $w_i(x)$
 - If there was a read $r_j(x_k)$ with
 - $\text{time}(t_k) < \text{time}(t_i) < \text{time}(t_j)$
 - abort t_i
 - Otherwise, write x with timestamp $ts(t_k)$

Multi-version Concurrency Control

- In order to avoid dirty reads:
 - Delay commits until all other transactions that have written a new version of what we read have finished

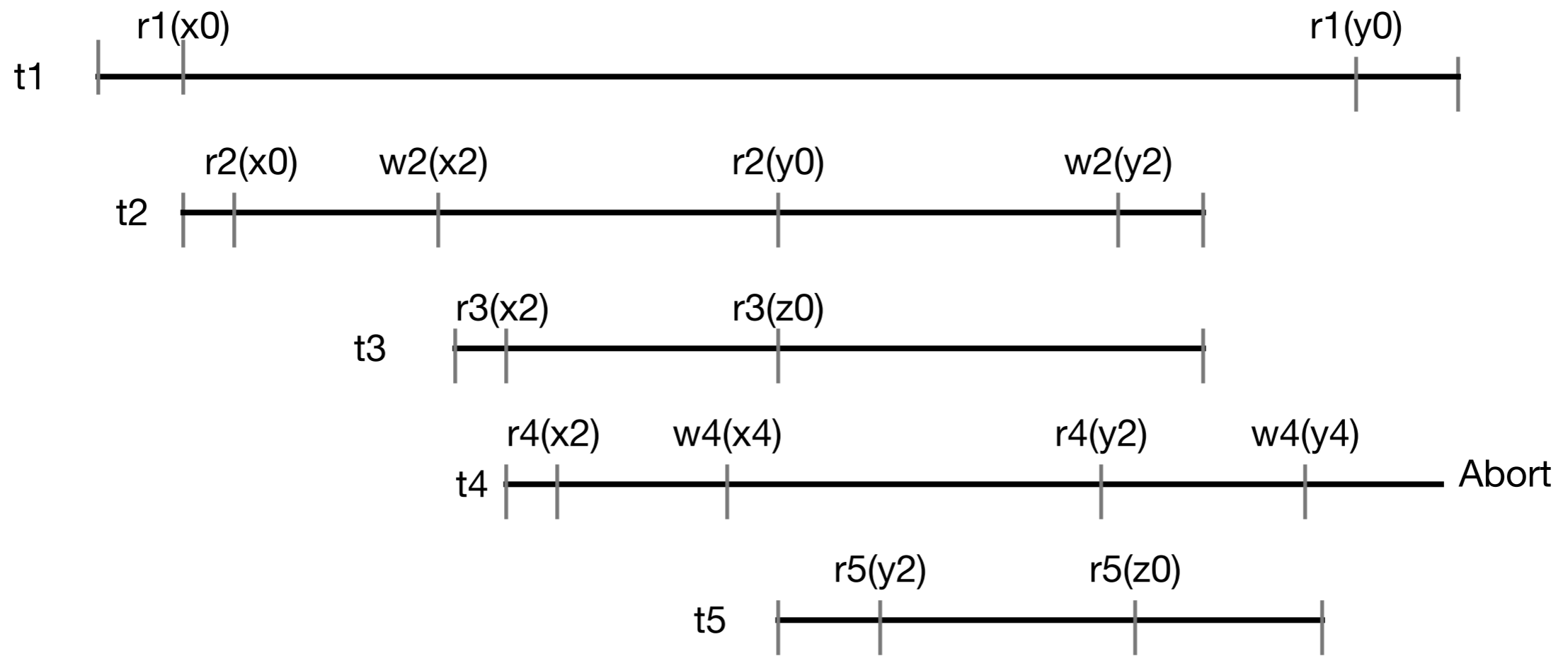
Multi-version Concurrency Control

t1 and t2 are interleaved



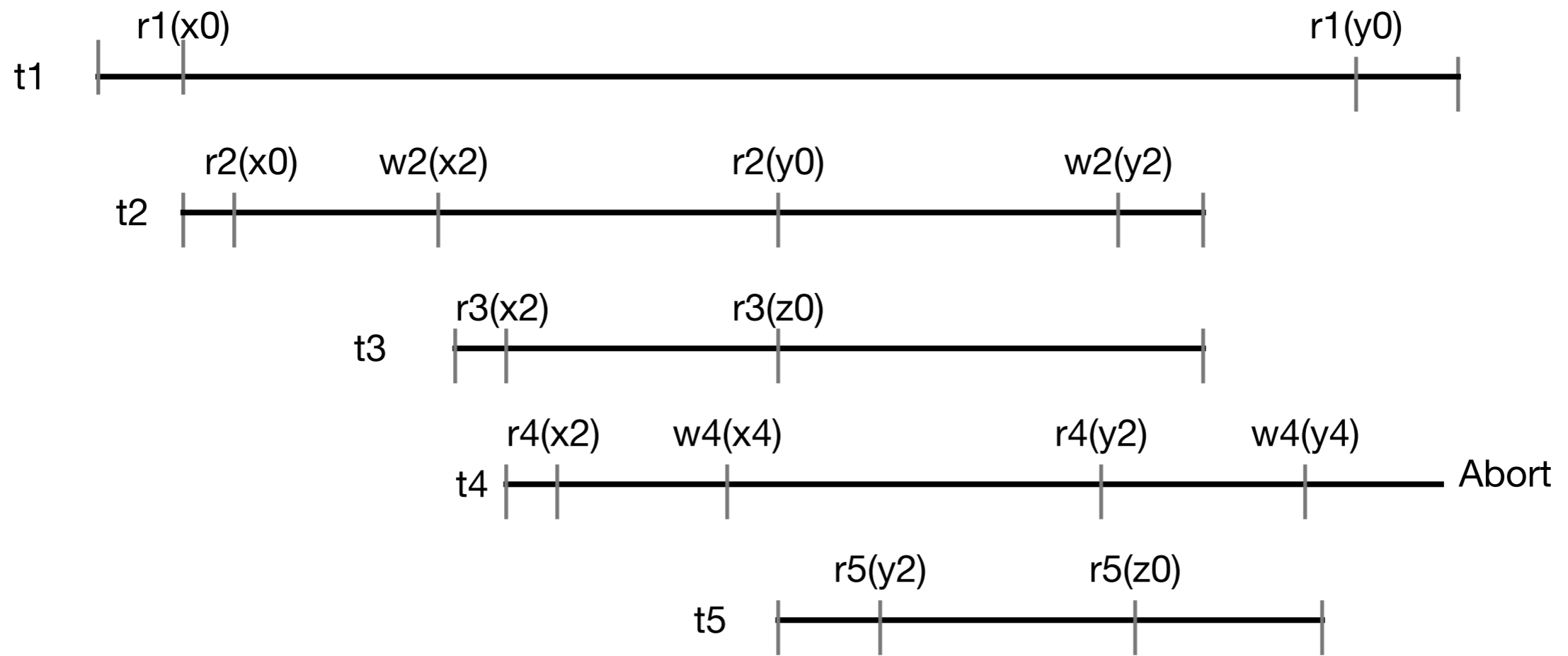
Multi-version Concurrency Control

t3 needs to wait until t2 terminates, because it read x2
Since its timestamp is larger, it has to read this value instead of x0



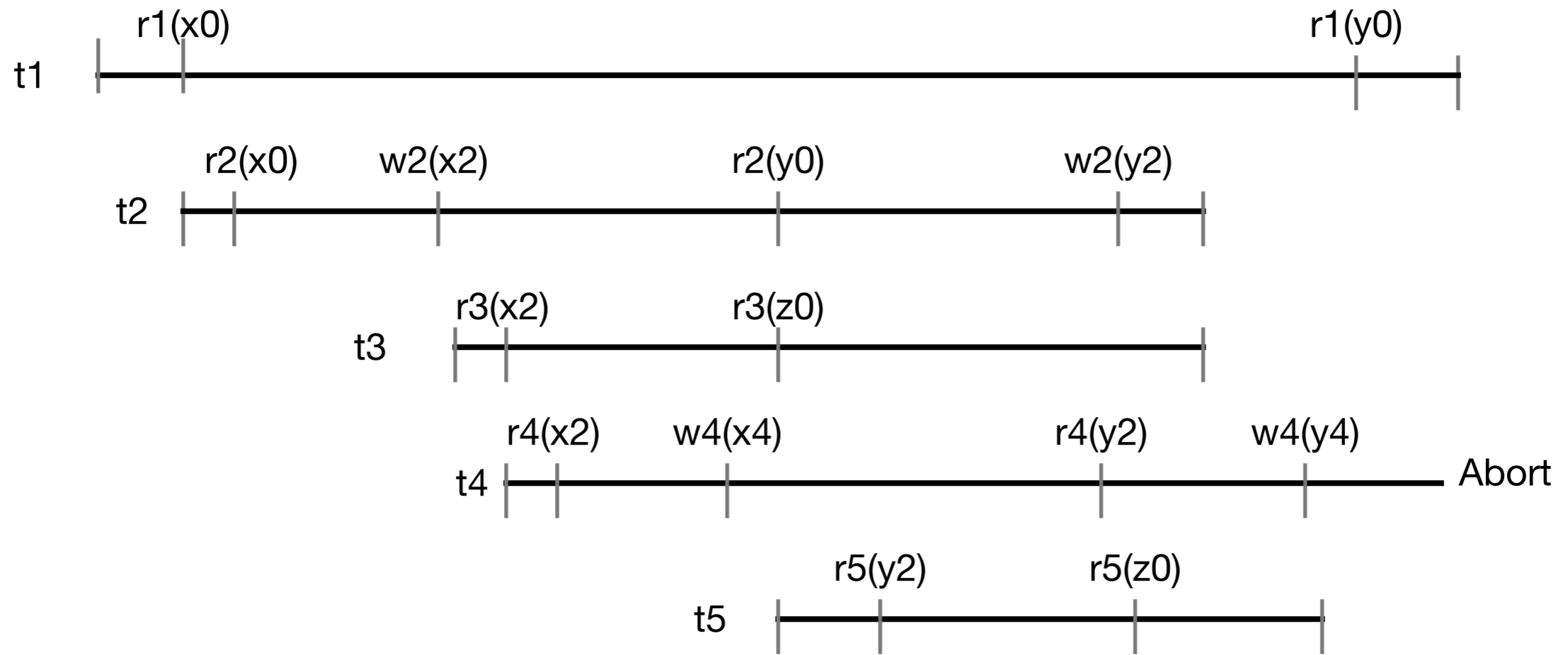
Multi-version Concurrency Control

t4 is a “late writer”. t5 (with a later timestamp) has already read y2, and t4 can no longer change it. So, t4 needs to be aborted.



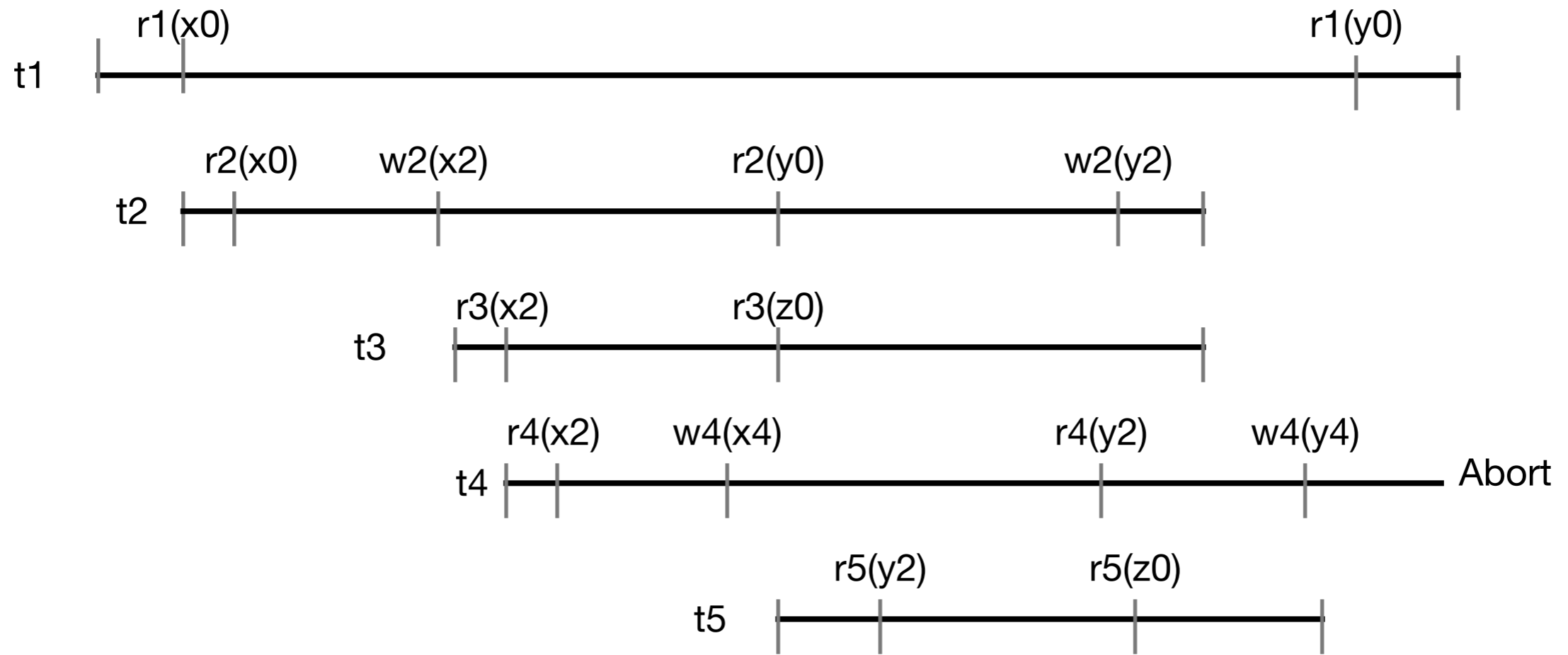
Multi-version Concurrency Control

t1 and t2 are interleaved



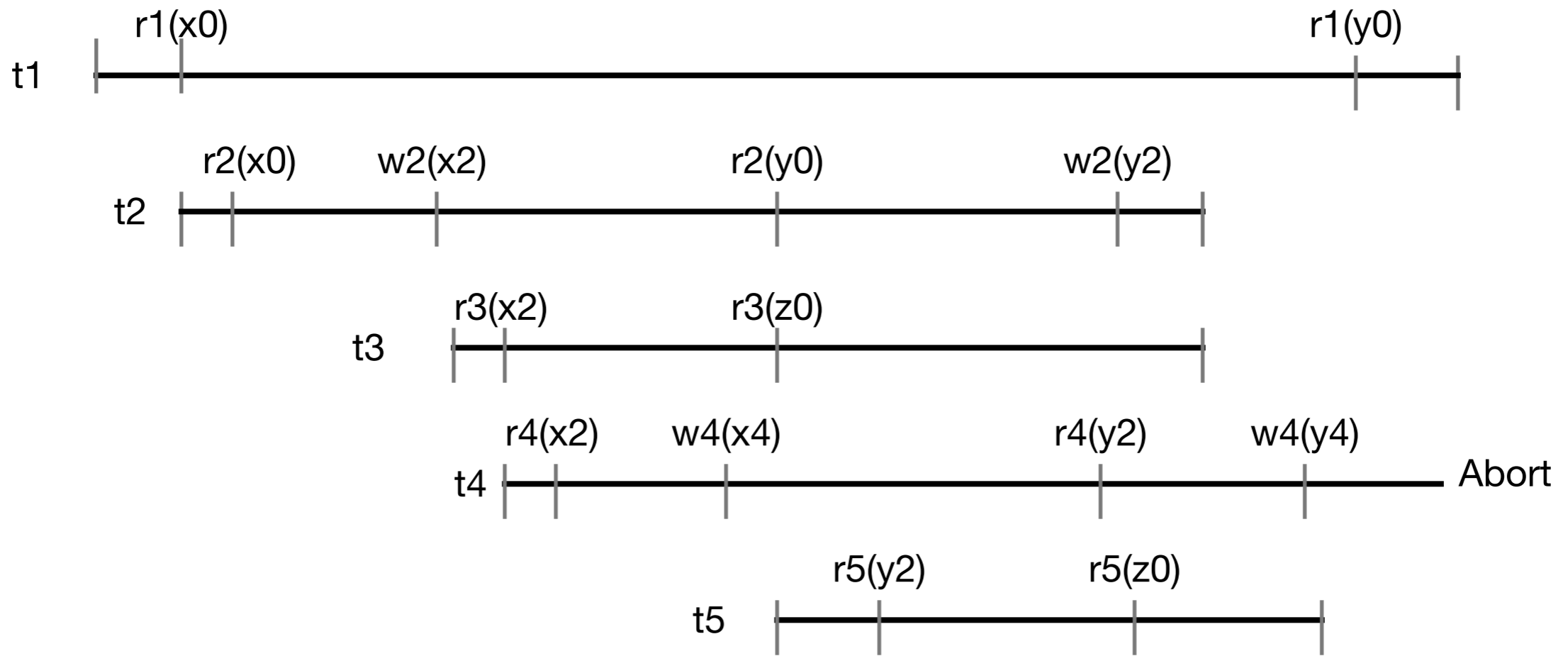
Multi-version Concurrency Control

t1 and t2 are interleaved



Multi-version Concurrency Control

t1 and t2 are interleaved



Write-Ahead Logging

- A family of protocols that use a log
 - Recovery after a crash
 - Aborting transactions
- Each site writes the operation it is about to perform on a page to the write-ahead log

```
x = 0;  
y = 0;  
BEGIN_TRANSACTION;  
  x = x + 1;  
  y = y + 2;  
  x = y * y;  
END_TRANSACTION;
```

(a)

Log

[x = 0/1]

(b)

Log

[x = 0/1]

[y = 0/2]

(c)

Log

[x = 0/1]

[y = 0/2]

[x = 1/4]

(d)

Write-Ahead Logging

- ARIES
 - Write-ahead log
 - Repeating history
 - After crash, retrace the actions to bring database up to the moment of crash
 - Undo transactions that were then pending
 - Logging Undo operations
 - Log undo operations in order to avoid repeating actions after repeated crashes

Write-Ahead Logging

- ARIES
 - Dirty Page Table (DPT)
 - Transaction Table (TT)
 - Log
 - Sequence Number, Transaction ID, Page ID, Redo, Undo, Previous Sequence Number
 - Redo and Undo: Information to redo and undo a transaction

Write-Ahead Logging

- ARIES
 - Analysis
 - Calculate the necessary information from the log
 - From last checkpoint:
 - Add all transactions started to TT
 - Remove transactions in the TT when finding a END LOG statement
 - Update the dirty pages table

Write-Ahead Logging

- ARIES
 - Redo
 - Use the DPT to calculate the minimal sequence number of a dirty page
 - From this sequence number, redo operations for pages in the DPT

Write-Ahead Logging

- ARIES
 - Undo
 - Undo the changes of uncommitted transactions
 - Run backward through the log
 - For each transaction in the TT
 - Undo the change for each touched page
 - Write the changes in a compensation log
 - (In case of a crash during recovery)

Distributed Transactions

- Issues:
 - Distributed decision making
 - by leader: Leader election
 - Commit protocols
 - Distributed locks:
 - Deadlock detection
 - Deadlock avoidance
 - Distributed checkpoints

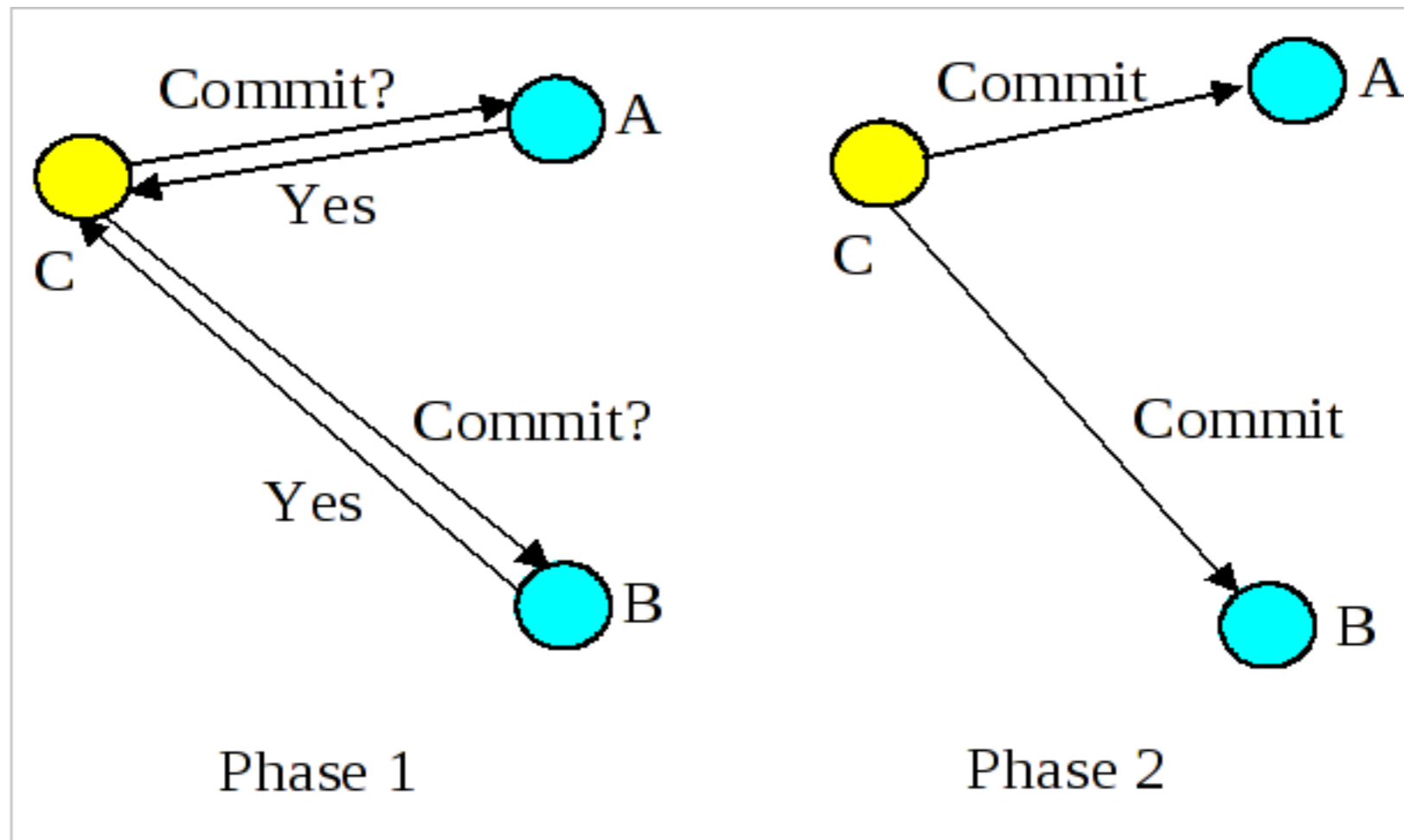
Distributed Transactions

- Distributed commit
 - All members of a group need to perform an action or none
- One phase commit:
 - Single coordinator
 - Sends a “commit” or a “not-commit” message
 - Has no feedback from participants
 - Cannot be used in practice

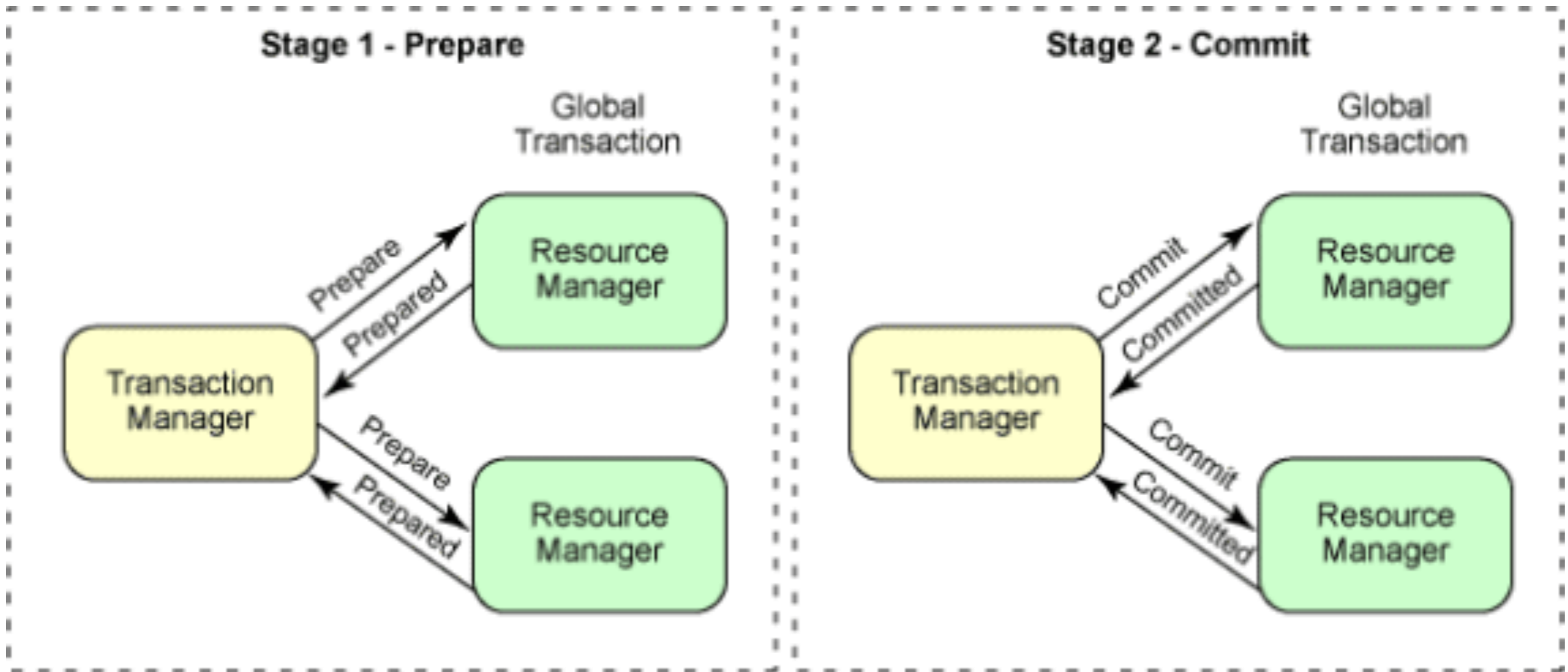
Distributed Transactions

- Two phase commit (Jim Gray)
 - Phase 1:
 - Coordinator sends vote request
 - Participants vote on whether they want to commit
 - Phase 2:
 - Coordinator decides vote
 - Single no is a veto
 - Coordinator sends participants message

Distributed Transactions



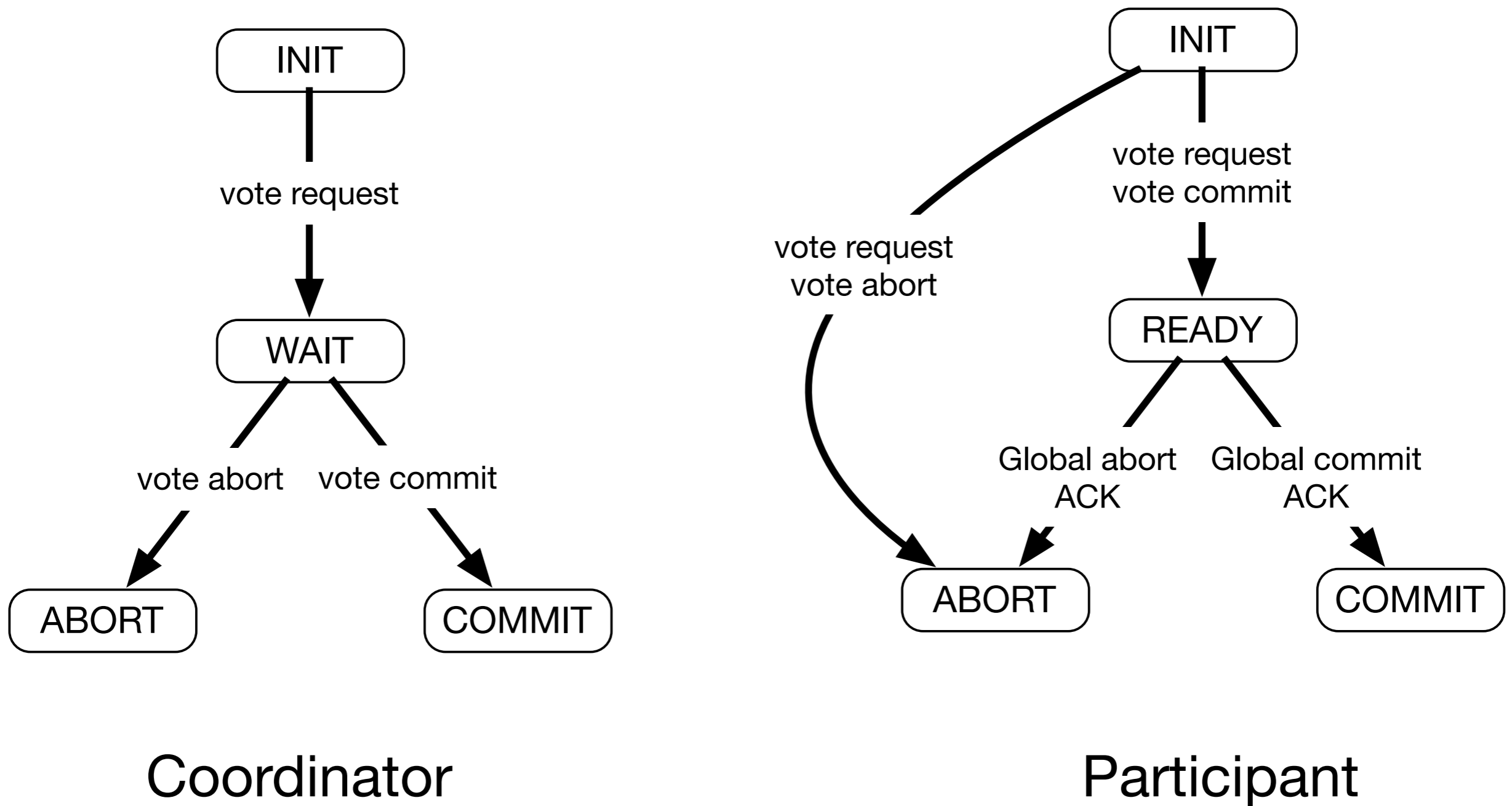
Distributed Transactions



Distributed Transactions

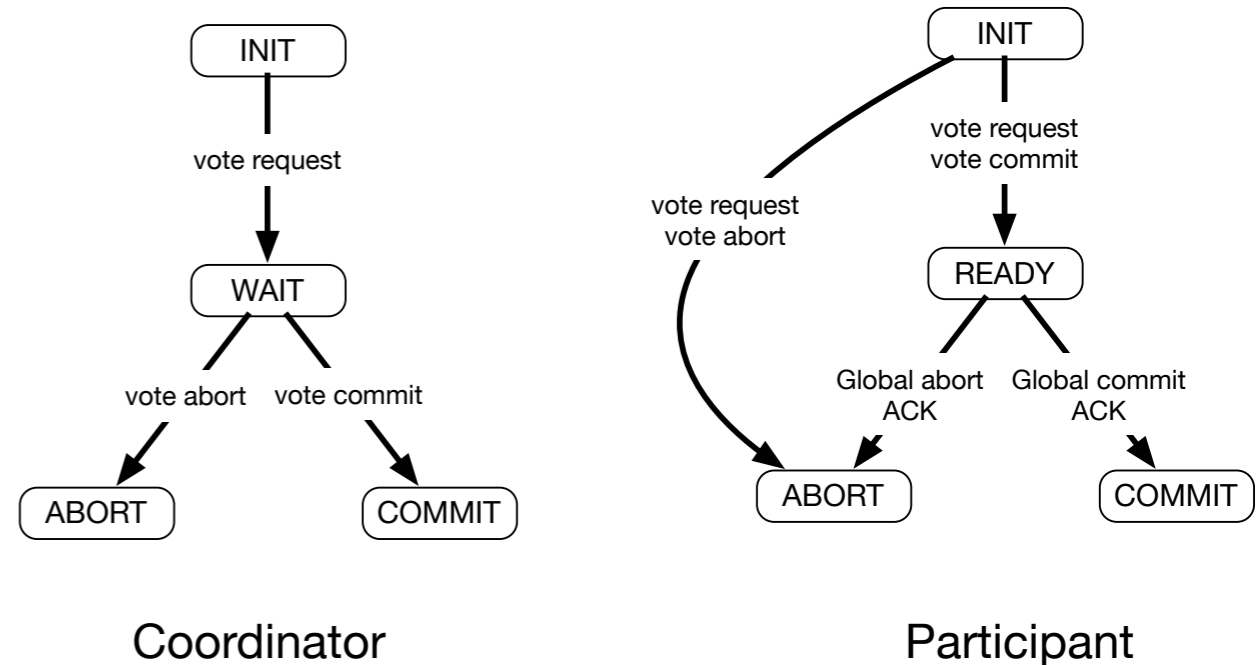
- 2PC can have problems with failures
 - Failure of coordinator or participants leads to blocking

Distributed Transactions



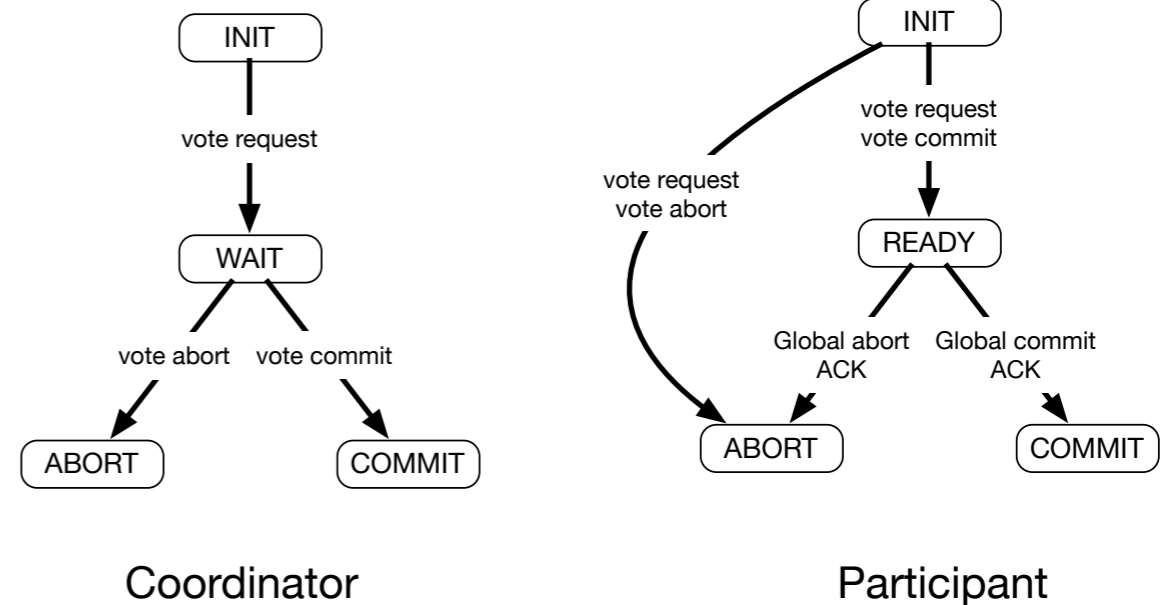
Distributed Transactions

- Participant in INIT waiting for request to vote
 - Can locally abort transaction



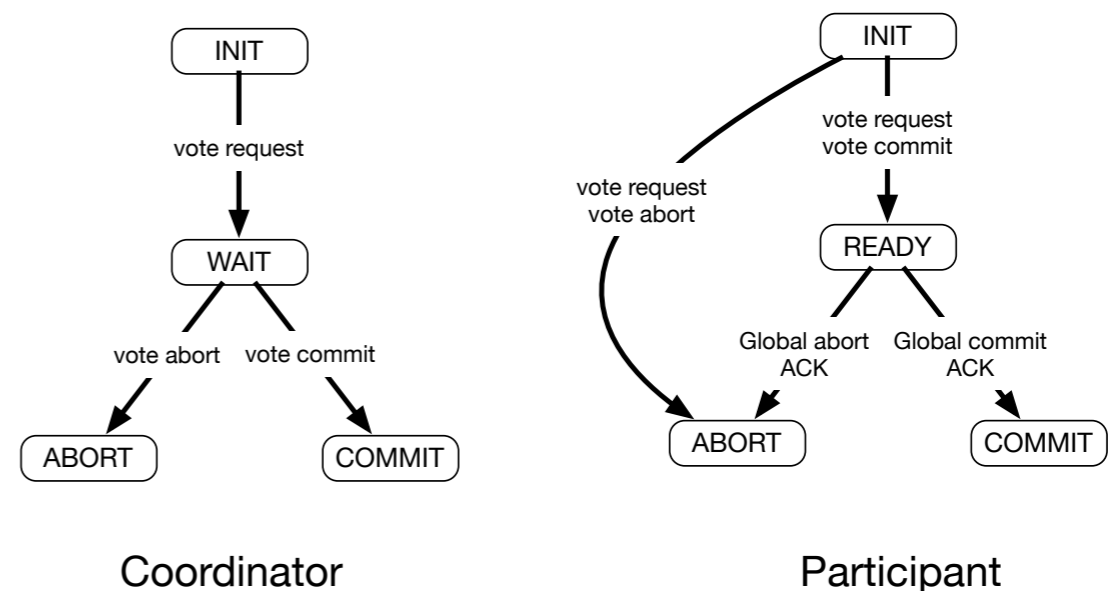
Distributed Transactions

- Coordinator in WAIT waiting for answers
- Can send Global_Abort



Distributed Transactions

- Participant in READY waiting for coordinator
- Cannot decide
 - Block until coordinator recovers
 - Or talk to other participant
- If all participants are in state READY, no decision can be taken



Distributed Transactions

- Participant in READY waiting for coordinator
- Cannot decide
 - Block until coordinator recovers
 - Or talk to other participant
- If all participants are in state READY, no decision can be taken

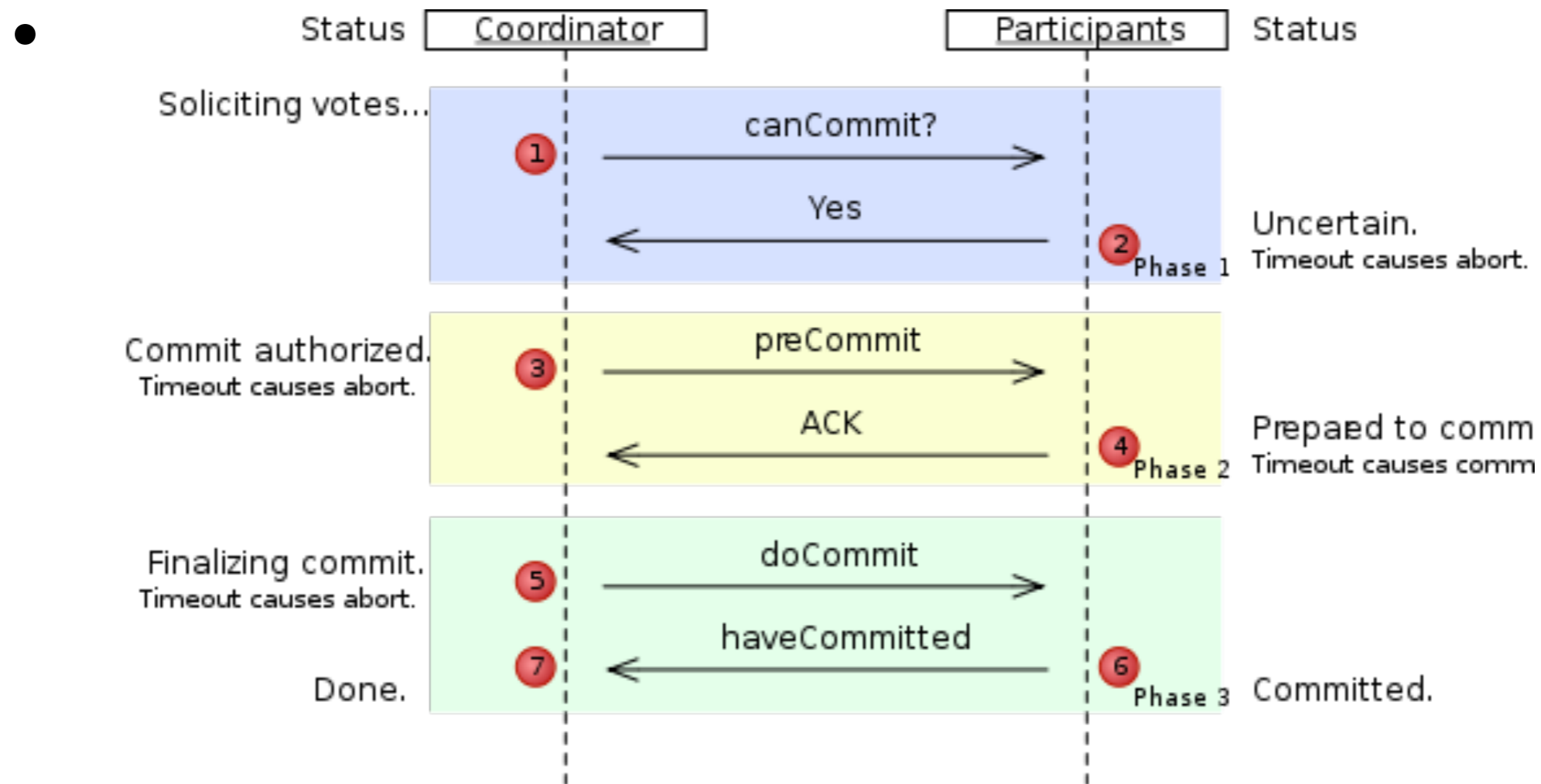
State of Q	Action by P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

Distributed Transactions

- Non-blocking solution
 - Use multicast primitive
 - Receiver immediately multicasts received message to all participants

Distributed Transactions

- Three Phase Commit
 - Crashed coordinator might leave participants hanging
Solution is to add another phase



Distributed Time Order

- TO rule:
 - If $p_i(x)$ and $q_j(x)$ are operations in conflict then
 $p_i(x)$ is executed before $q_j(x) \iff ts(t_i) < ts(t_j)$
 - Use Lamport clock to generate time stamps

Server 1 :	$r_1(x)$	$w_2(x)$...
Server 2 :	$r_2(y)$	$w_1(y)$...
- Both transactions have local timestamp 1 but are ordered so that 2nd transaction aborts

Distributed Transactions

- Locking protocols
 - Need to reach global decision when to release a lock
 - Primary 2PL:
 - All locking is done at a *primary* site
 - Distributed 2PL
 - Strict 2PL with commit releases locks

Distributed Transactions

- Optimistic protocols
 - Protect validation / write phase by using 2PC or 3PC

Distributed Transactions

- Distributed deadlock handling
 - Detection
 - Time-outs
 - Edge chasing
 - Blocked transaction sends out a probe to all transactions it is waiting for
 - Those forward probe
 - When probe returns to sending transaction:
 - Deadlock exist
 - Path(s) of returned probe(s) indicates which transaction to abort
 - Path pushing
 - Collect local “waits-for” graphs at a single server

Ticket-Based Concurrency

- Used in federated databases
 - Need to “force” conflicts between competing local transactions.
 - Example: Database $D1=\{a,b\}$, $D2=\{c,d\}$
 - Global transactions $t1=r(a)r(c)$, $t2=r(b)r(d)$
 - Local transactions $t3=w(a)w(b)$, $t4=t(c)t(d)$

$$s_1 = r_1(a)c_1w_3(a)w_3(b)c_3r_2(b)c_2$$

$$s_2 = w_4(c)r_1(c)c_1r_2(d)c_2w_4(d)c_4$$

- is incorrect

$$s_1 \approx t_1t_3t_2, \quad s_2 \approx t_2t_4t_1$$

Ticket-Based Concurrency

- Each local database maintains a ticket
 - Accessed only by global transactions
 - Operations are:
 - Read ticket
 - Take-a-ticket:
 - Read ticket, write back incremented ticket value
- Each transaction takes a ticket at a local database
- Can commit only if ticket values have the same relative order at all participating sites

Ticket-Based Concurrency

- Optimistic ticket method

- Transaction manager maintains a ticket graph

$$t_i \rightarrow t_j \iff$$

transaction i reads a ticket value smaller than transaction j

- Before commit, make sure that ticket graph has no cycles

Ticket-Based Concurrency

- Conservative ticket method
 - Insures that transactions can only get tickets in the same order