Distributed Transactions

Presenter: Yoongu Kim

CMU 15-799 (Andy Pavlo)

9/16/2013

Today's Papers

1. ***Concurrency Control in Distributed Database Systems (1981).***
   Bernstein & Goodman

2. ***Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment (1993).***
   Samaras, Britton, Citron, Mohan

Executive Summary (*Bernstein et al.*)

1. **Problem**: When multiple users issue concurrent transactions to a distributed database, they experience interference

2. **Solution**: **Concurrency Control**
   - Resolves interference to preserve correctness
   - Provides the illusion that each transaction is running on a dedicated database system

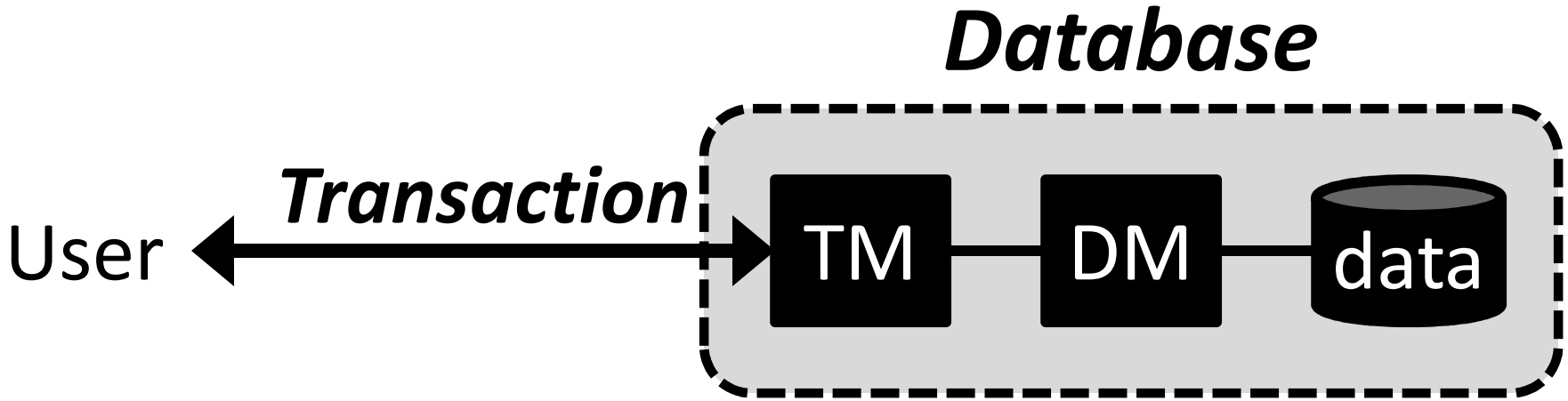
Executive Summary (*Bernstein et al.*)

3. **Goal**: Survey the state-of-the-art in concurrency control algorithms using a standardized set of terminology and assumptions
   - Algorithm #1: **Two-Phase Locking**
   - Algorithm #2: **Timestamped Ordering**

Outline

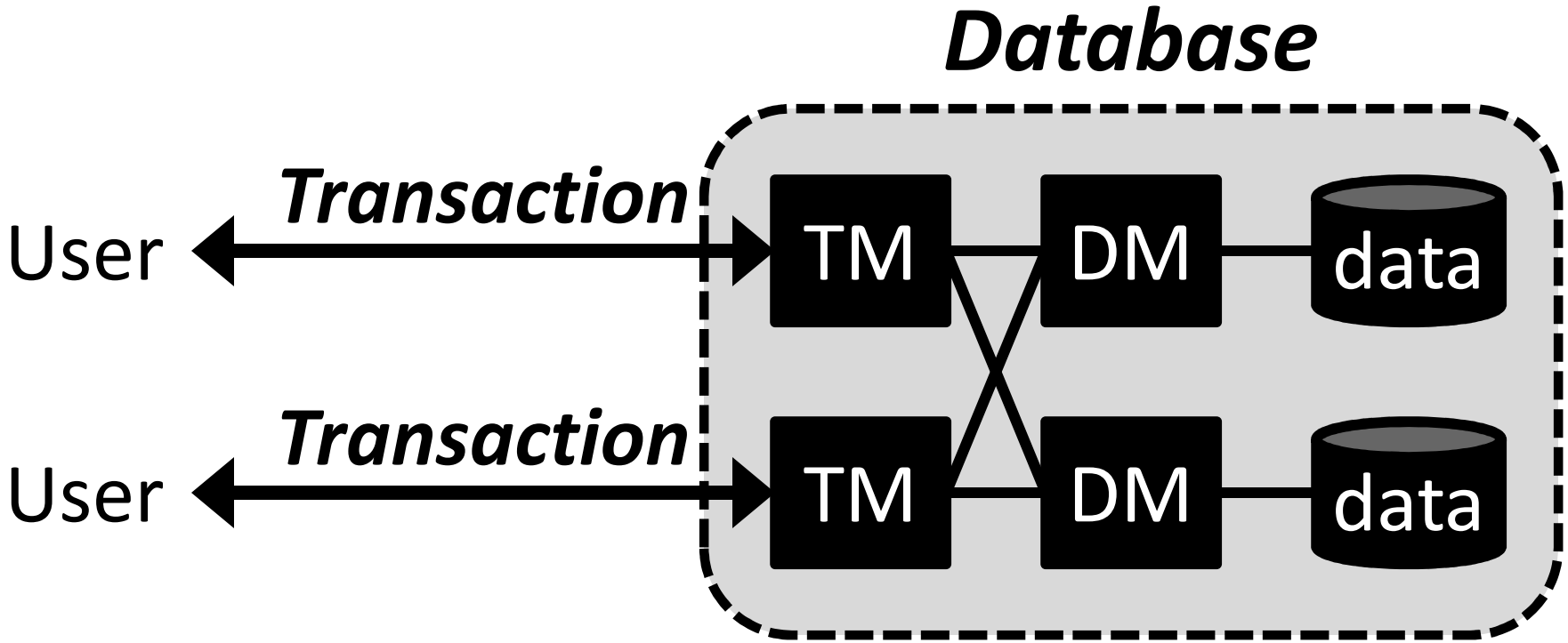
- **Distributed Database Systems**
- Correctness: Serializability
- Two-Phase Locking
- Timestamp Ordering

Centralized Databases



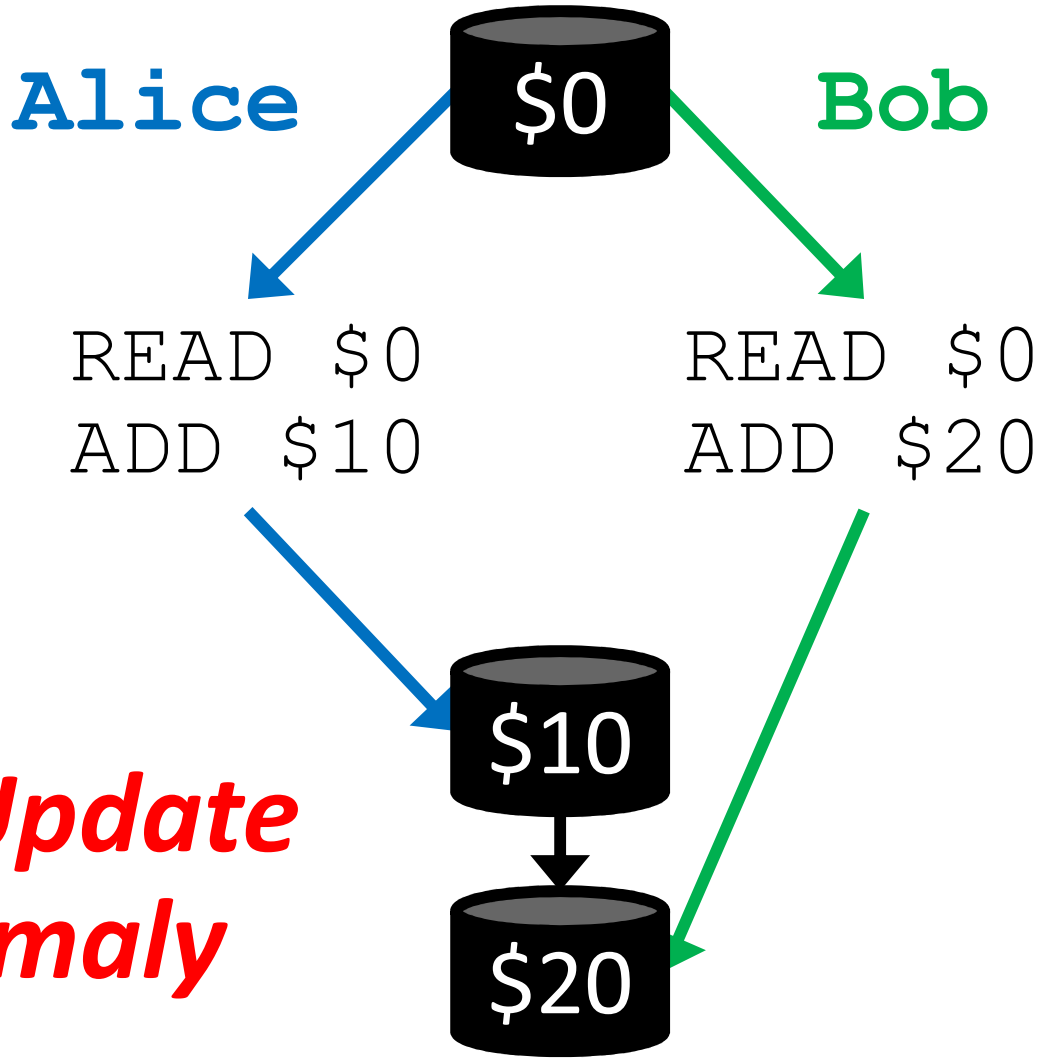
- ***Transaction Manager (TM)***
  - supervises interactions between users
- ***Data Manager (DM)***
  - supervises the actual database

Distributed Databases



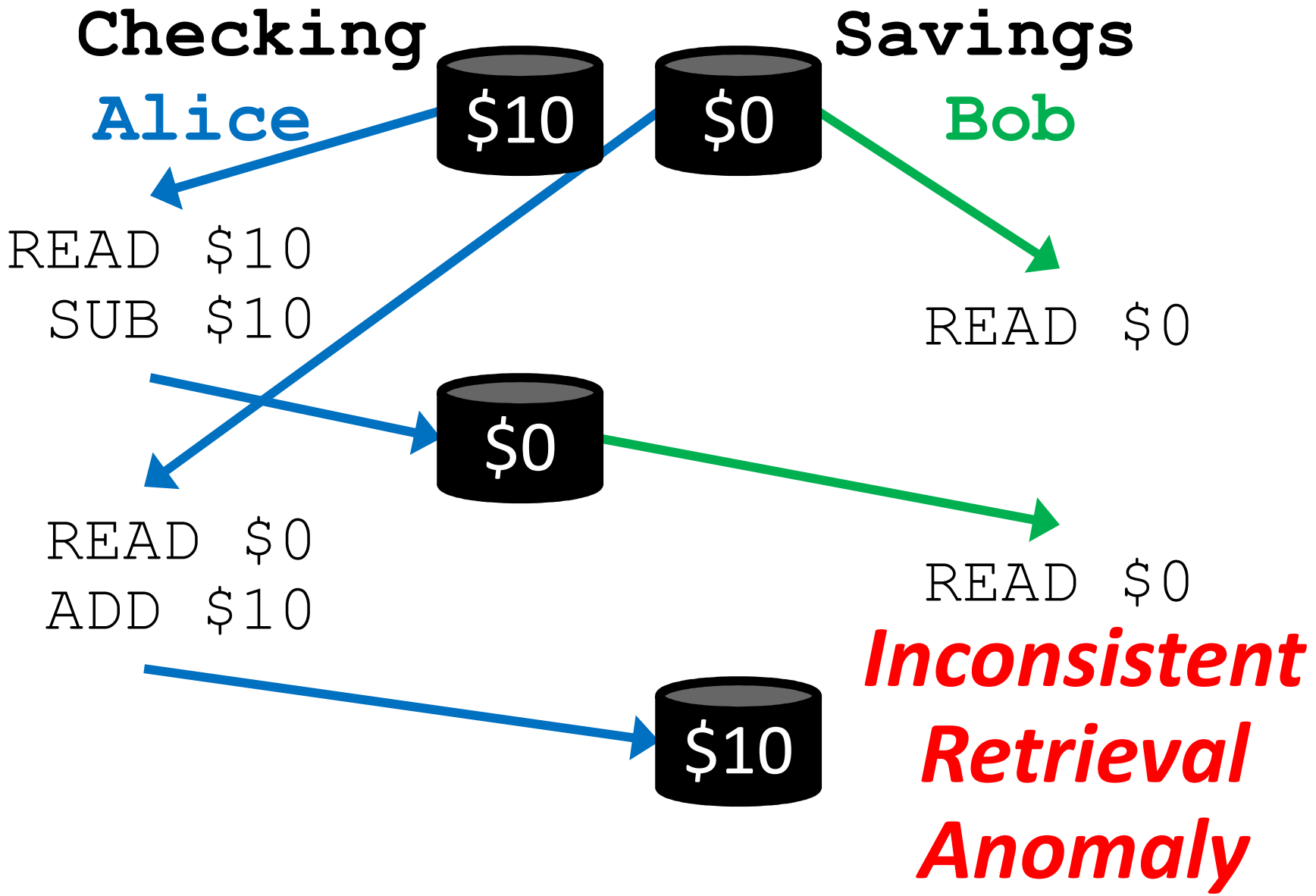
- Parallelism is a double-edged sword
  - Pro: higher performance
  - Con: interference between users

What can go wrong?



Lost Update Anomaly

What can go wrong? (cont'd)



Four Desirable Properties

- **Atomicity**: a transaction either commits in its entirety or does not commit at all
- **Consistency**: a transaction does not leave the database in an illegal state
- **Isolation**: transactions do not interfere with each other

Concurrency Control

- **Durability**: a committed transaction stays committed

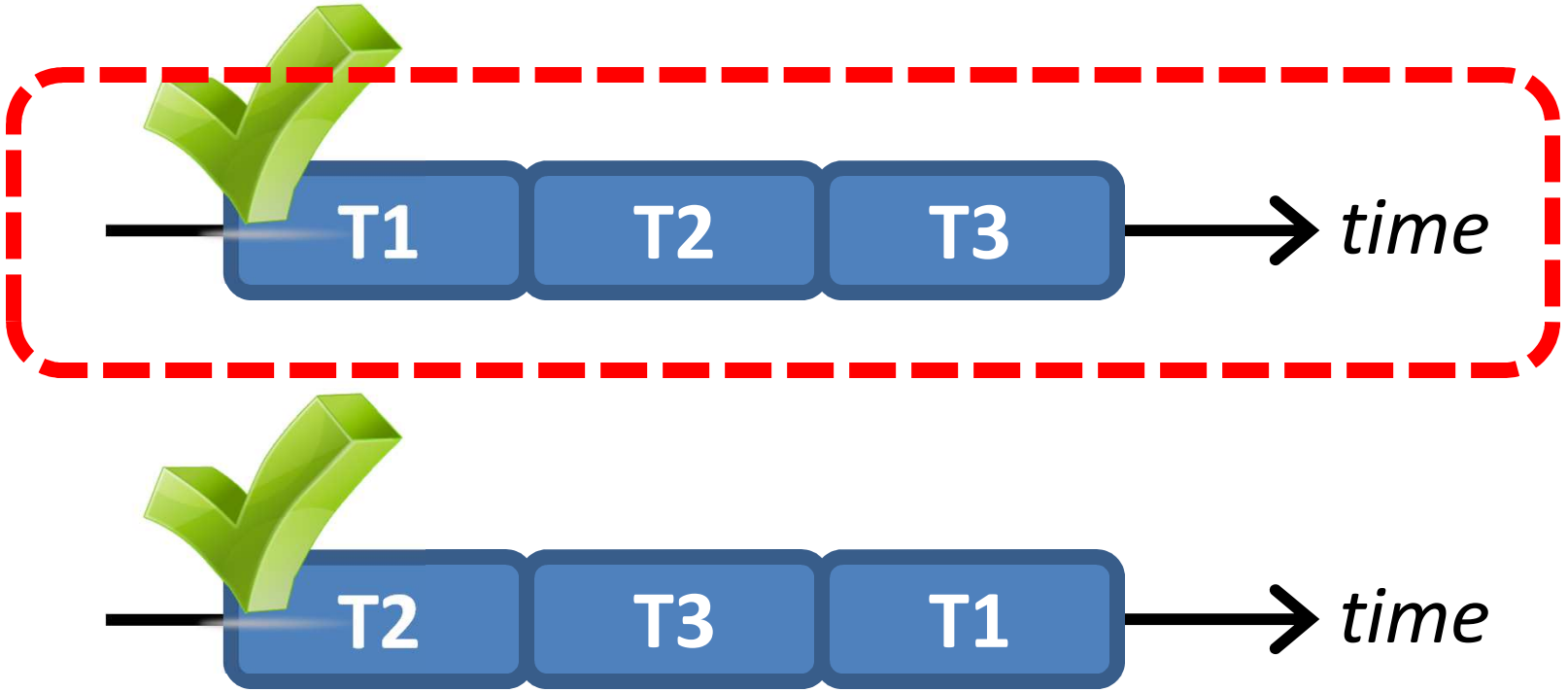
*Referred to as **ACID** properties*

Outline

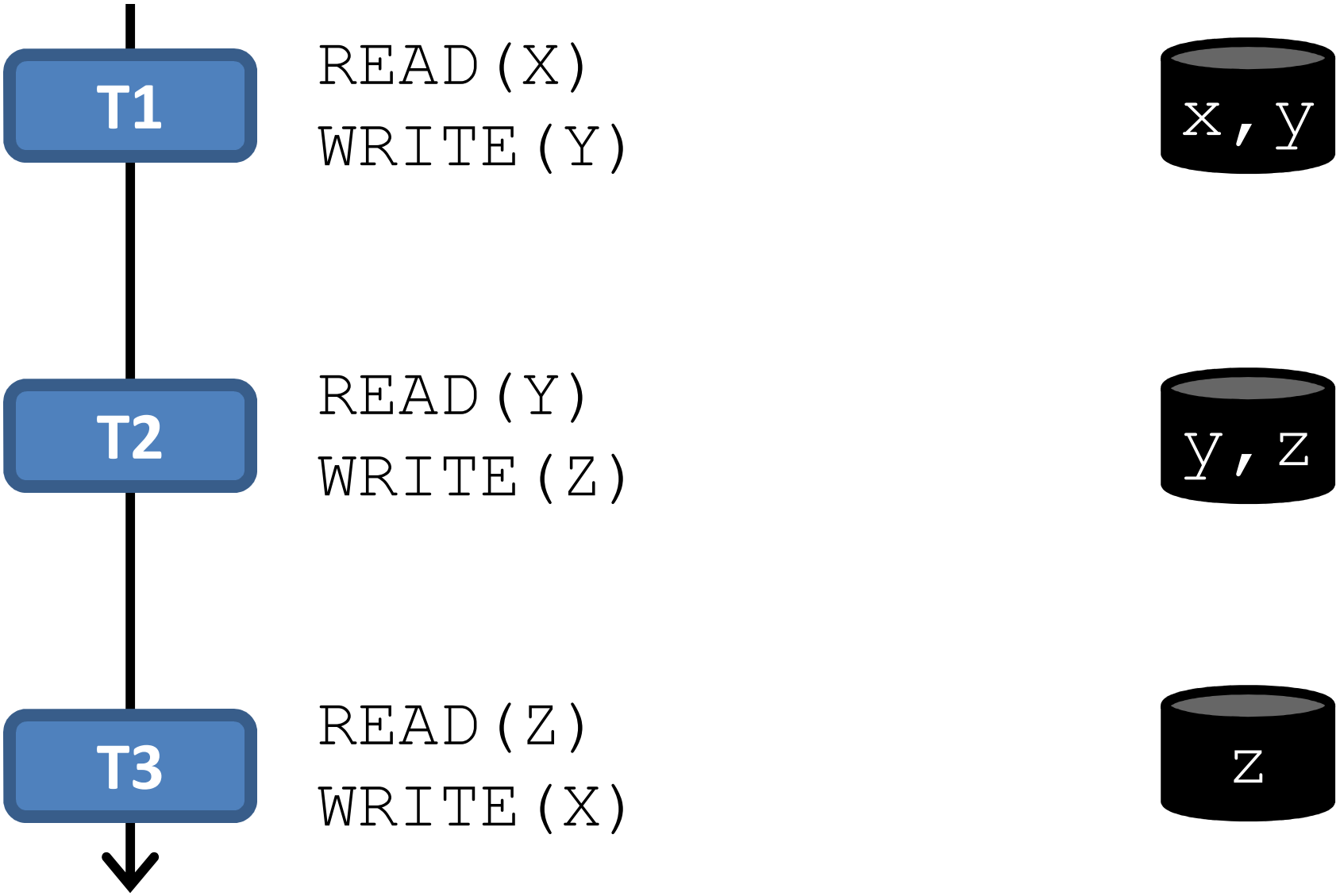
- Distributed Database Systems
- **Correctness: Serializability**
- Two-Phase Locking
- Timestamp Ordering

Serial Execution of Transactions

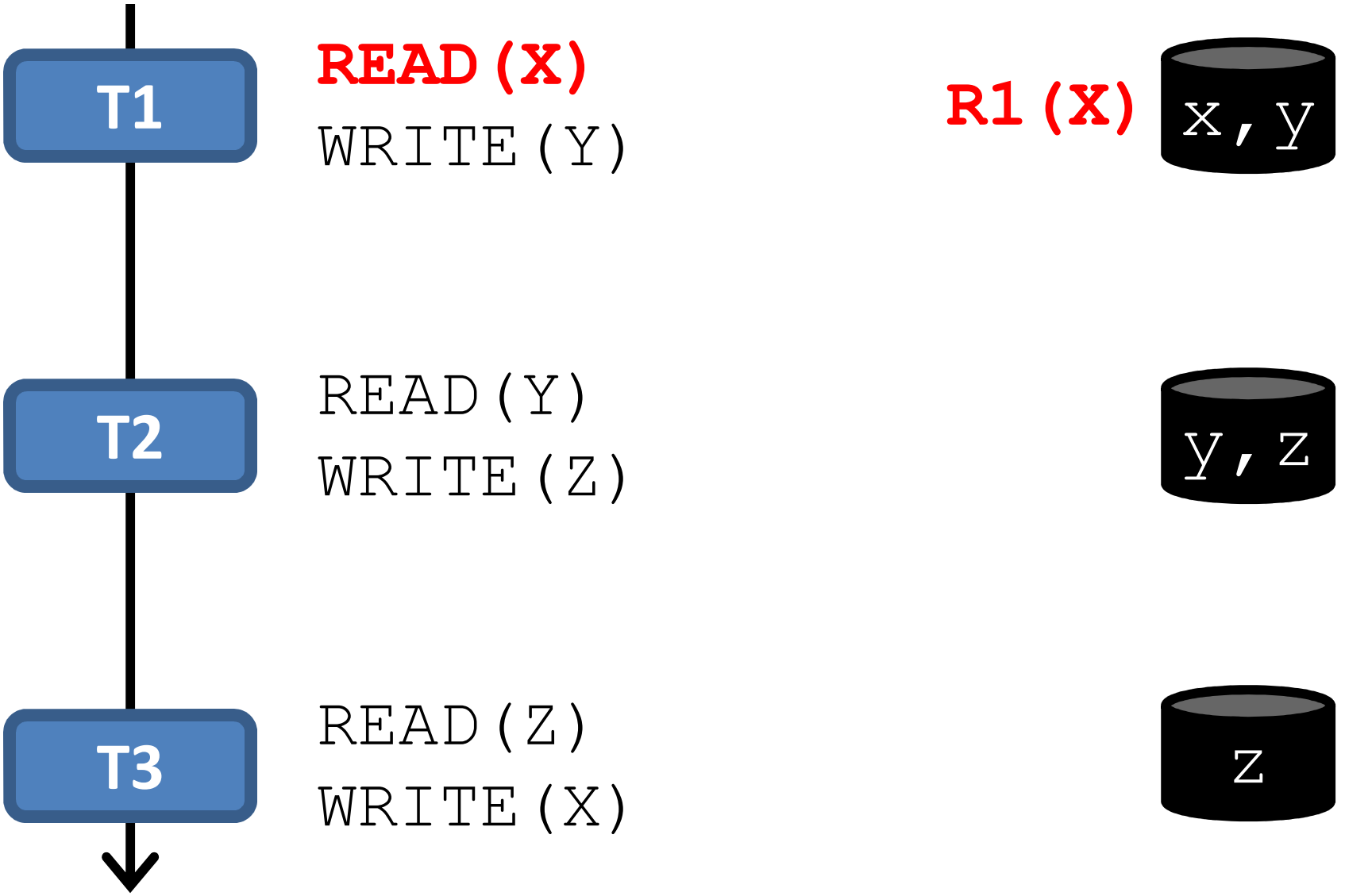
Serial Execution: When each transaction is executed to completion before the next one starts



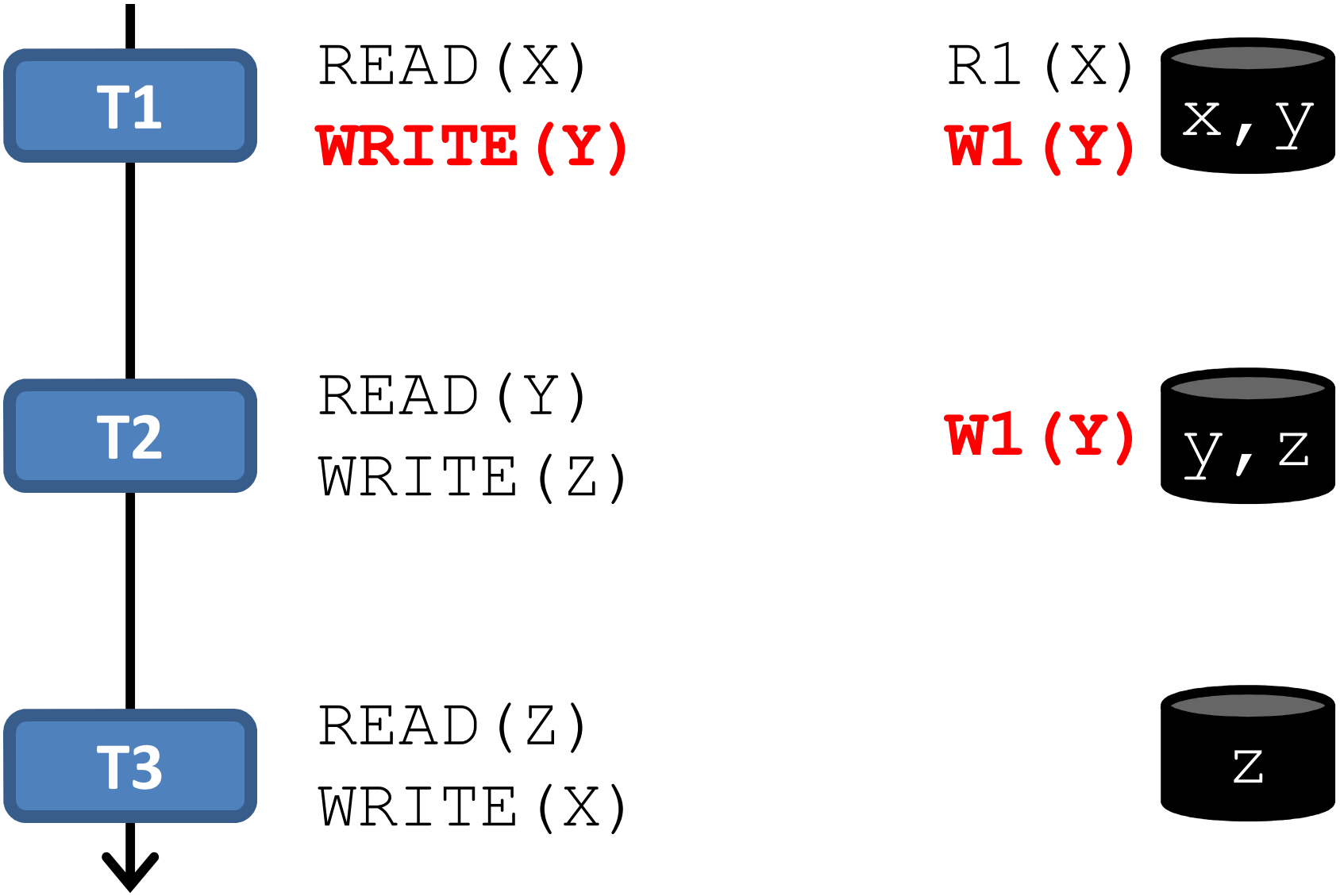
A Closer Look at Serial Execution



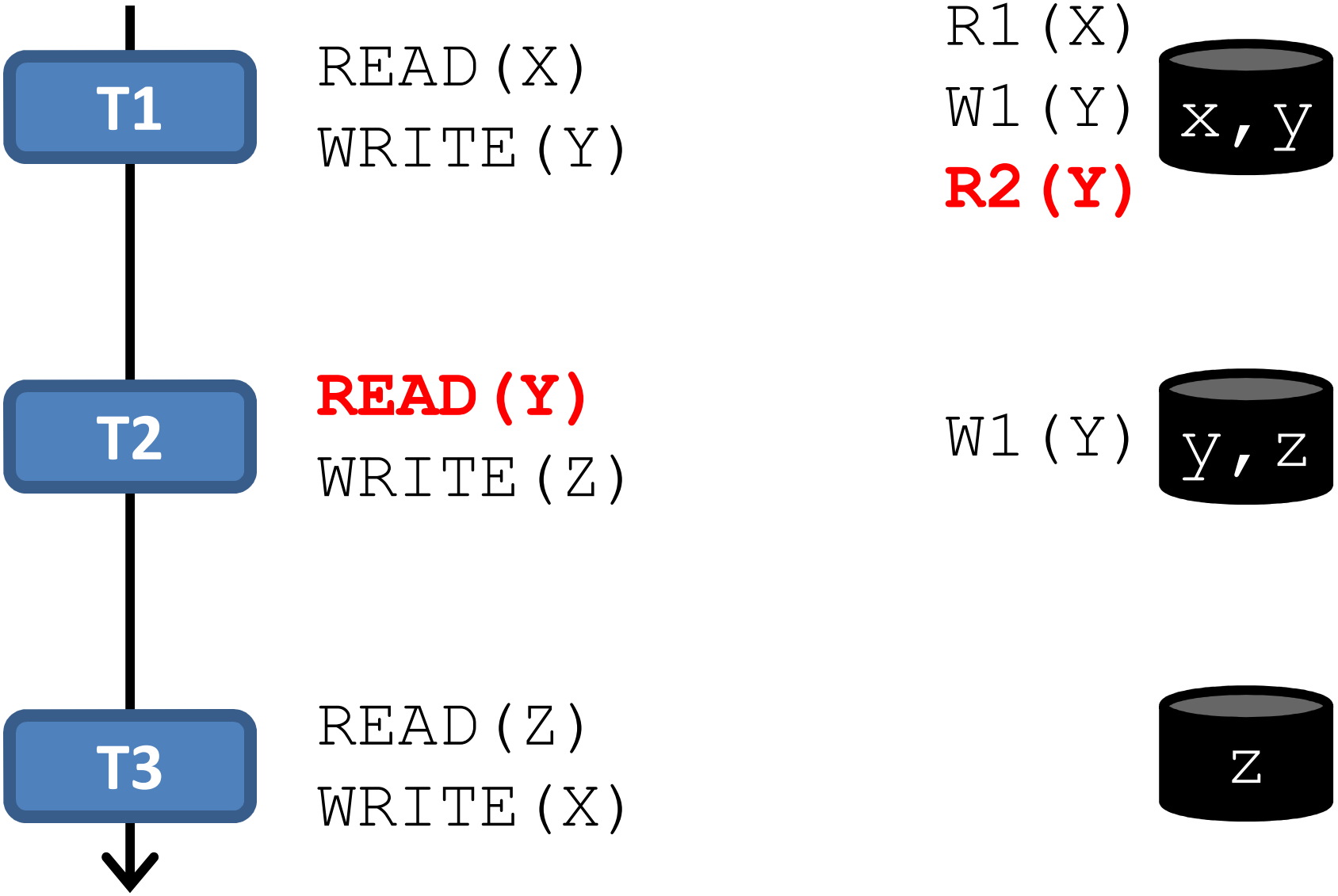
A Closer Look at Serial Execution



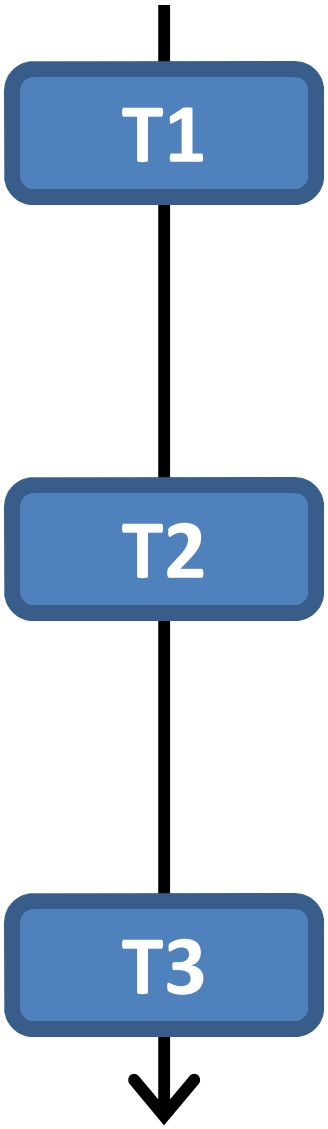
A Closer Look at Serial Execution



A Closer Look at Serial Execution



A Closer Look at Serial Execution

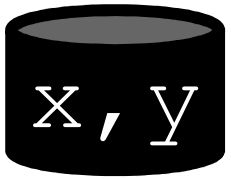


READ(X)
WRITE(Y)

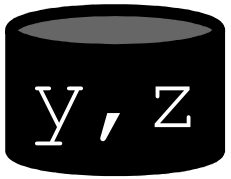
READ(Y)
WRITE(Z)

READ(Z)
WRITE(X)

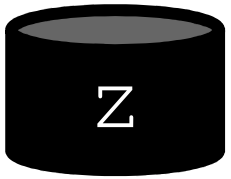
R1(X)
W1(Y)
R2(Y)



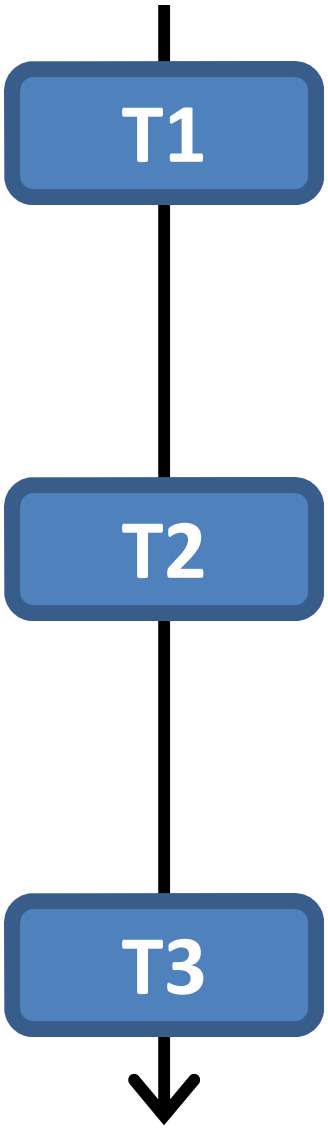
W1(Y)
W2(Z)



W2(Z)

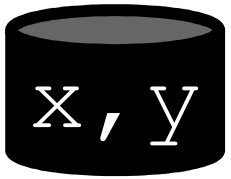


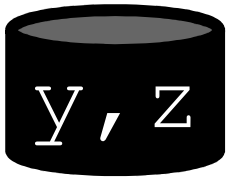
A Closer Look at Serial Execution

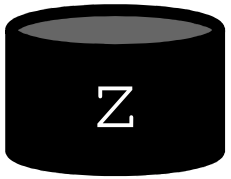



```
READ(X)
WRITE(Y)

READ(Y)
WRITE(Z)

READ(Z)
WRITE(X)
```

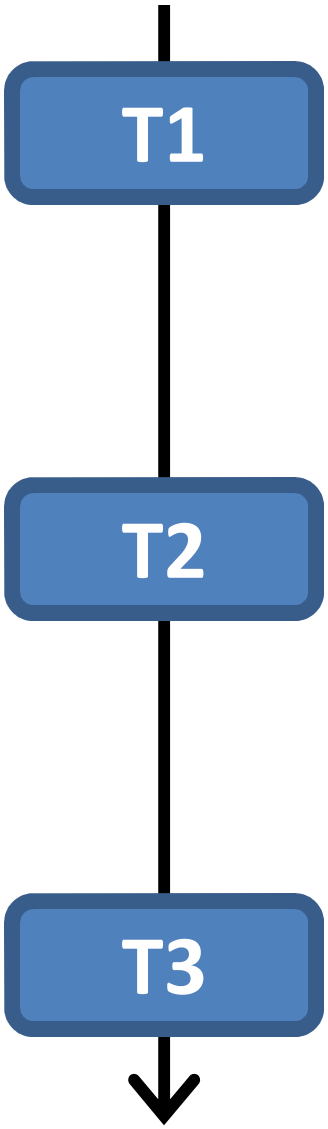
```
R1(X)
W1(Y)
R2(Y)

W1(Y)
W2(Z)

W2(Z)
R3(Z)
```





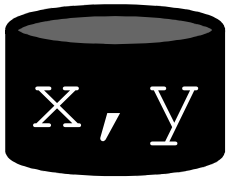


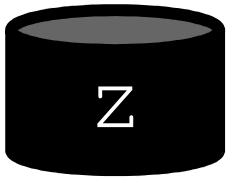
A Closer Look at Serial Execution



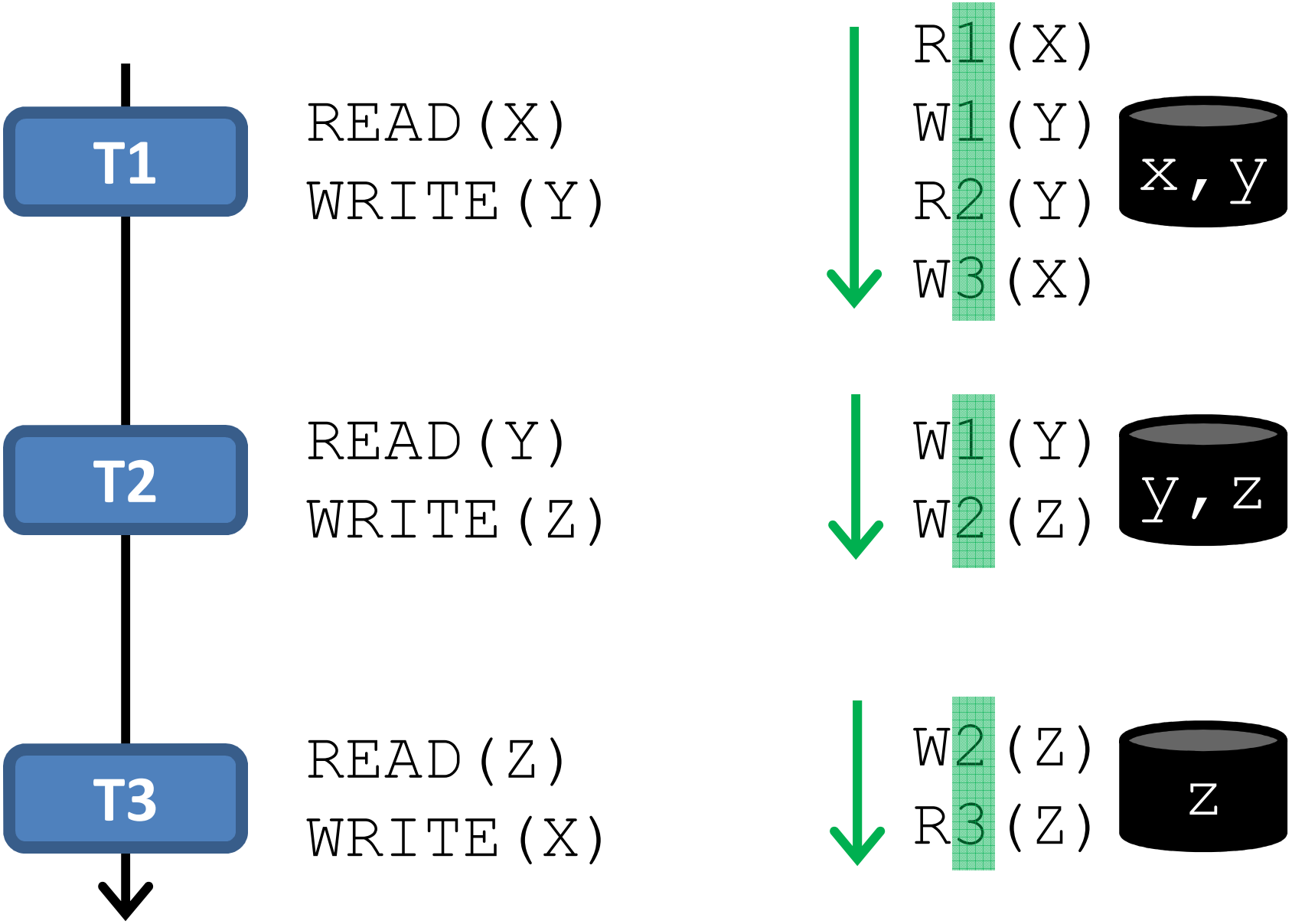

```
READ(X)
WRITE(Y)

READ(Y)
WRITE(Z)

READ(Z)
WRITE(X)
```

```
R1(X)
W1(Y)
R2(Y)
W3(X)

W1(Y)
W2(Z)

W2(Z)
R3(Z)
```

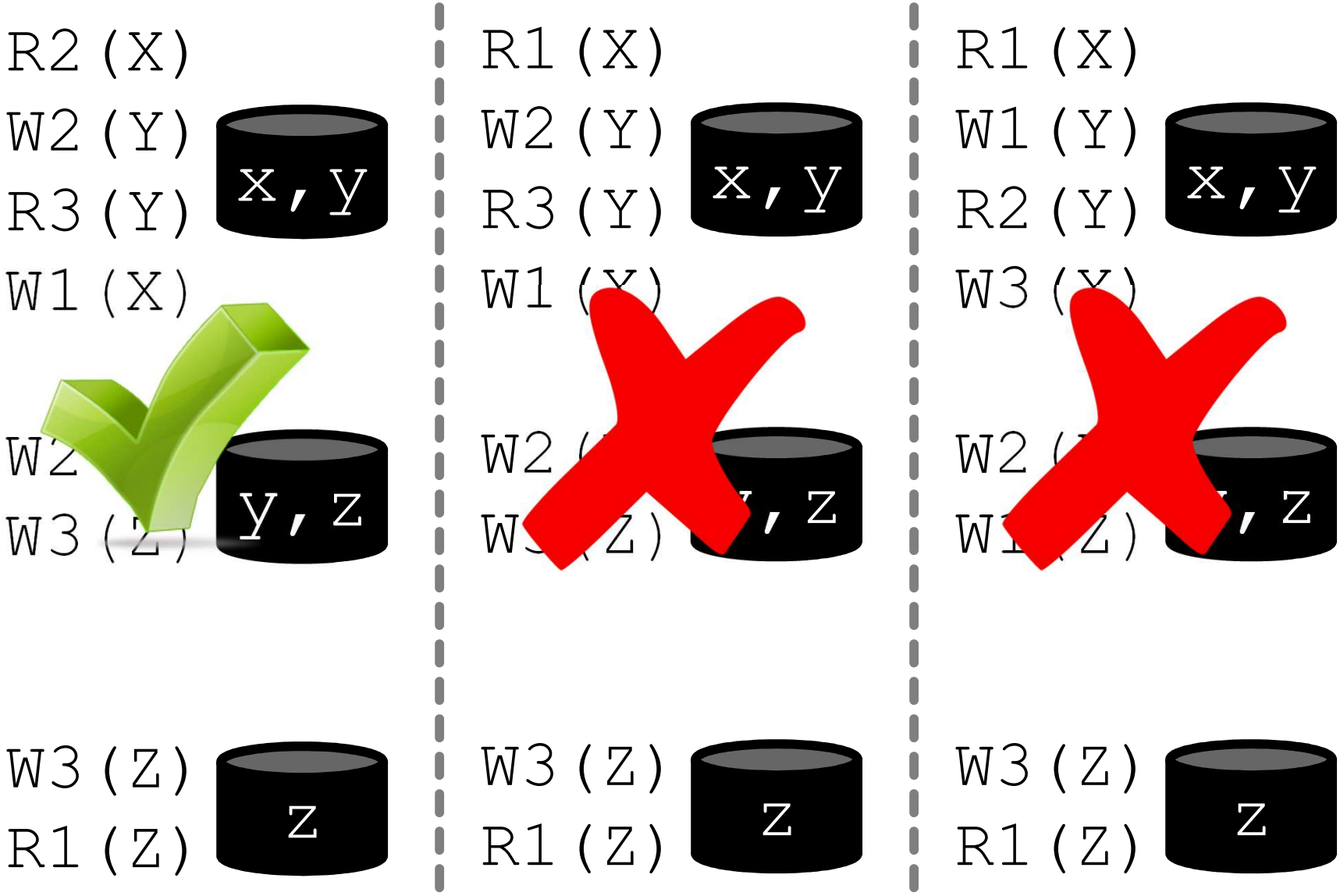




A Closer Look at Serial Execution



Serial or Not?



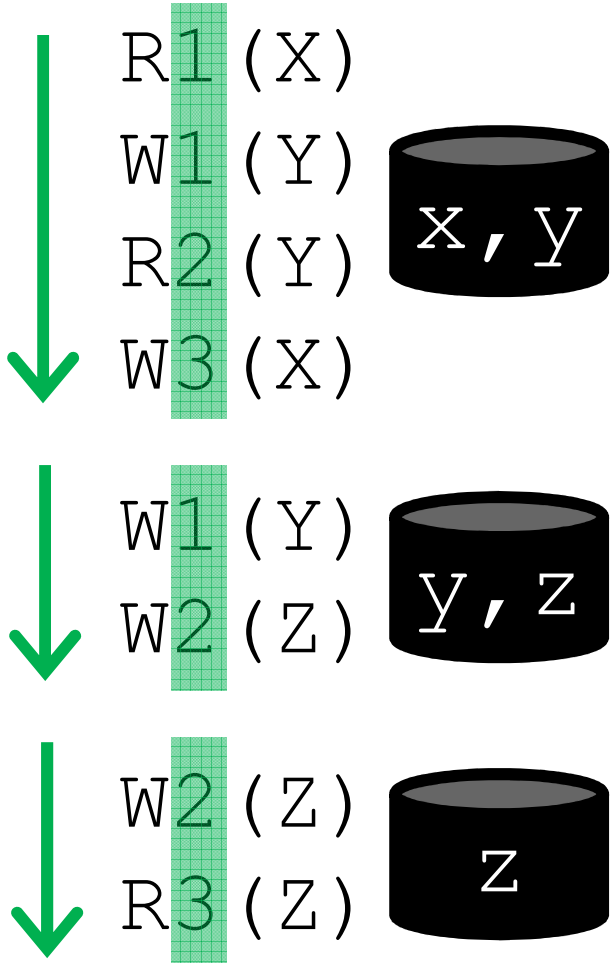
Conflicting Operations

- Two operations *conflict* with each other...
  1. if they access the same data item
  2. AND one of them is a write

- Examples
  - R(X) → W(X)
  - W(X) → R(X)
  - W(X) → W(X)

Conflicting Operations (cont'd)

Total ordering of all operations



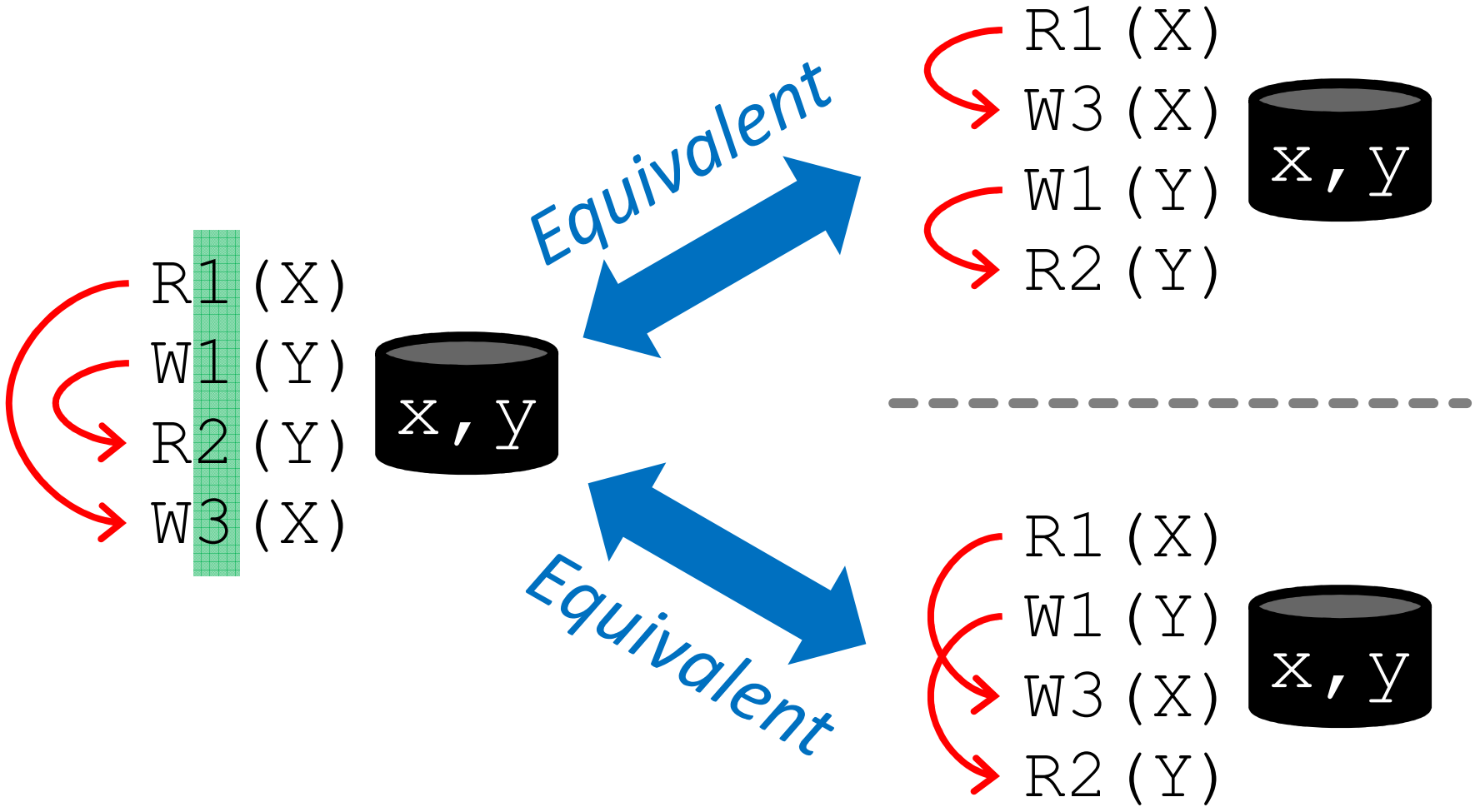
Partial ordering of conflicting operations

R1(**X**) → W3(**X**)

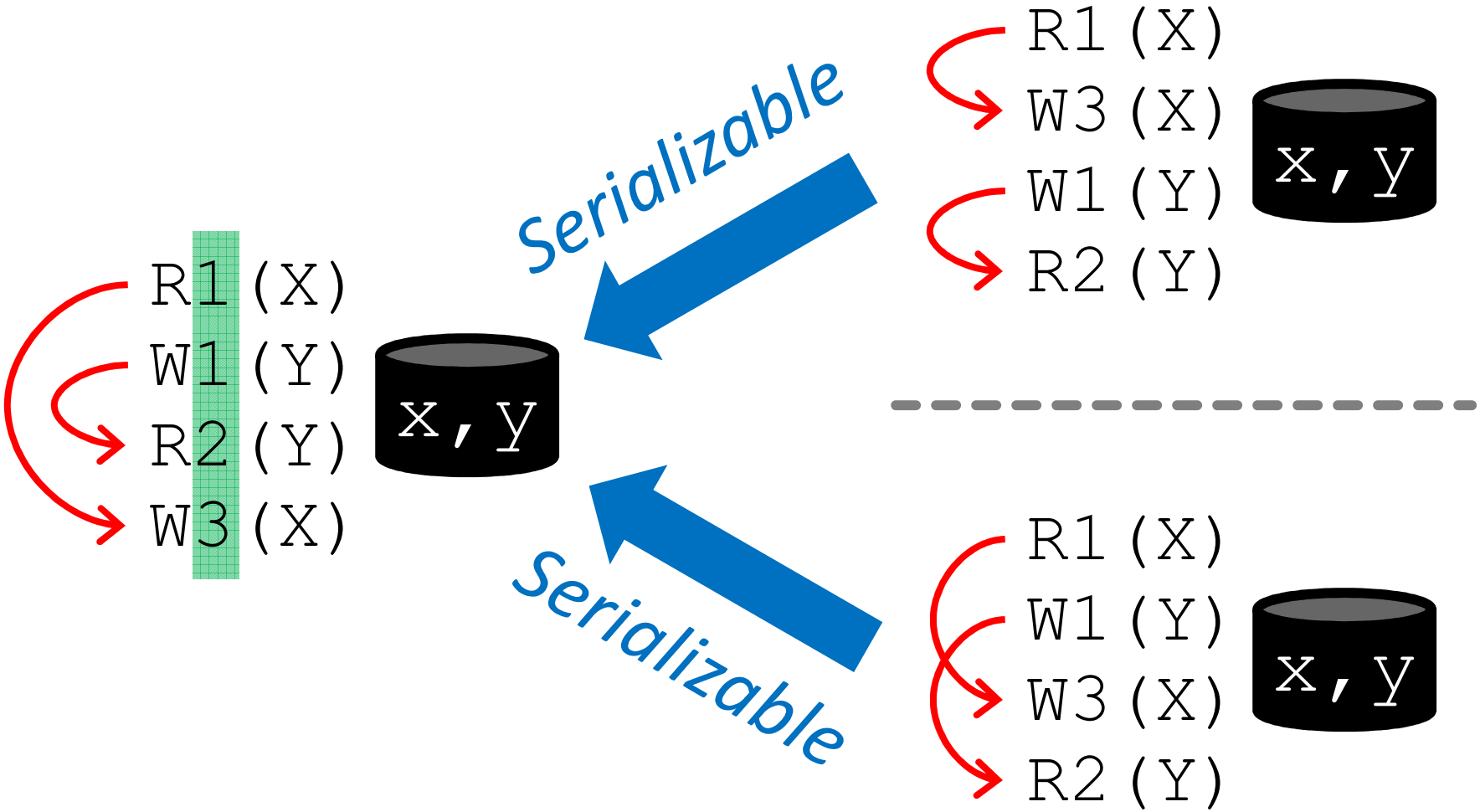
W1(**Y**) → R2(**Y**)

W2(**Z**) → R3(**Z**)

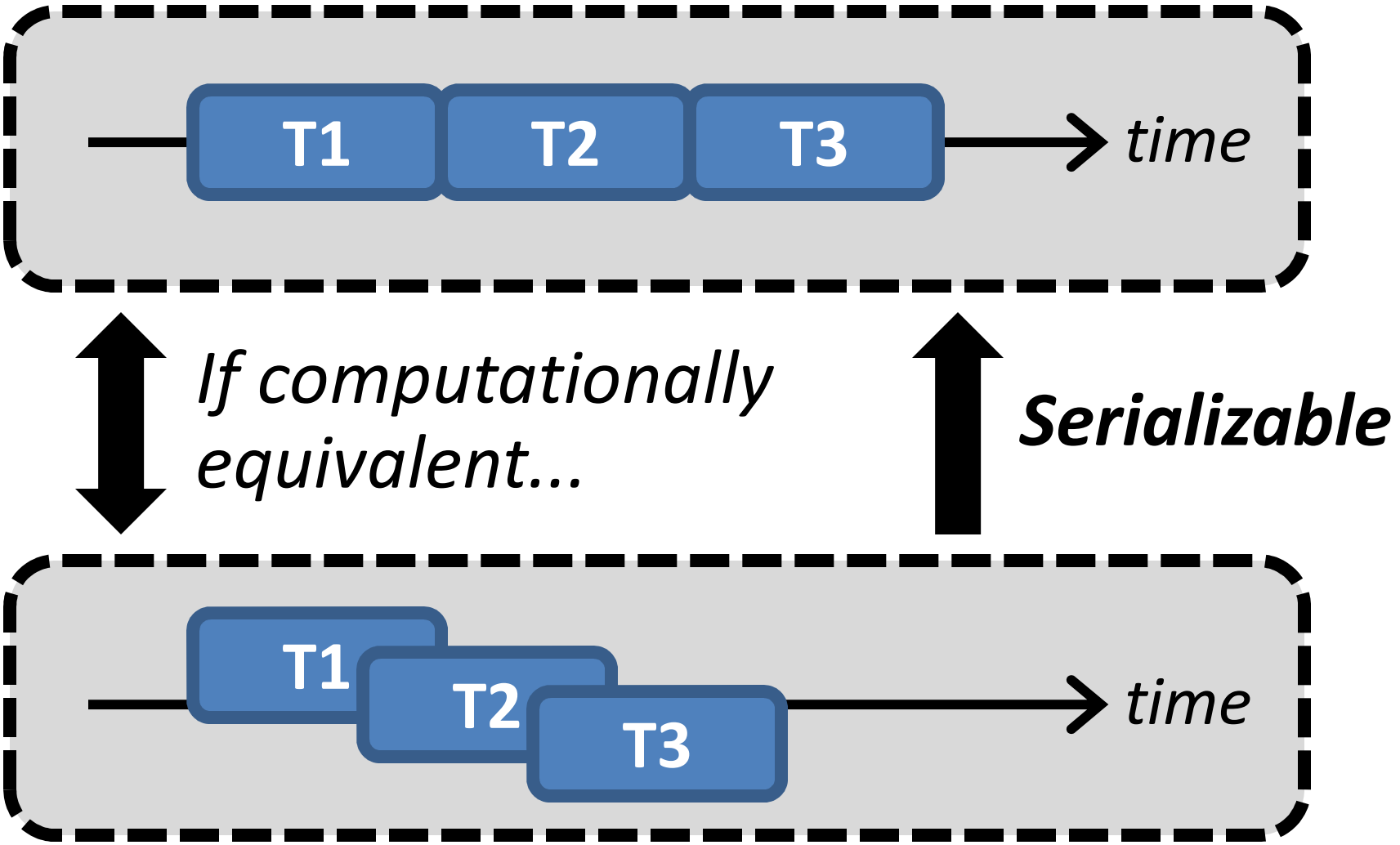
Computational Equivalence



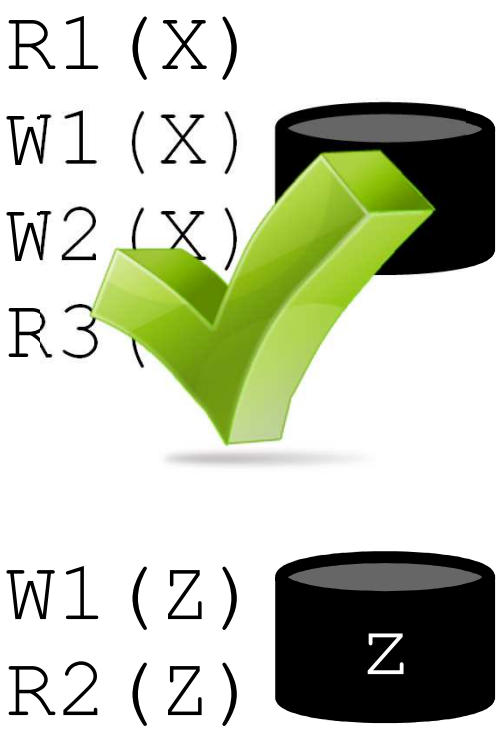
Serializability

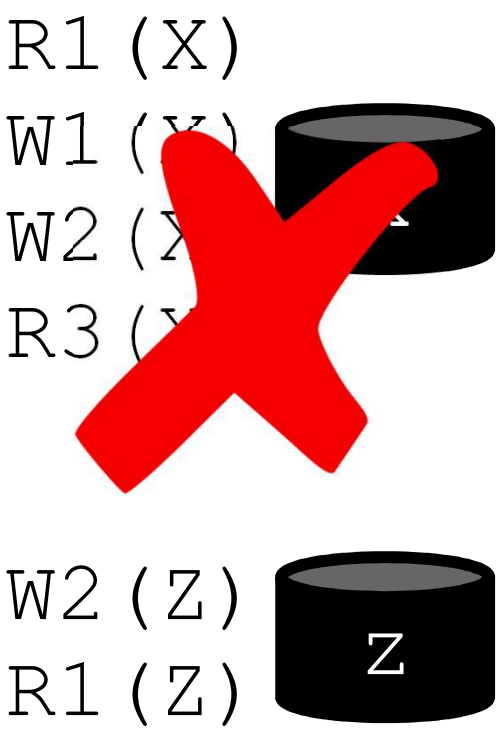


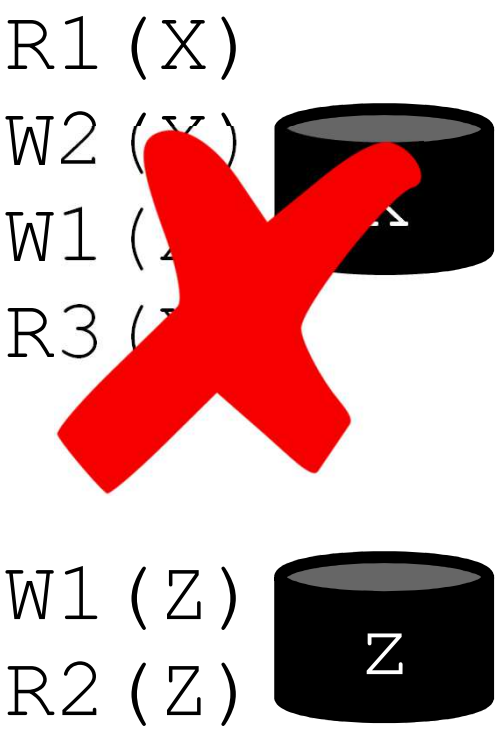
Serializability (cont'd)



Serializable or Not?



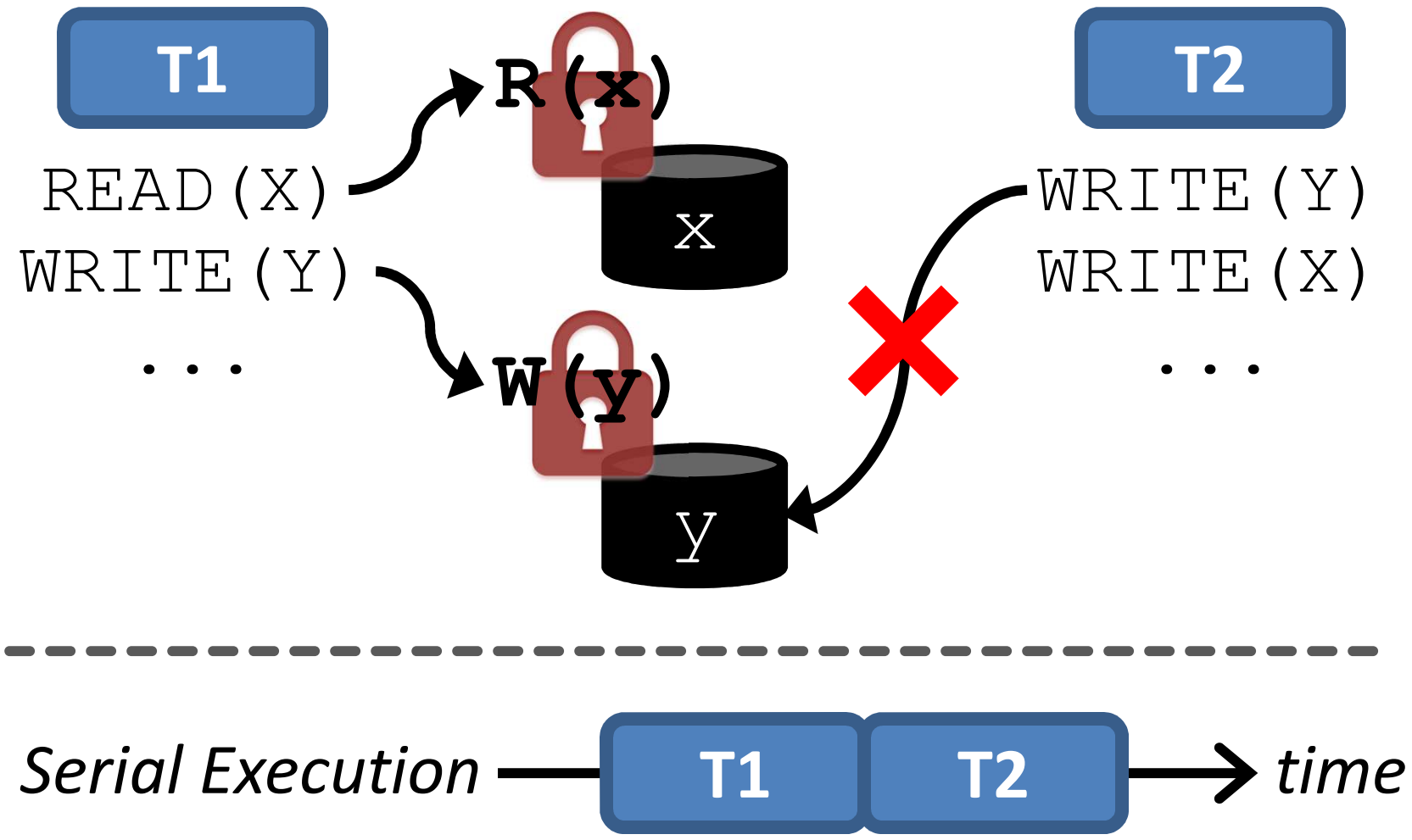




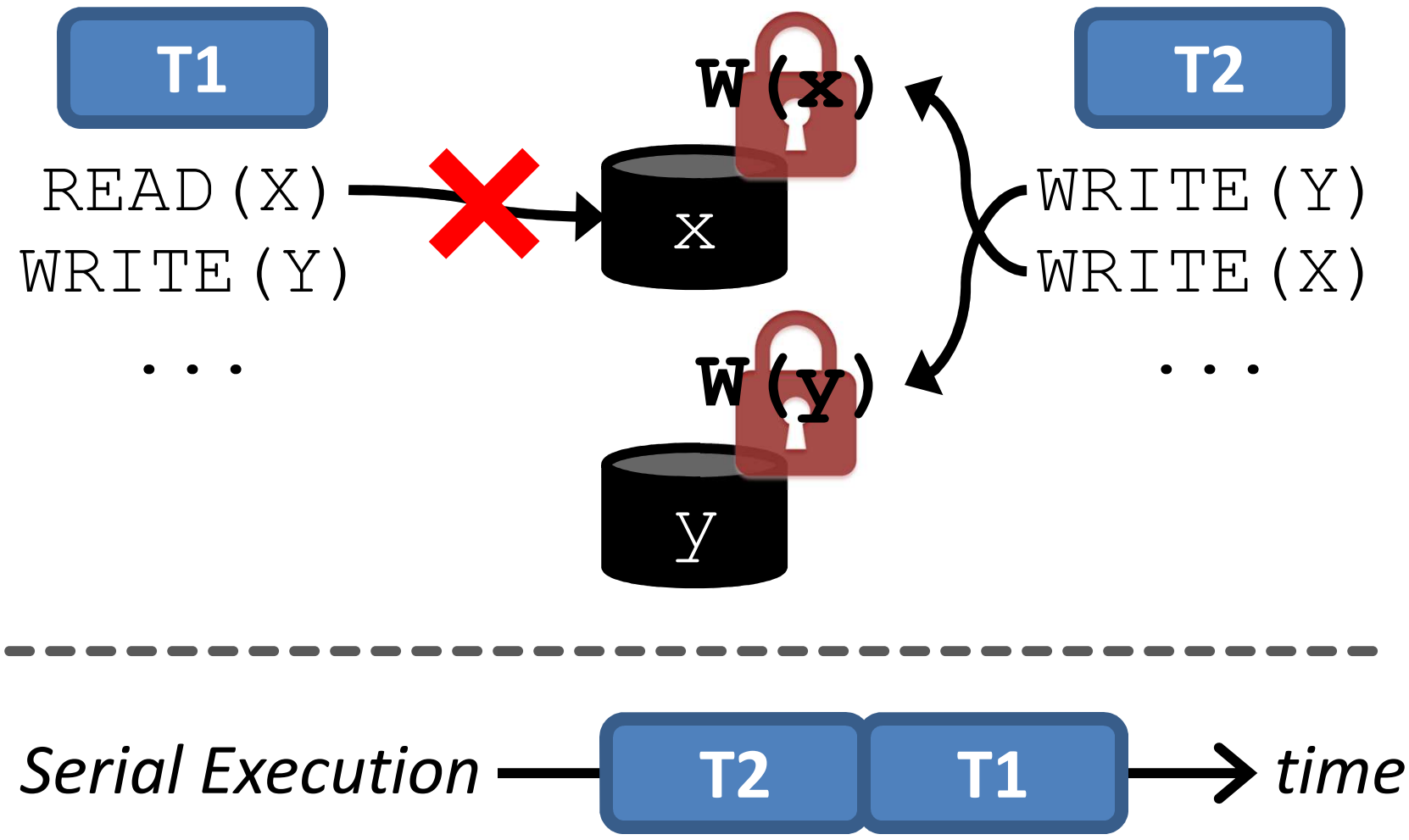
Outline

- Distributed Database Systems
- Correctness: Serializability
- **Two-Phase Locking**
- Timestamp Ordering

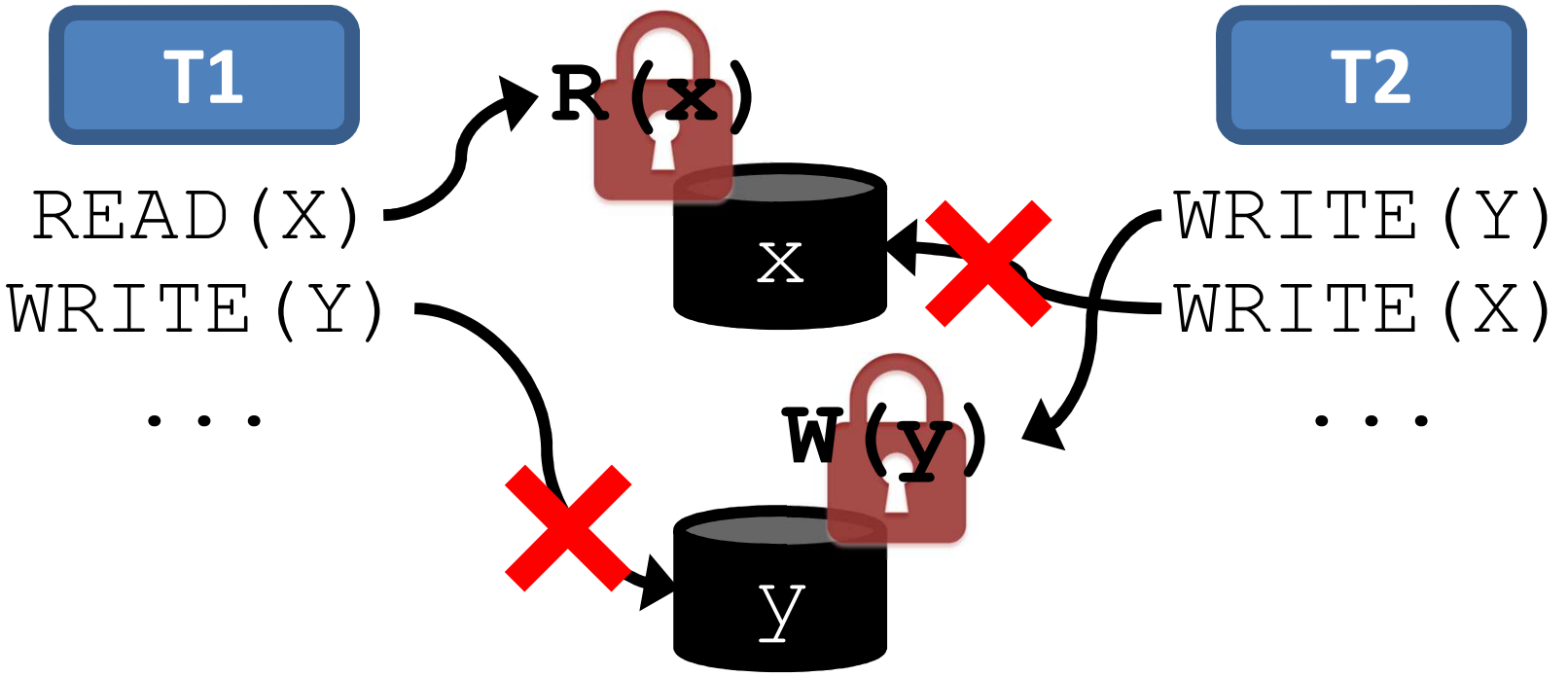
Locks: Exclusive Access to Data



Locks (cont'd)


T1
READ(X)
WRITE(Y)
. . .
W(x)
x
W(y)
y
T2
WRITE(Y)
WRITE(X)
. . .
Serial Execution
T2
T1
time

Locks (cont'd)


T1
T2
R(x)
READ(X)
WRITE(Y)
. . .
x
WRITE(Y)
WRITE(X)
. . .
W(y)
y
Serial Execution
Deadlock
time

A Closer Look at Locks

- Two types of locks
  - ReadLocks: **`R(x)`**
    - ➔ *Required before a read access*
  - WriteLocks: **`W(x)`**
    - ➔ *Required before a write access*

- Two locks *conflict* with each other...
  1. if they are on the same data item
  2. AND one of them is a write

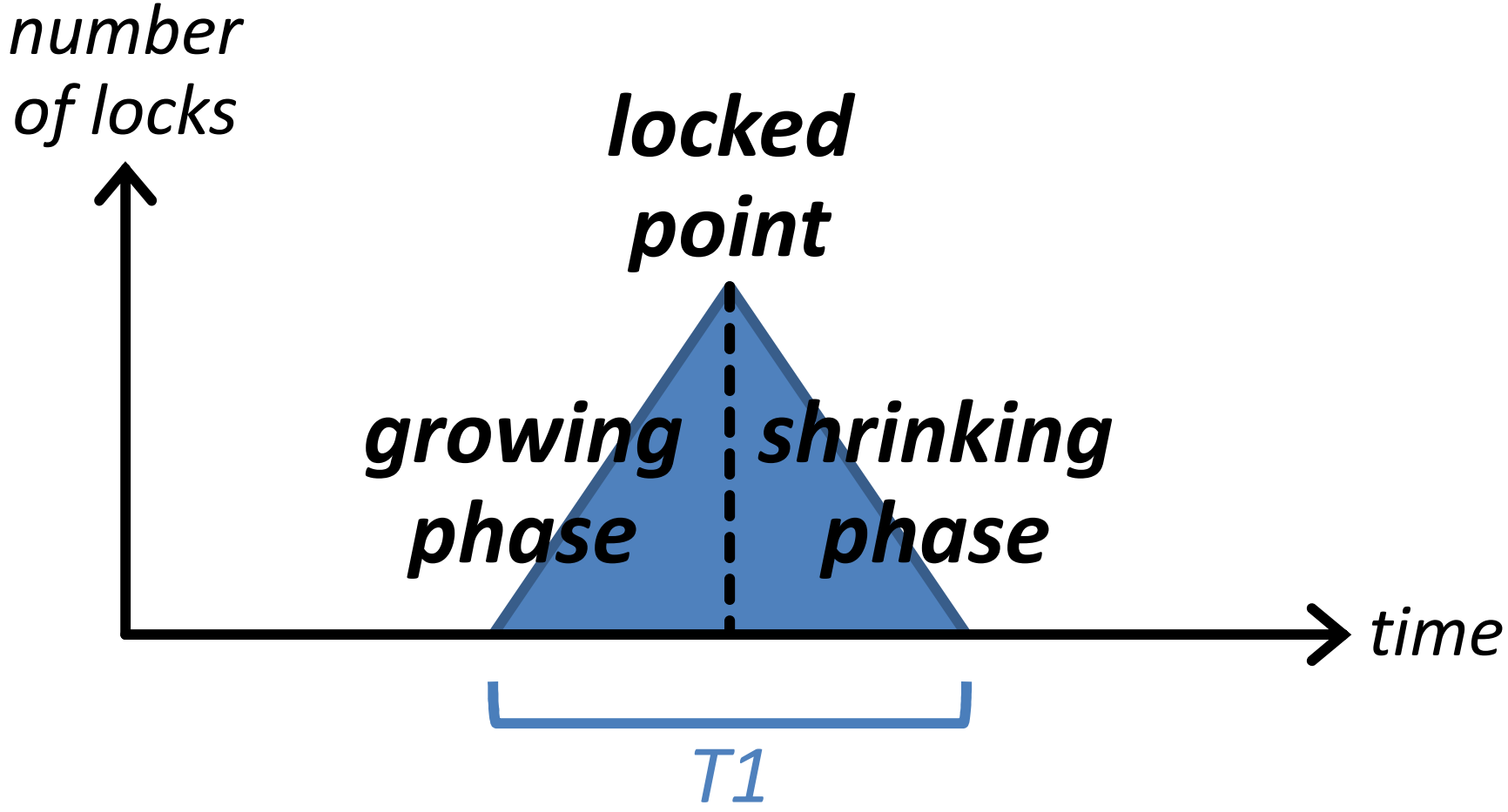
A Closer Look at Locks (cont'd)

- **Lock Ownership Rules**
  1. Different transactions cannot have conflicting locks

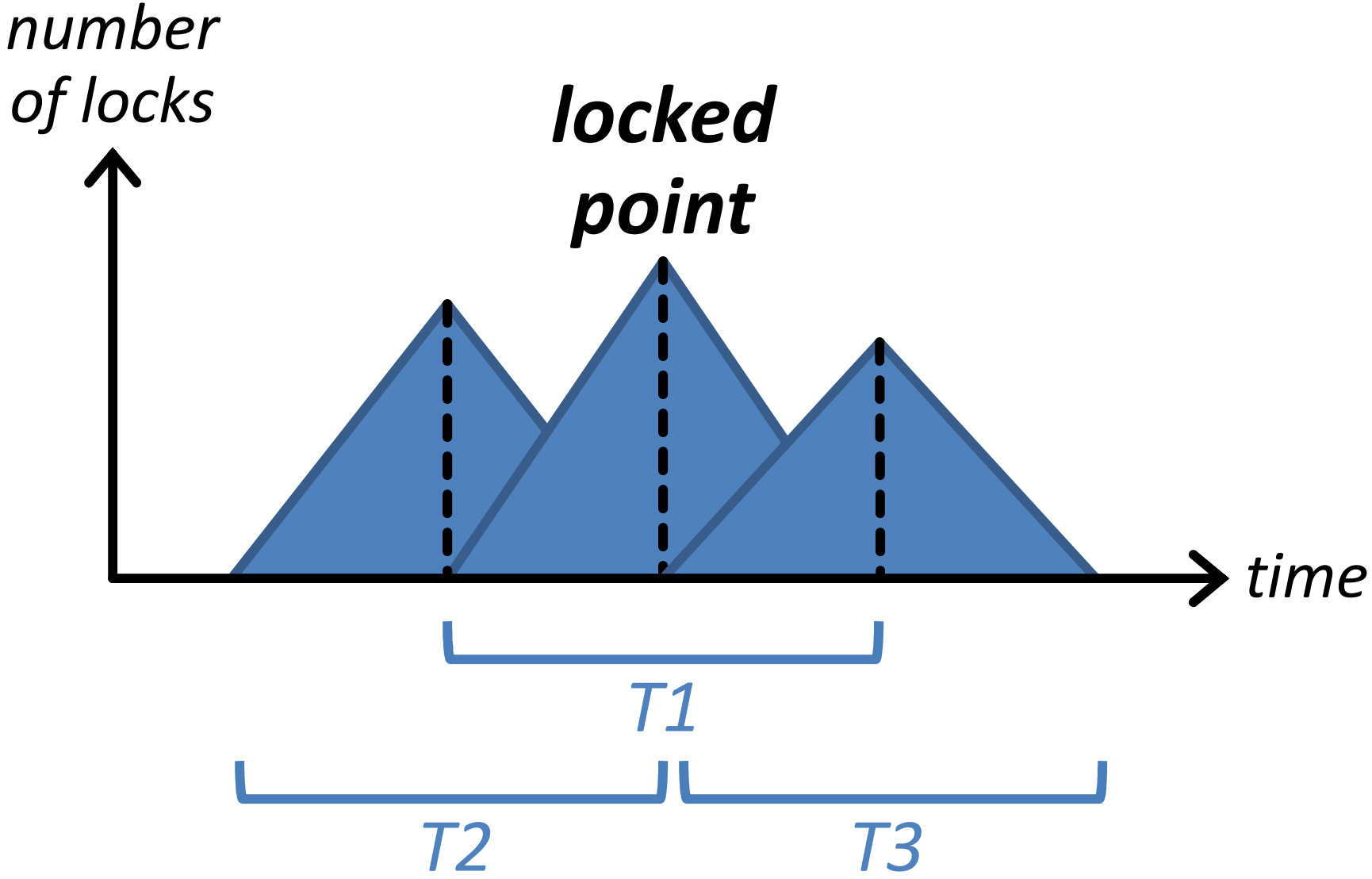
     ➔ *Exclusive access*
  2. Once a transaction relinquishes a lock, it cannot obtain additional locks

     ➔ *Obtain, obtain, relinquish, relinquish*

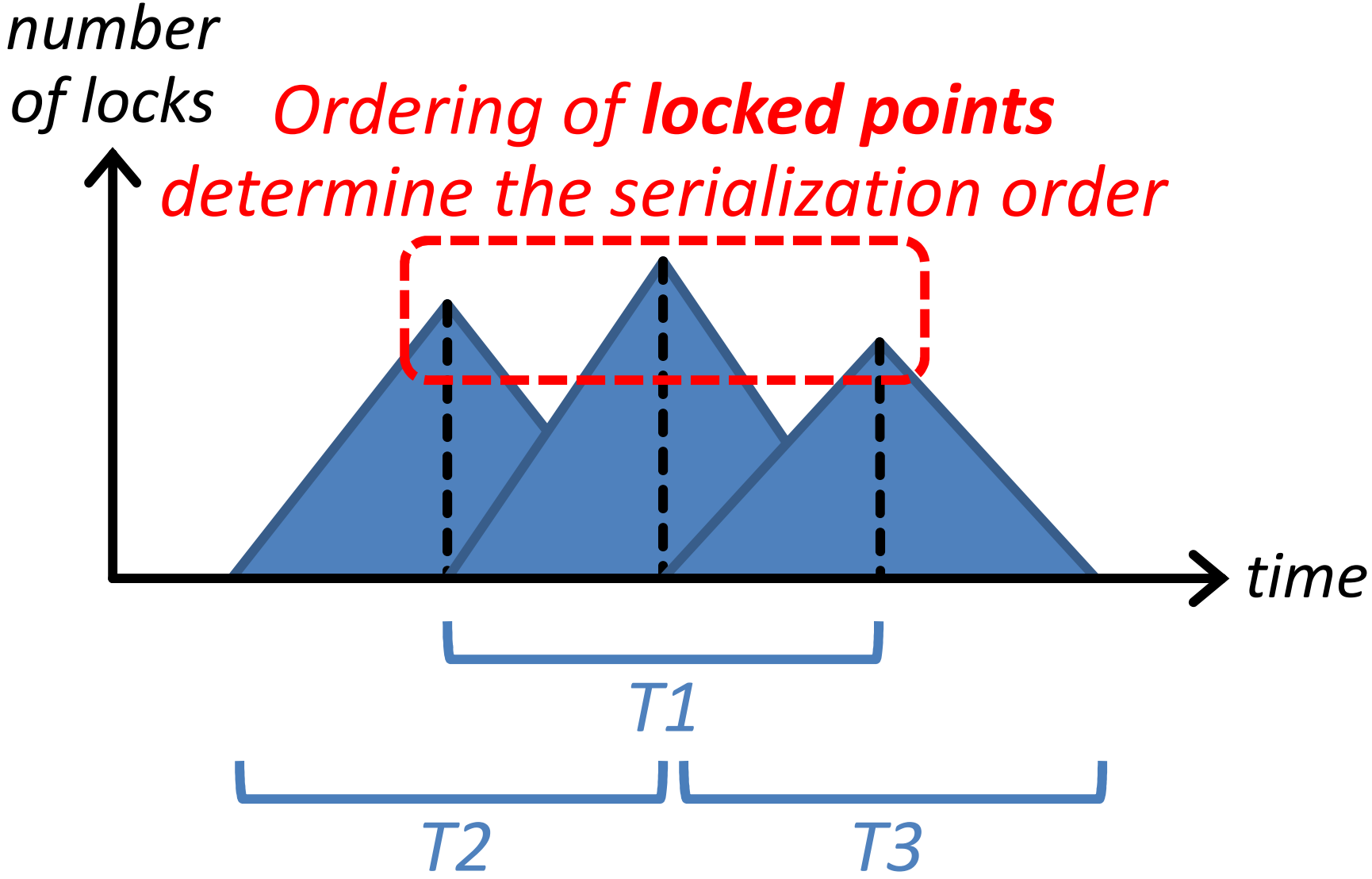
Two Phases


number
of locks
locked
point
growing
phase
shrinking
phase
time
T1

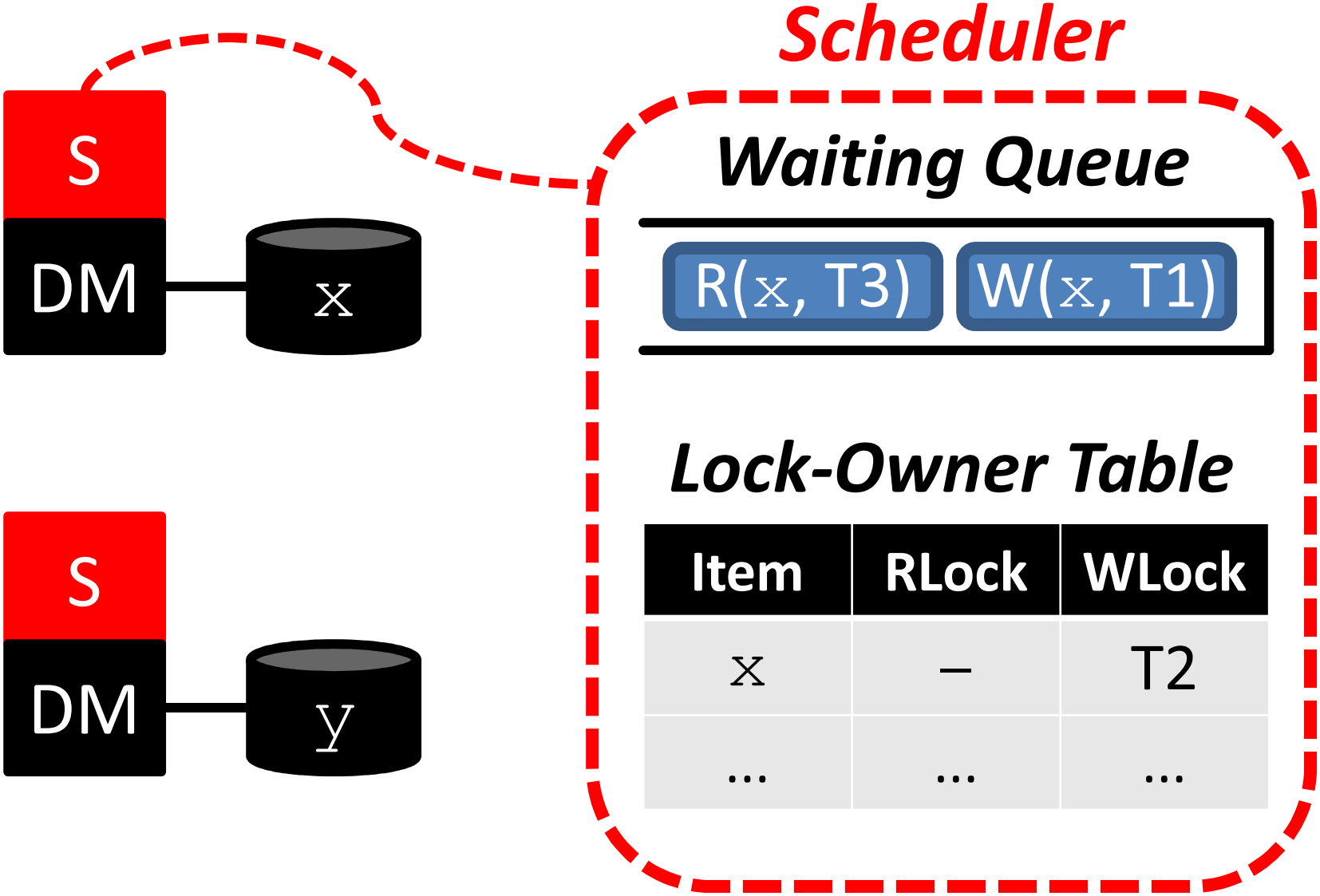
Two Phases (cont'd)



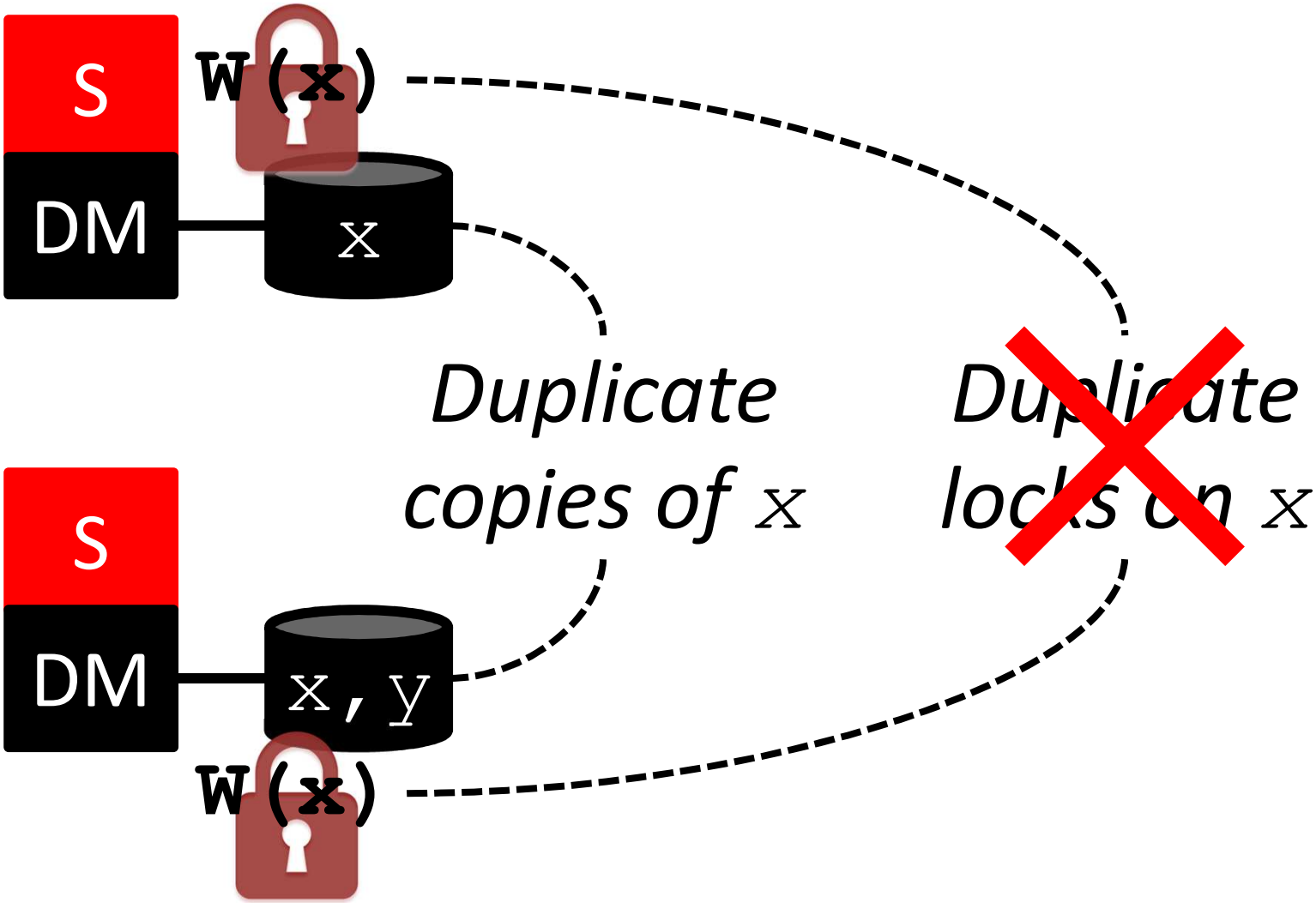
Two Phases (cont'd)



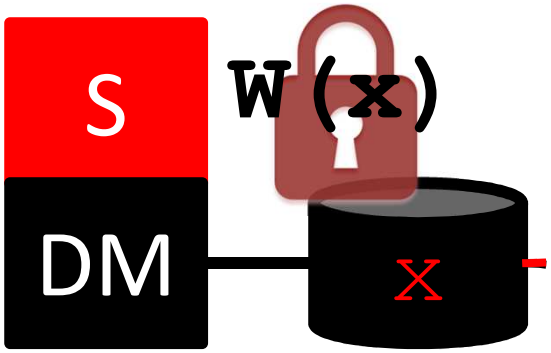
Basic Lock Scheduler



Variant: Primary Copy



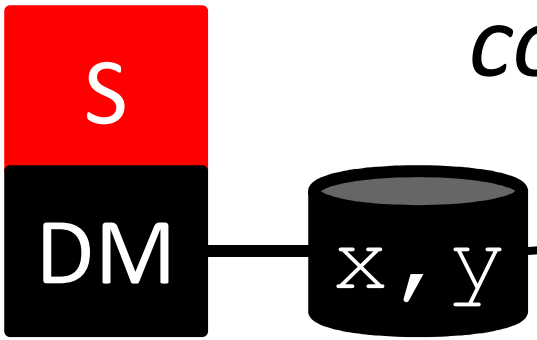
Variant: Primary Copy



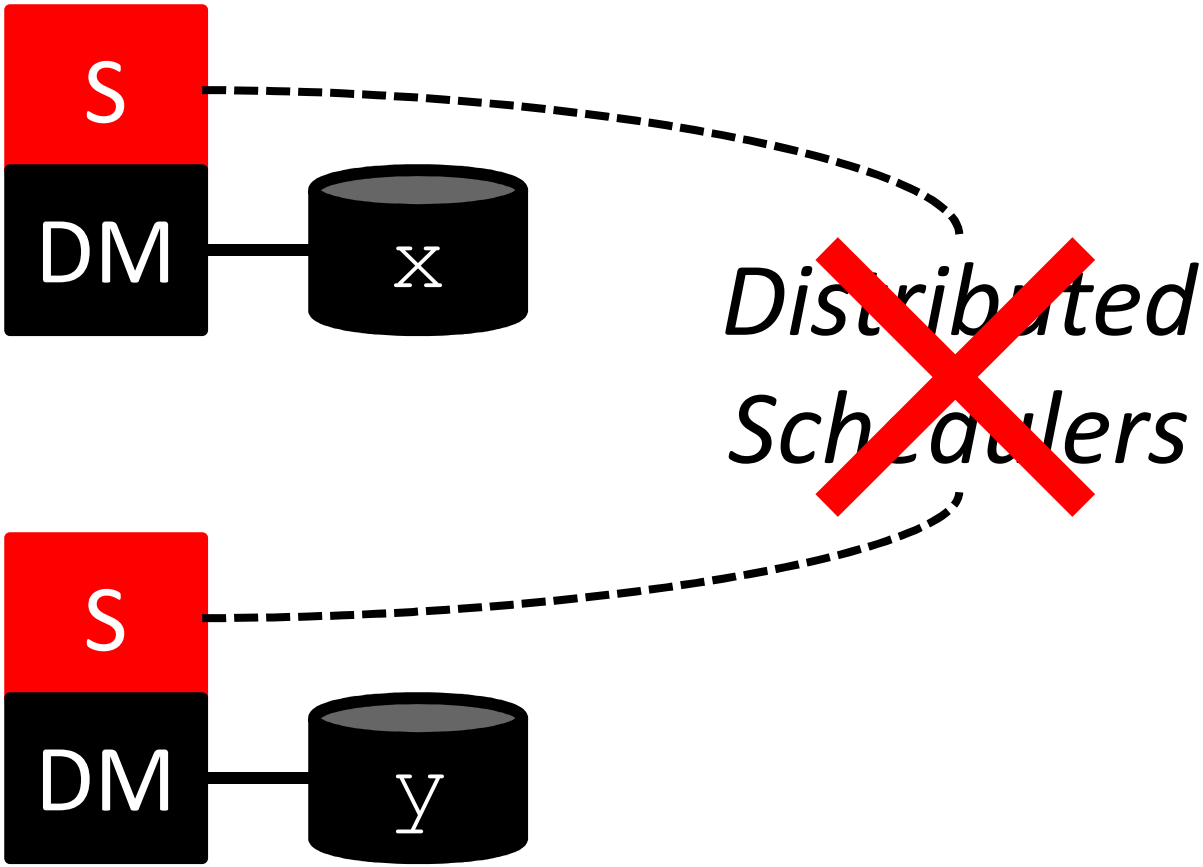
Primary Copy:

Resolves <u>all</u> locks on x

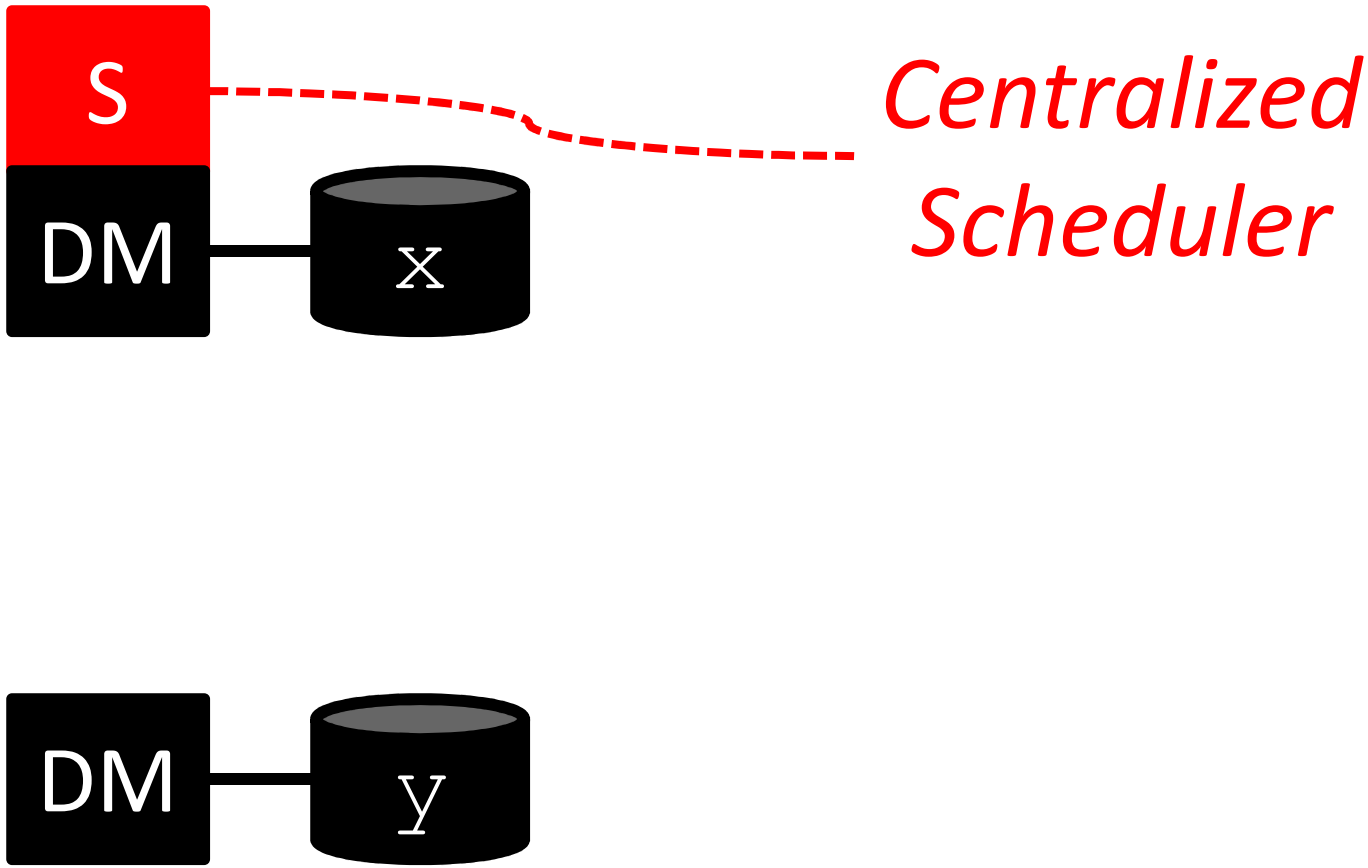
Duplicate copies of x



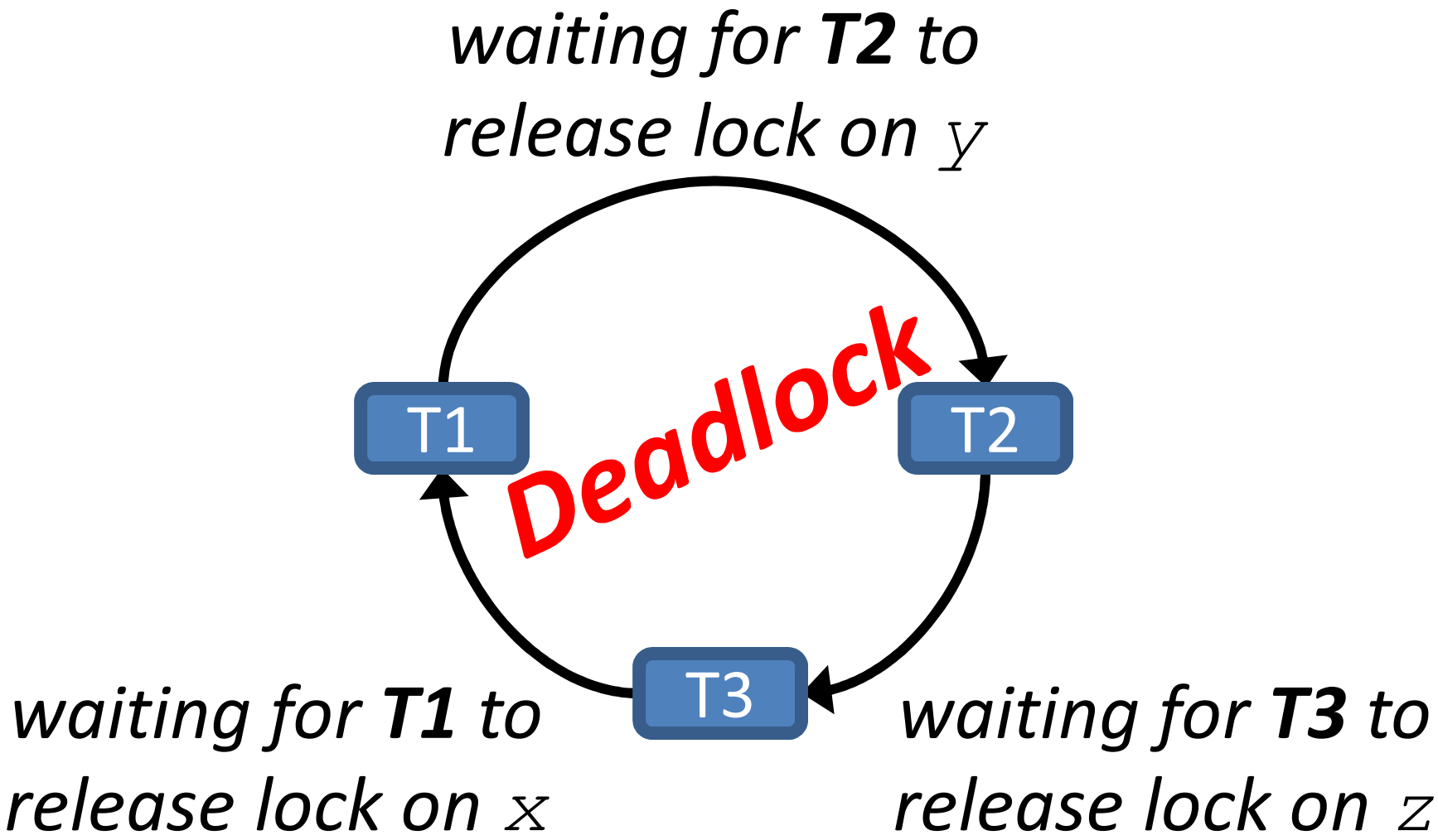
Variant: Centralized



Variant: Centralized



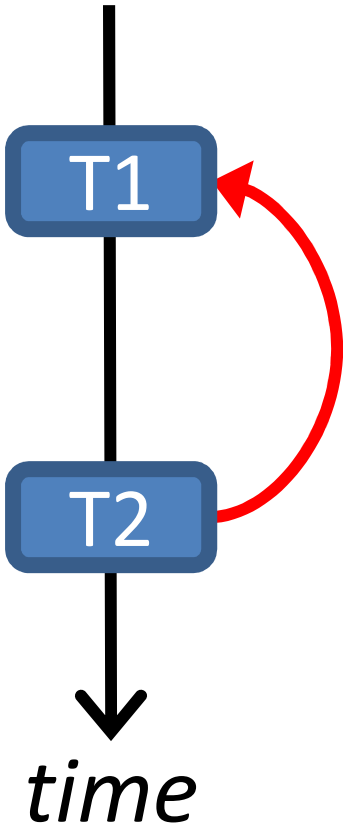
When Two-Phase Locking Fails



Handling Deadlocks

1. **Deadlock Prevention**
   - "Pessimistic" approach
   - Abort/restart a transaction when a deadlock might occur
2. **Deadlock Detection**
   - "Optimistic" approach
   - Abort/restart a transaction only when a deadlock actually occurs

1. Deadlock Prevention



waits-for

Q. Which to abort?

- ***Preemptive**: T1*
- ***Non-preemptive**: T2*

Q. When to abort?

- ***Desc. Priority**: T1 > T2*
- ***Asc. Priority**: T1 < T2*

Note: An aborted transaction is transparently restarted by the TM; the user remains oblivious of the fact

1. Deadlock Prevention (cont'd)

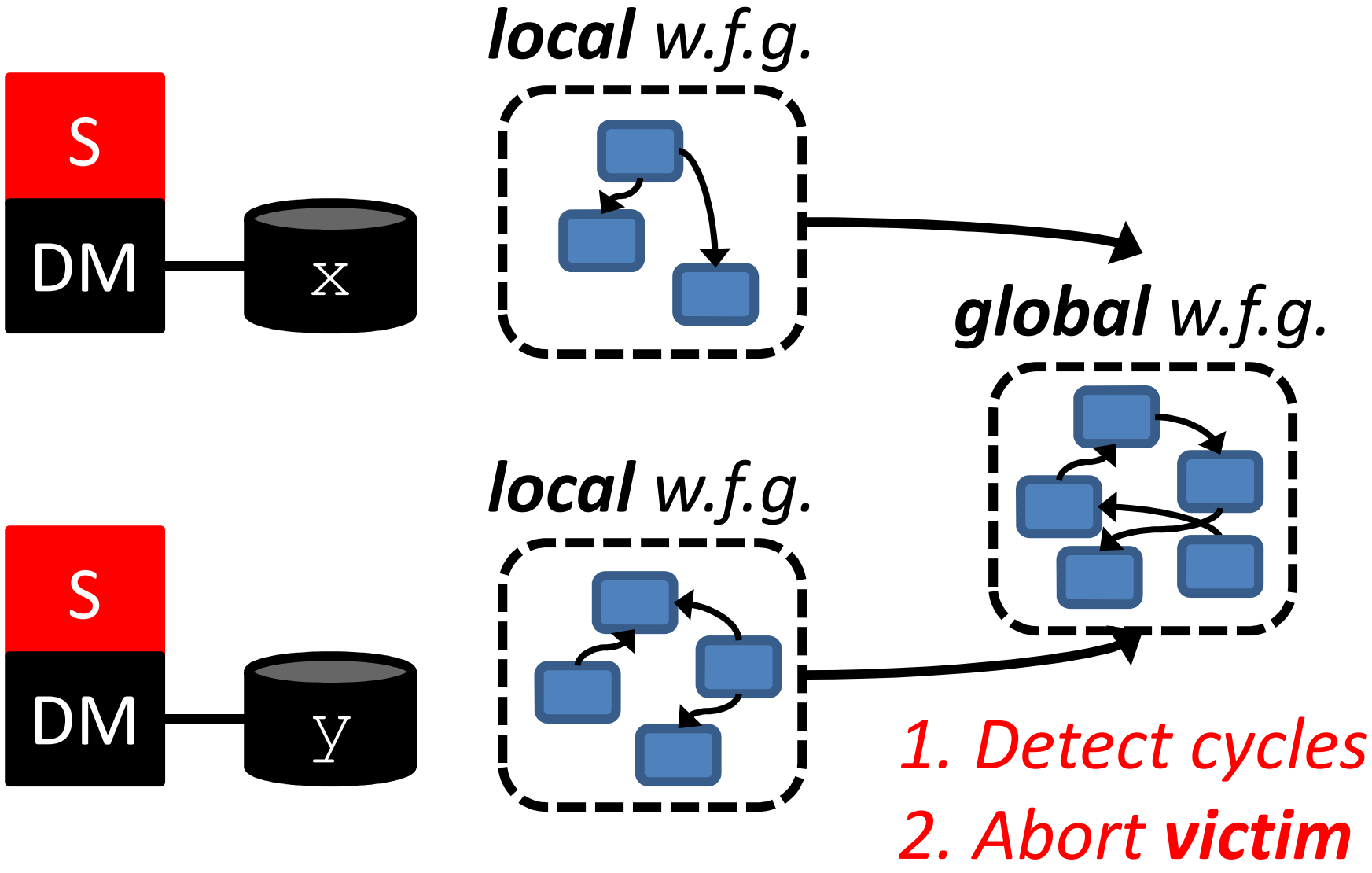
Scheme #1: **"Wound-Wait"**

- Preemptive
- Descending Priority

Scheme #2: **"Wait-Die" Scheme**

- Non-Preemptive
- Ascending Priority

2. Deadlock Detection



2. Deadlock Detection (cont'd)

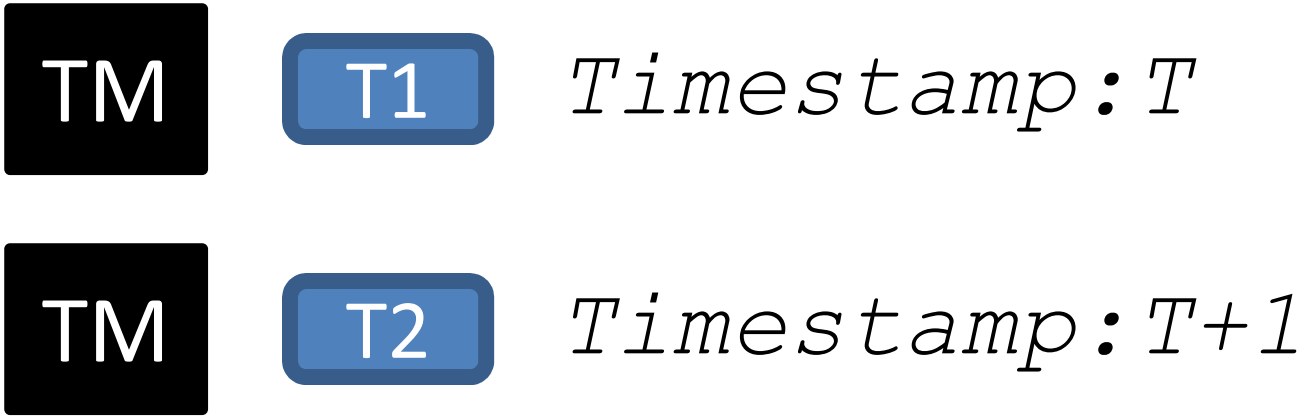
- **Requires global view of database**
  - Local views are insufficient to detect cycles
  - Local views must be merged periodically
  - Constructing a global view is *expensive*

- **Takes a long time to detect deadlocks**
  - Depends on how often merges are performed
  - Typical interval: 100s of milliseconds

Outline

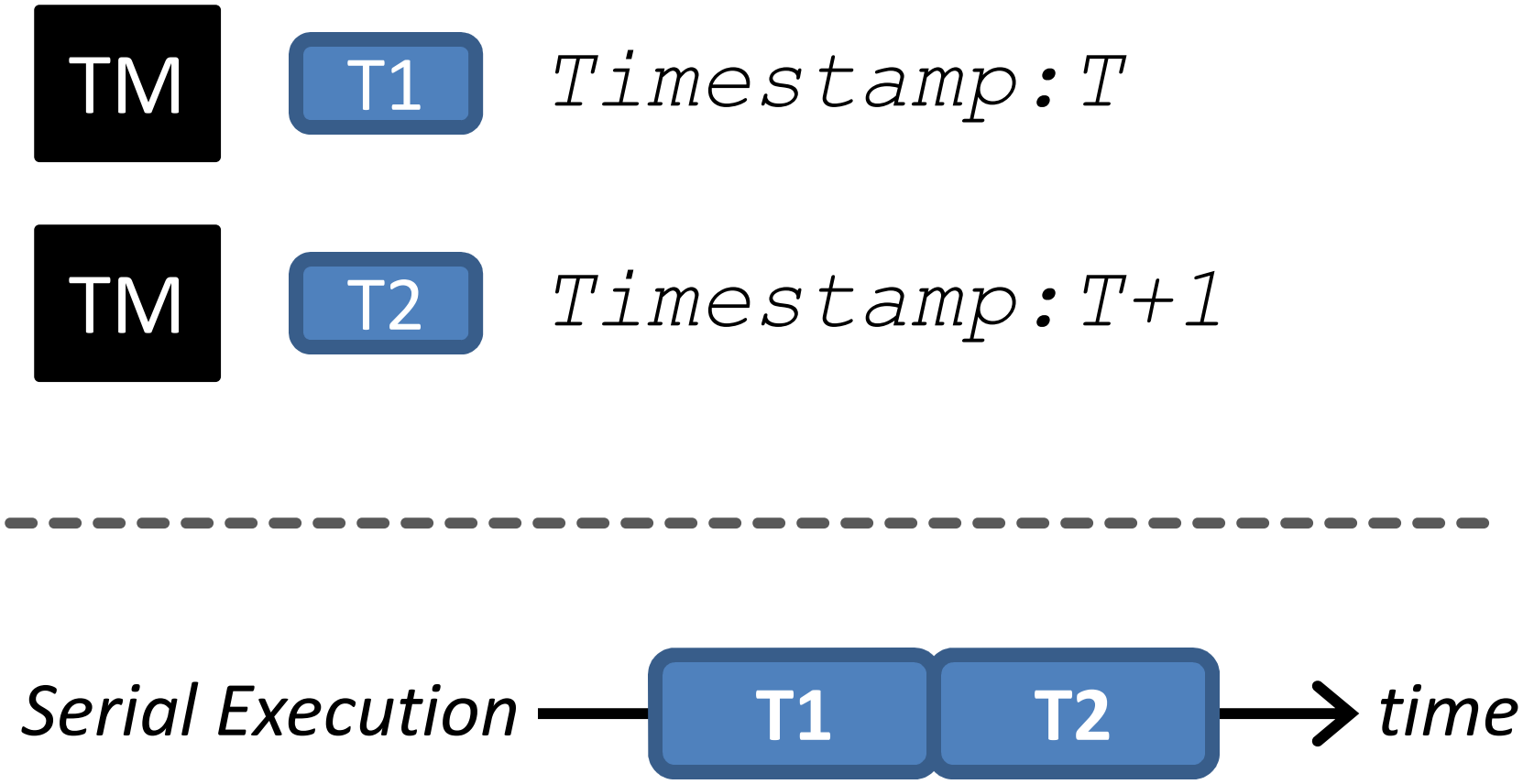
- Distributed Database Systems
- Correctness: Serializability
- Two-Phase Locking
- **Timestamp Ordering**

Timestamps

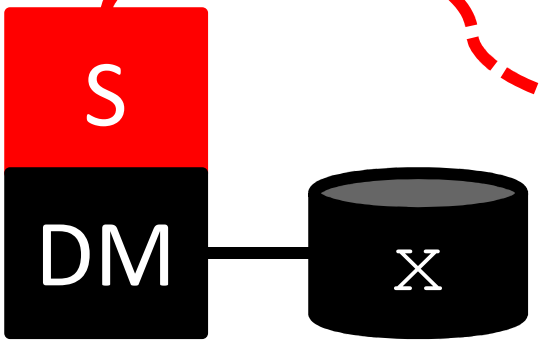


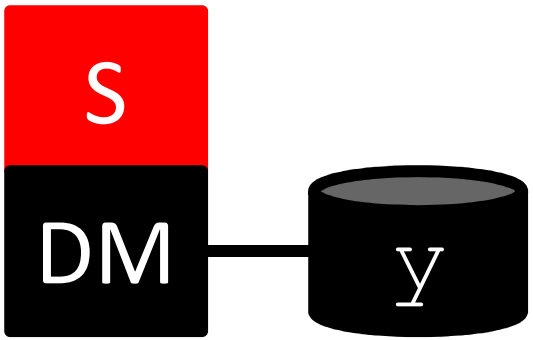
- Each transaction assigned a unique ***timestamp***
  - e.g., POSIX time concatenated with TM identifier
- All conflicting operations are ordered with respect to the timestamp of their transaction

Timestamps (cont'd)



Timestamp Scheduler





Scheduler

Timestamp Table

Item	RTS	WTS
x	T	T+1
…	…	…

Stores the largest timestamp operation that accessed an item

Timestamp Scheduler (cont'd)

- **READ Operation**

```
if (TS < WTS(x))
  abort();
else
  RTS(x) = max(TS,RTS(x));
```

- If a transaction is aborted, it is assigned a newer/larger timestamp, then restarted

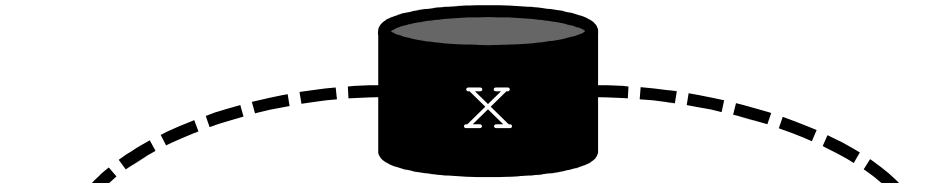
Timestamp Scheduler (cont'd)

- **`WRITE` Operation**

```
if (TS < RTS(x) || TS < WTS(X))
  abort();
else
  WTS(x) = max(TS,WTS(x));
```

- If a transaction is aborted, it is assigned a newer/larger timestamp, then restarted

Variant: Multiversion



set

RTS
T
T+2
...

versions

WTS	value
T-1	0
T+1	1
...	

Variant: Multiversion (cont'd)

- **`READ` Operation**
  1. Read the *version* of x with largest timestamp less than TS
  2. Add operation to *set*

- Reads are never aborted

Variant: Multiversion (cont'd)

- **`WRITE` Operation**
  - Find the *version* of x that has the next largest timestamp compared to TS
  - If an entry of x in *set* lies between these two timestamps, then abort
  - Else add operation to *version*

Half-Time

Today's Papers

1. ***Concurrency Control in Distributed Database Systems (1981).***
   Bernstein & Goodman

2. ***Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment (1993).***
   Samaras, Britton, Citron, Mohan

Executive Summary (*Samaras et al.*)

1. **Problem**: Two-phase commit (the widely employed commit protocol) incurs high networking/logging overheads

2. **Solution**: *"Make the common case fast"*
   - 11 performance optimizations
   - Reduces the overheads of two-phase commit

Outline

- **Two-Phase Commit (2PC)**
- Baseline 2PC
- Optimized 2PC

Four Desirable Properties

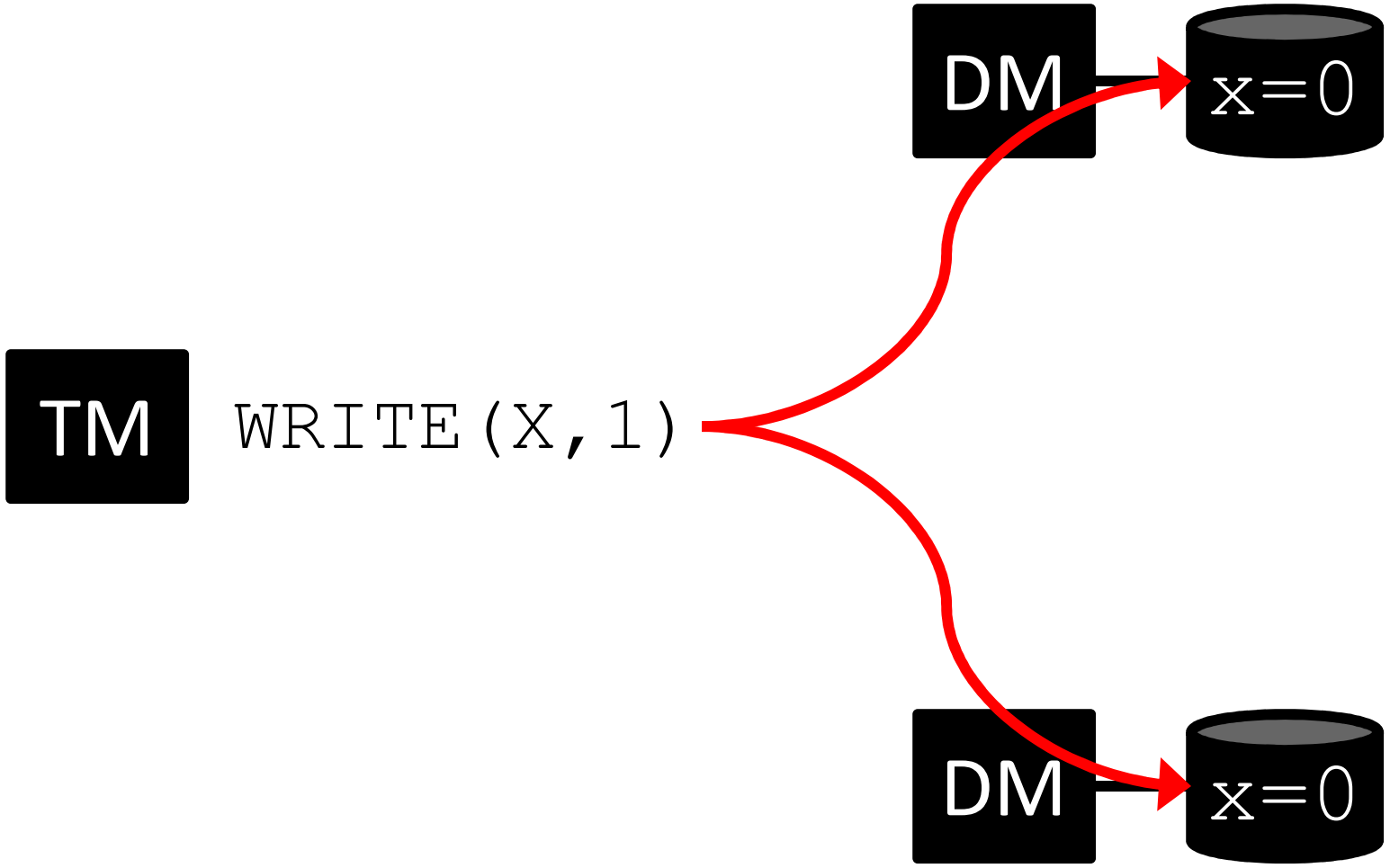
- **Atomicity**: a transaction either commits in its entirety or does not commit at all
- **Consistency**: a transaction does not leave the database in an illegal state

Commit Protocol

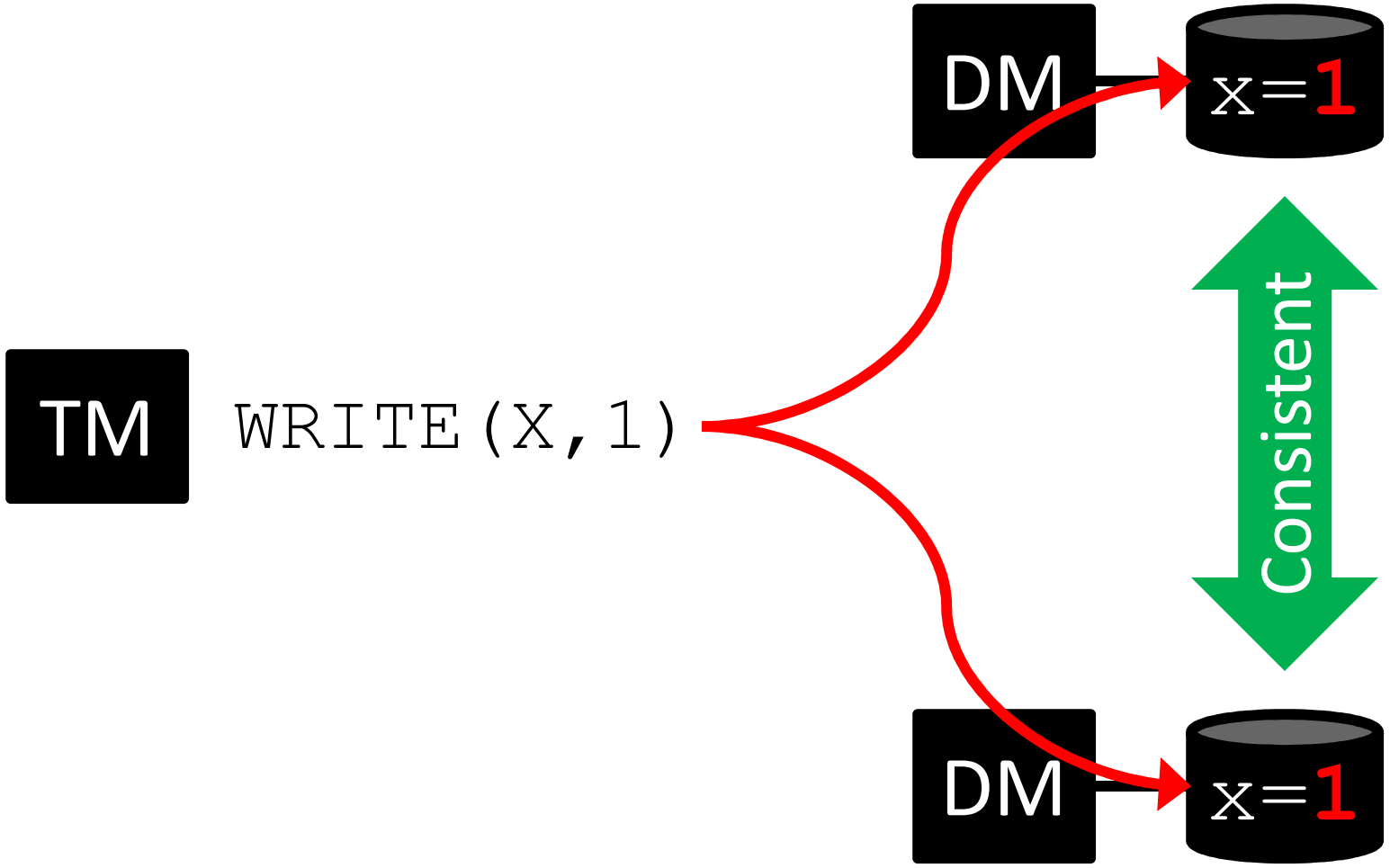
- **Isolation**: transactions do not interfere with each other
- **Durability**: a committed transaction stays committed

*Referred to as **ACID** properties*

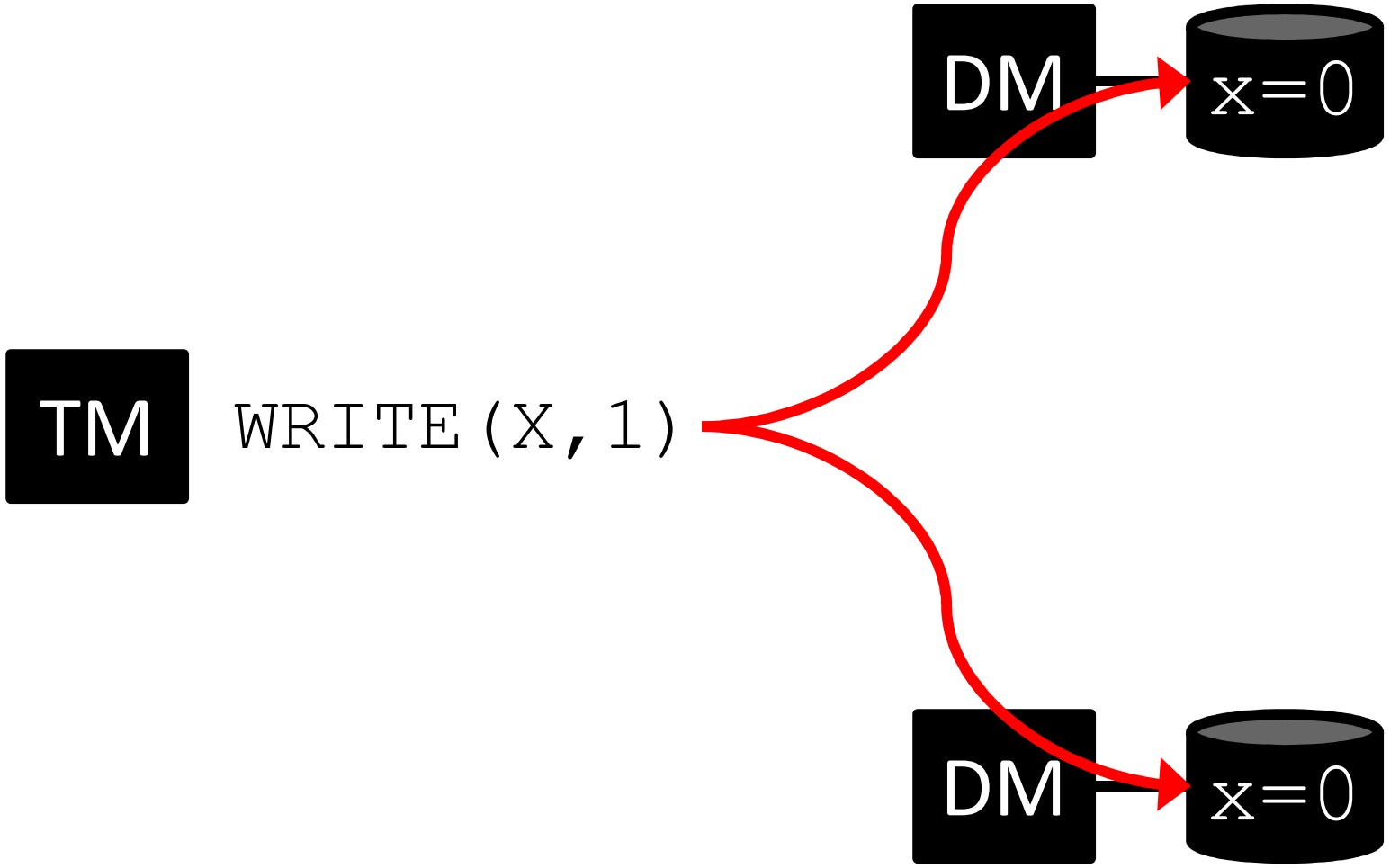
Single-Phase Commit (1PC)



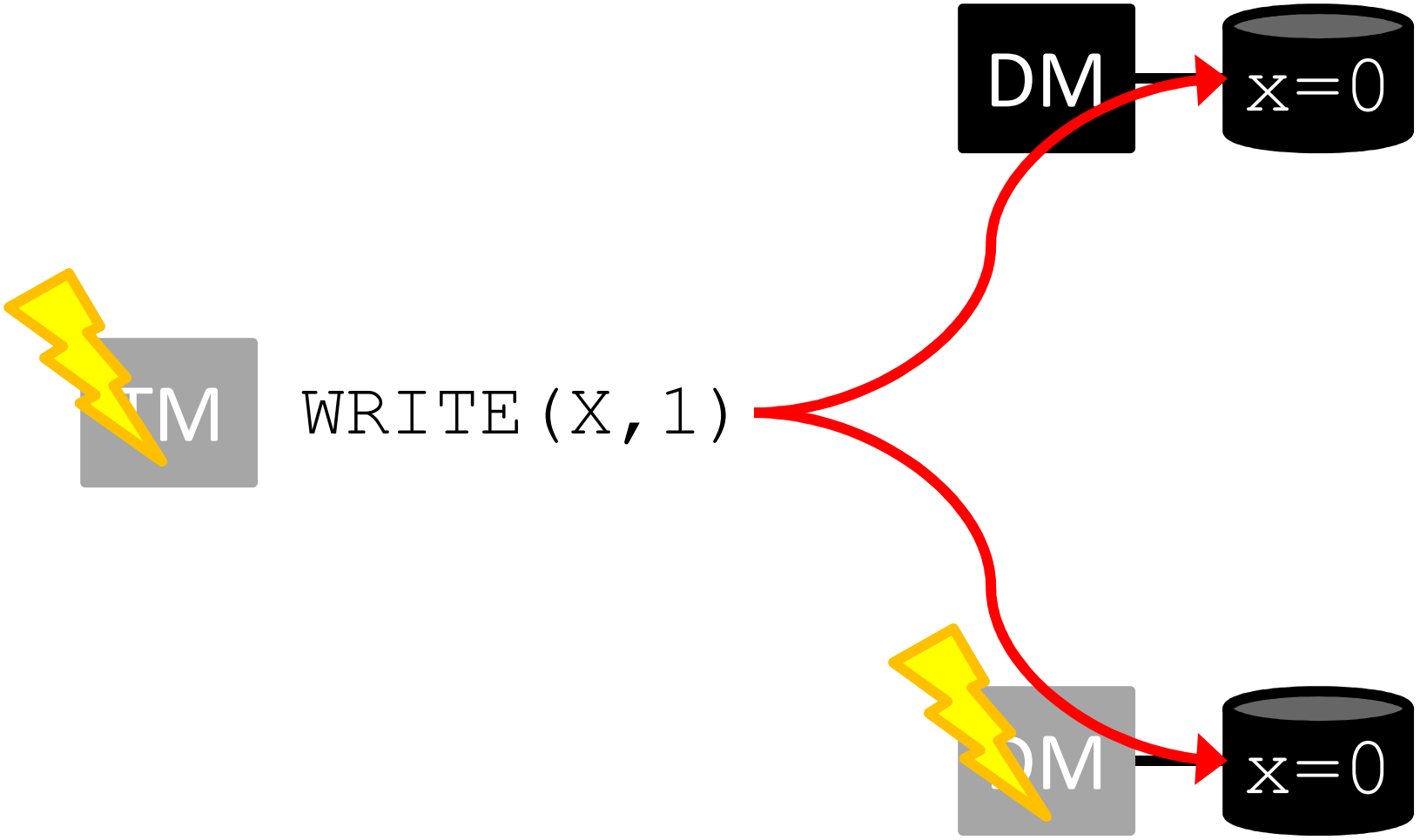
Single-Phase Commit (1PC)



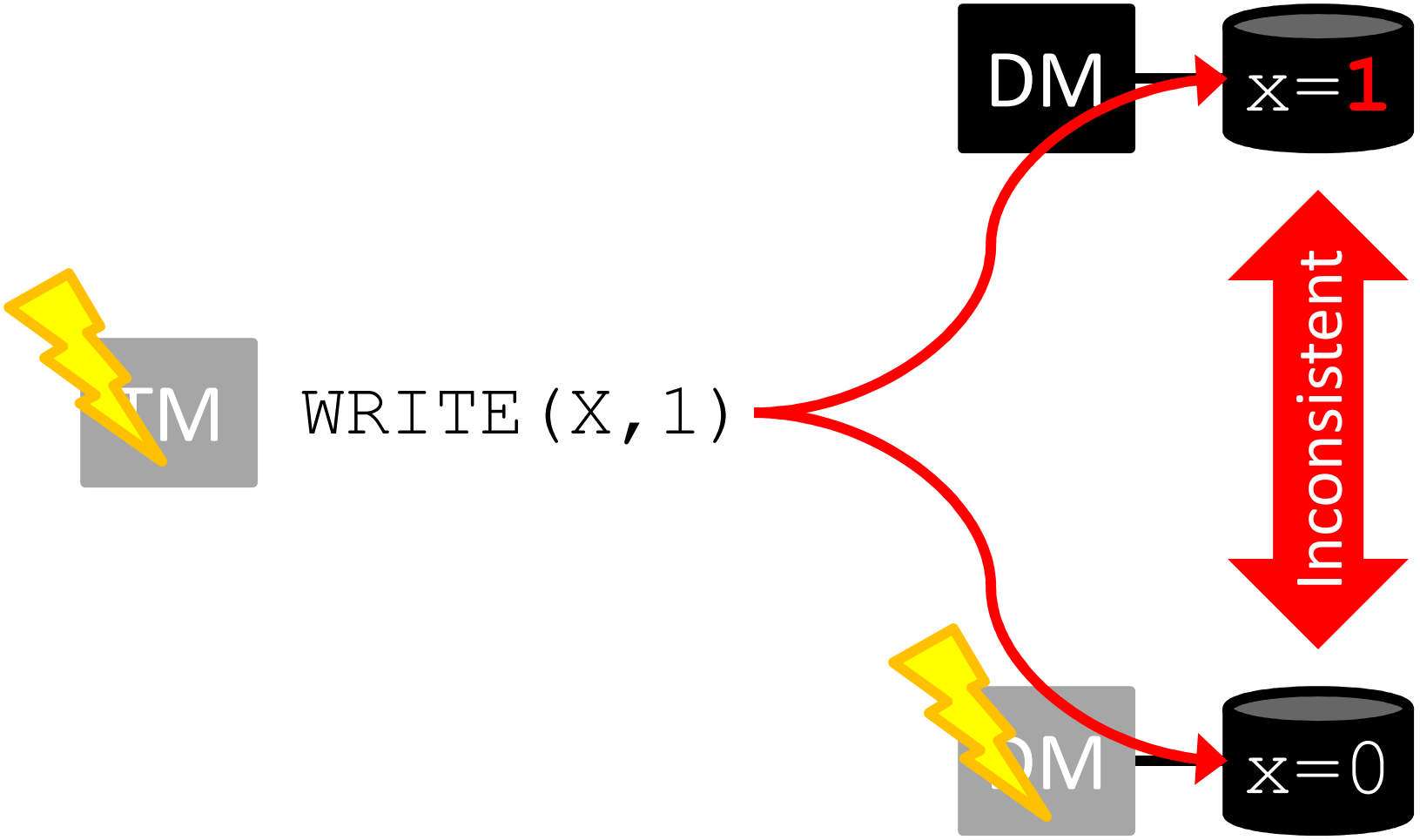
1PC: What can go wrong?



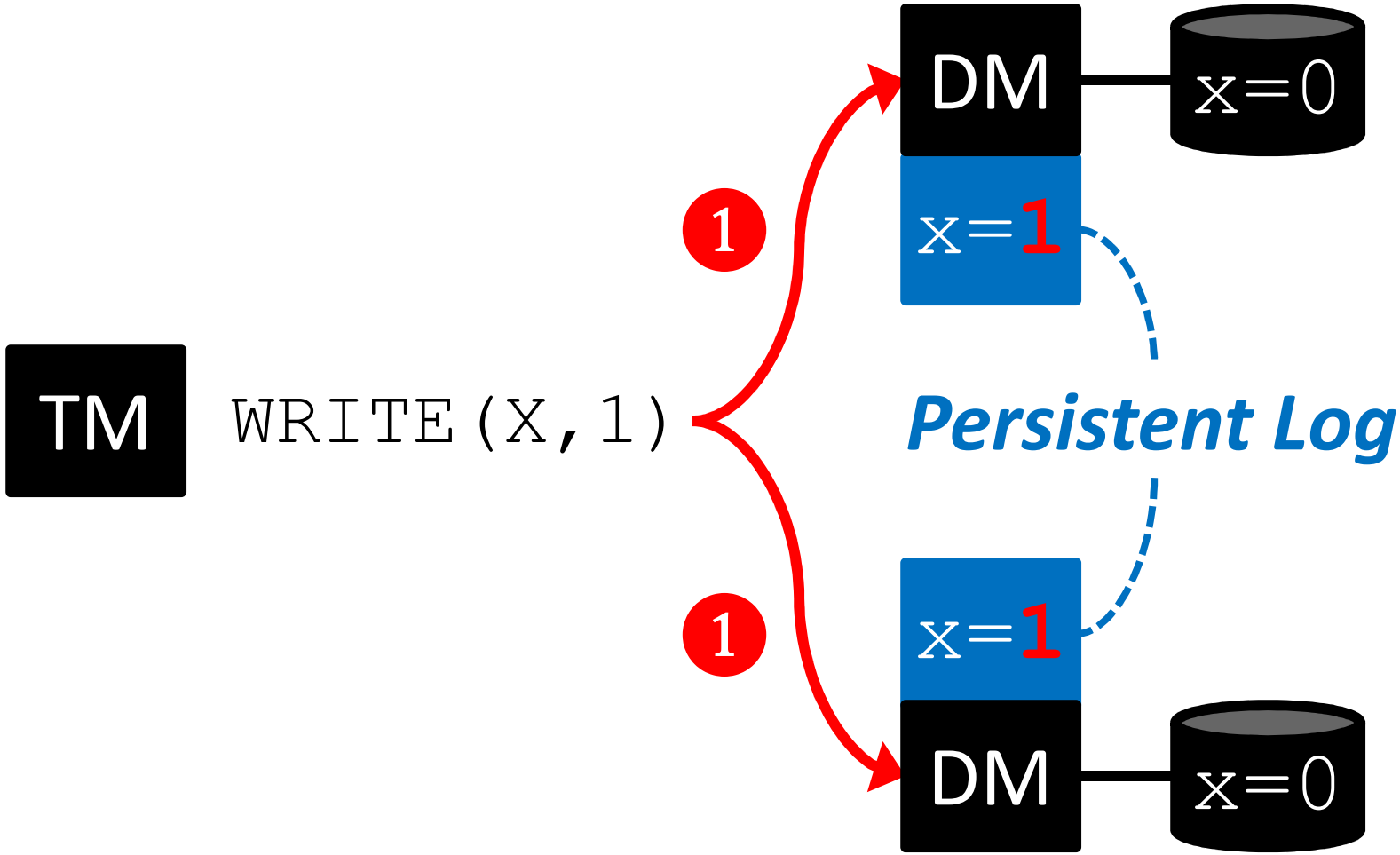
1PC: What can go wrong?



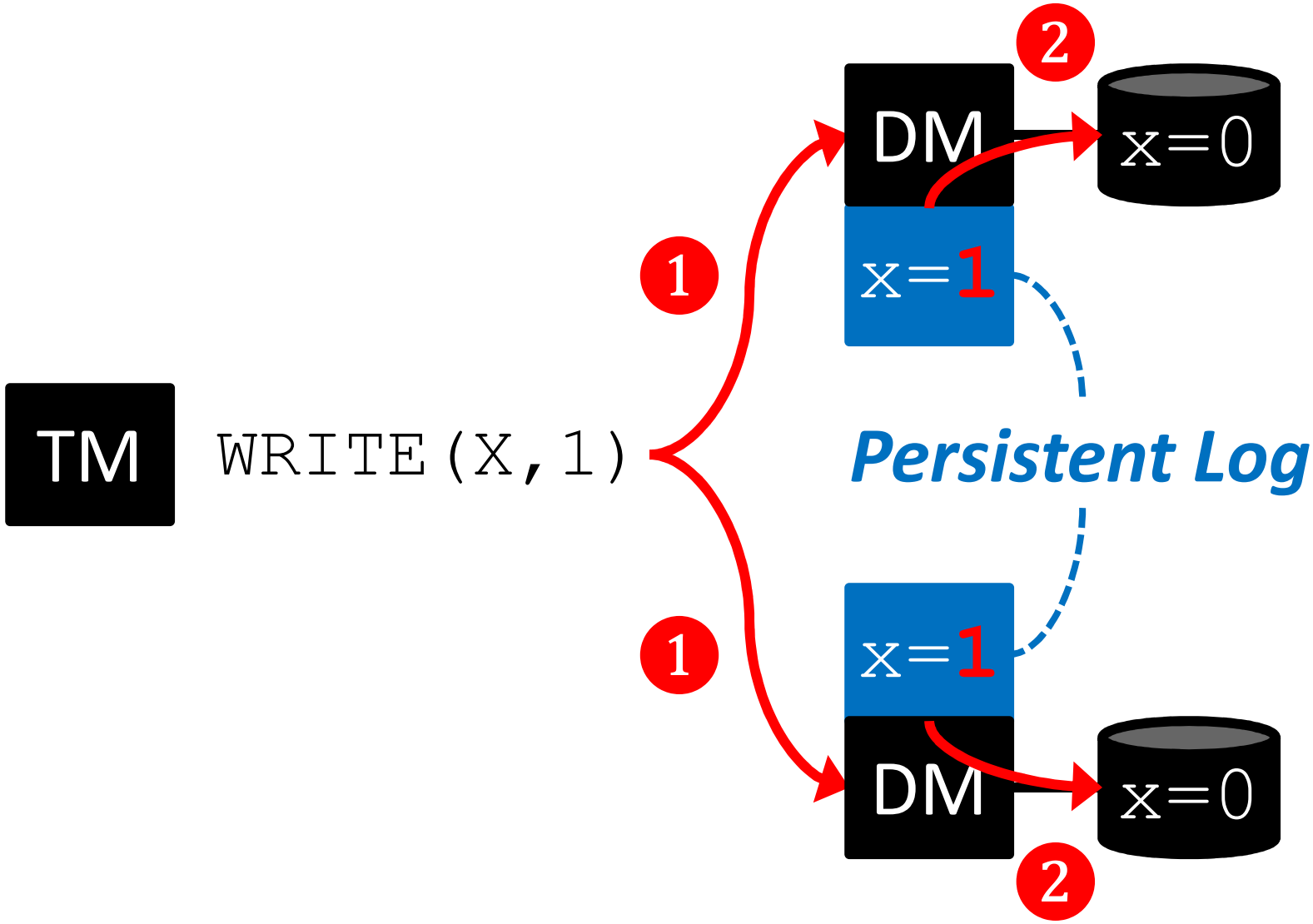
1PC: What can go wrong?



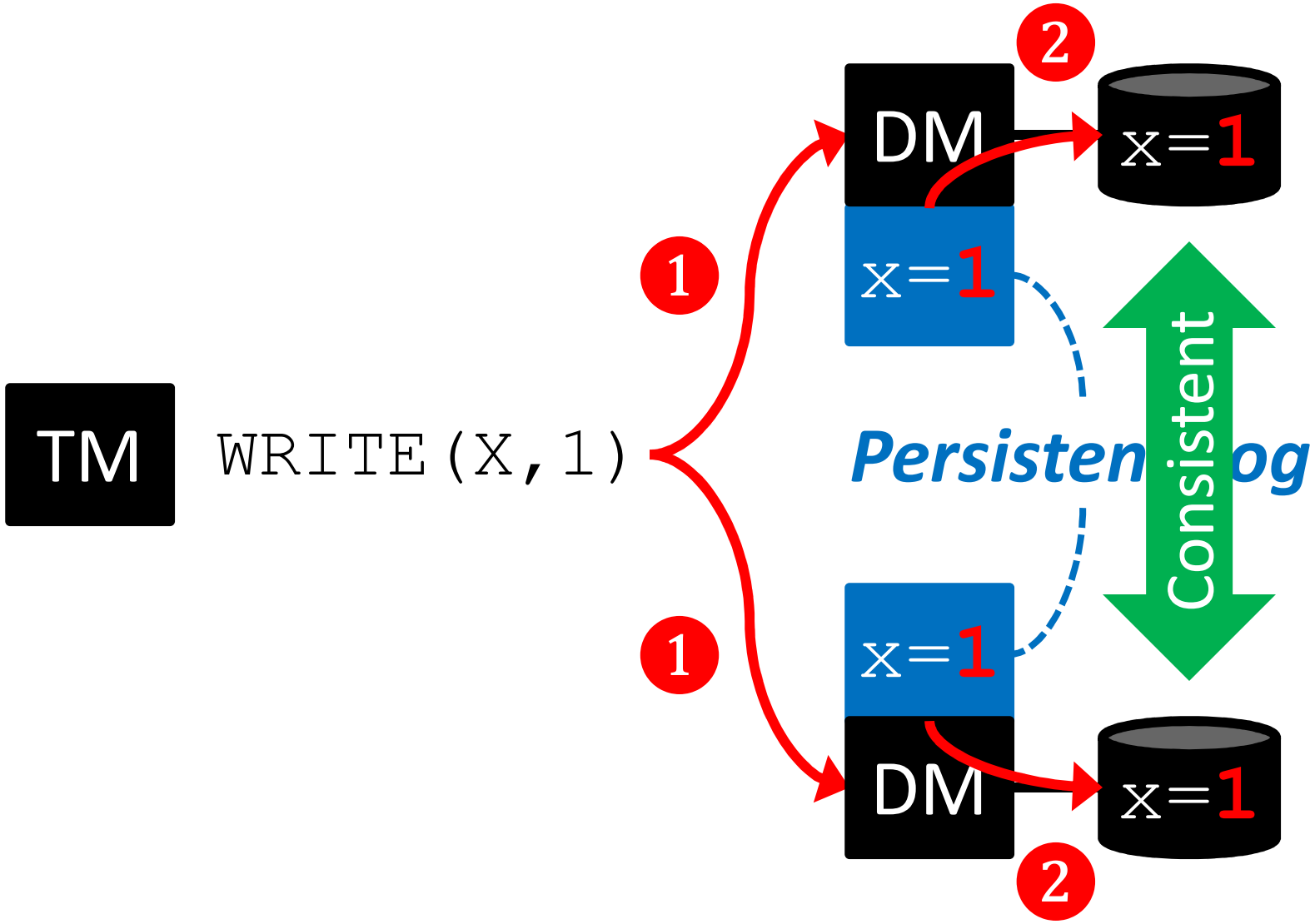
Two-Phase Commit (2PC)



Two-Phase Commit (2PC)



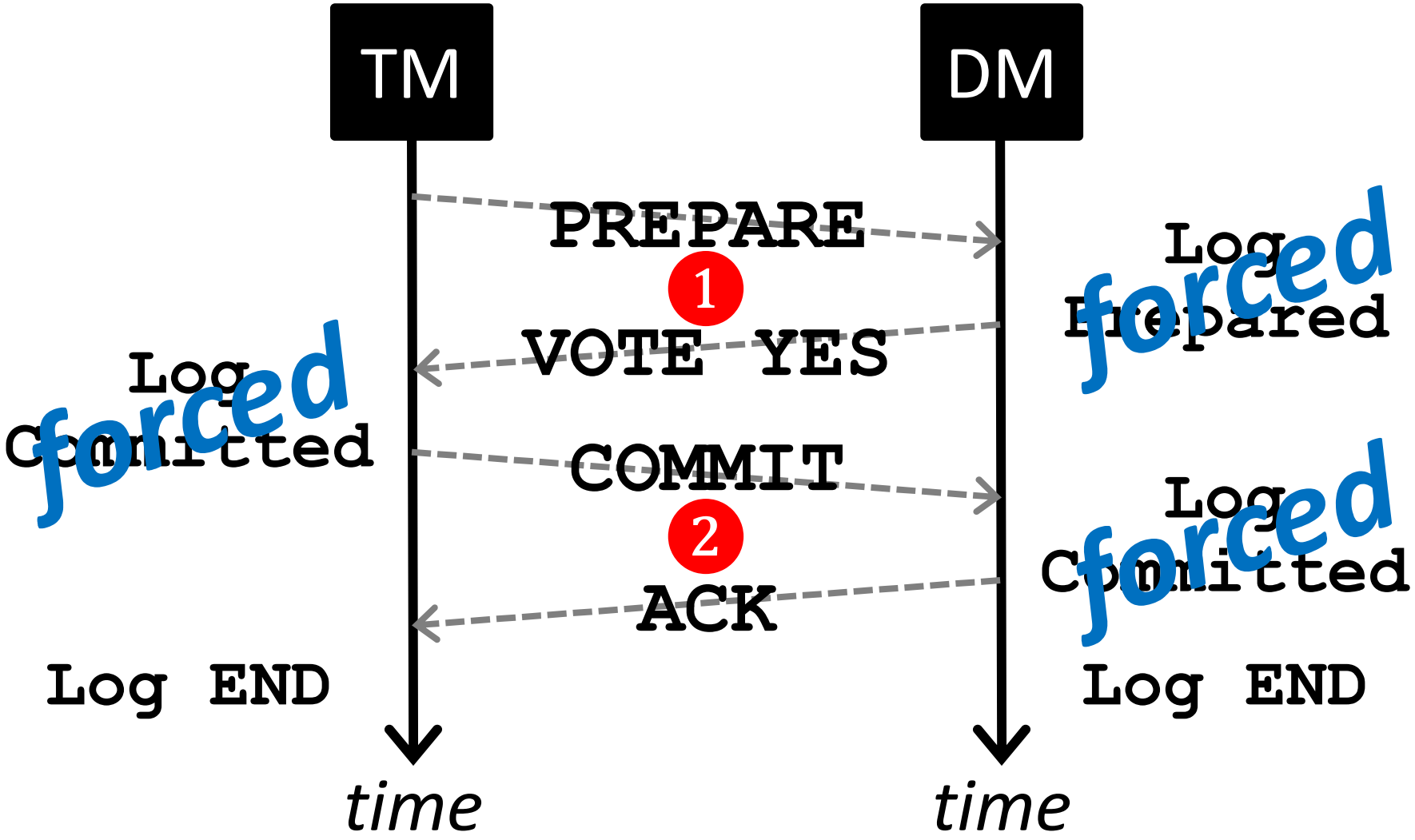
Two-Phase Commit (2PC)



Outline

- Two-Phase Commit (2PC)
- **Baseline 2PC**
- Optimized 2PC

A Closer Look at 2PC



Quantifying the Cost of 2PC

Assume ***N*** participants: ***1*** TM & ***N-1*** DMs

- **Network Traffic**
  - Each (TM, DM) pair ➔ 4 messages
  - Total Messages: *4(N-1)*
- **Log Traffic**
  - Each TM ➔ 1 forced, 1 unforced logs
  - Each DM ➔ 2 forced, 1 unforced logs
  - Total Forced Logs: *1 + 2(N-1) = 2N-1*
  - Total Unforced Logs: *1 + (N-1) = N*

expensive!

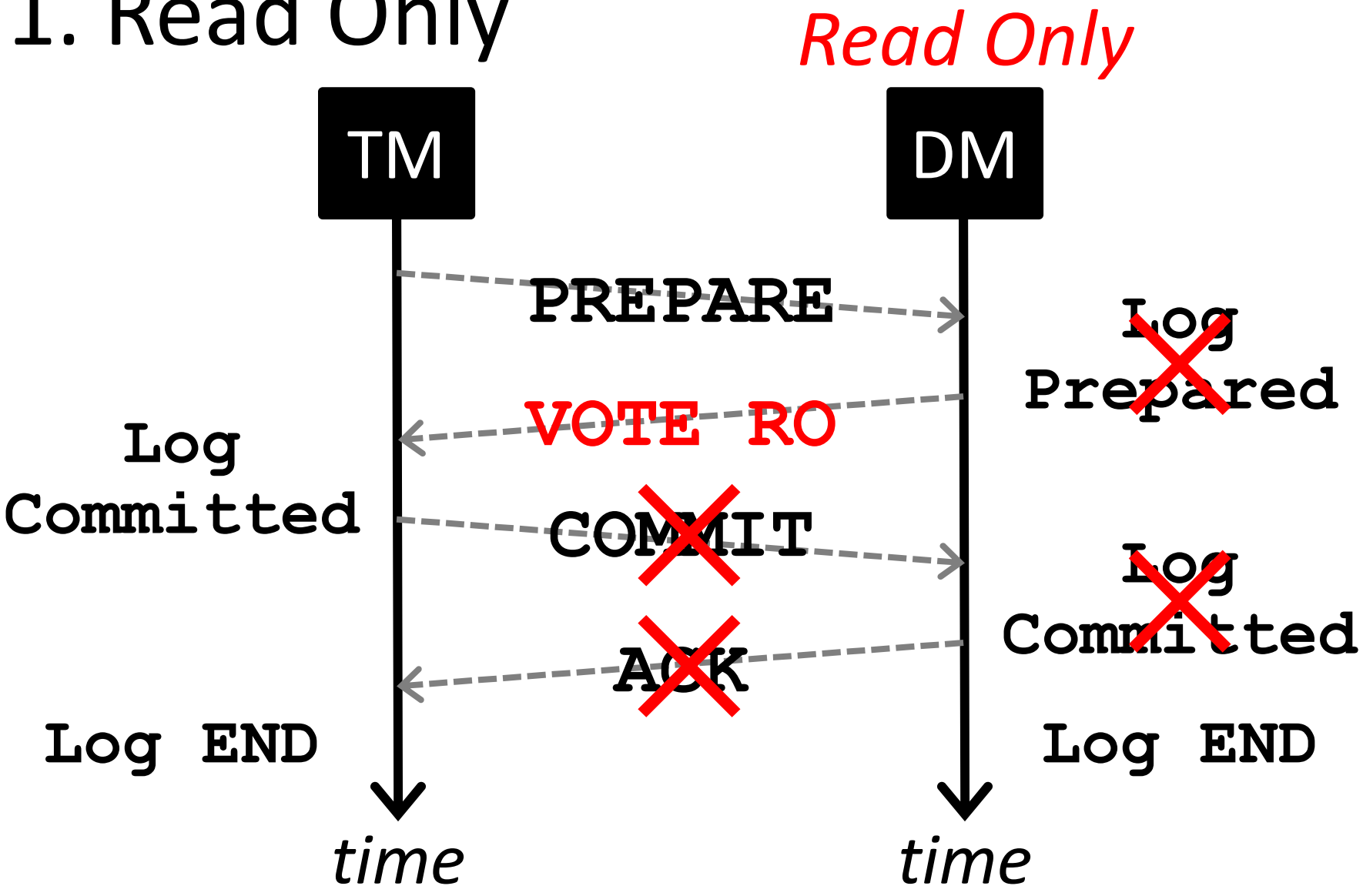
Outline

- Two-Phase Commit (2PC)
- Baseline 2PC
- **Optimized 2PC**

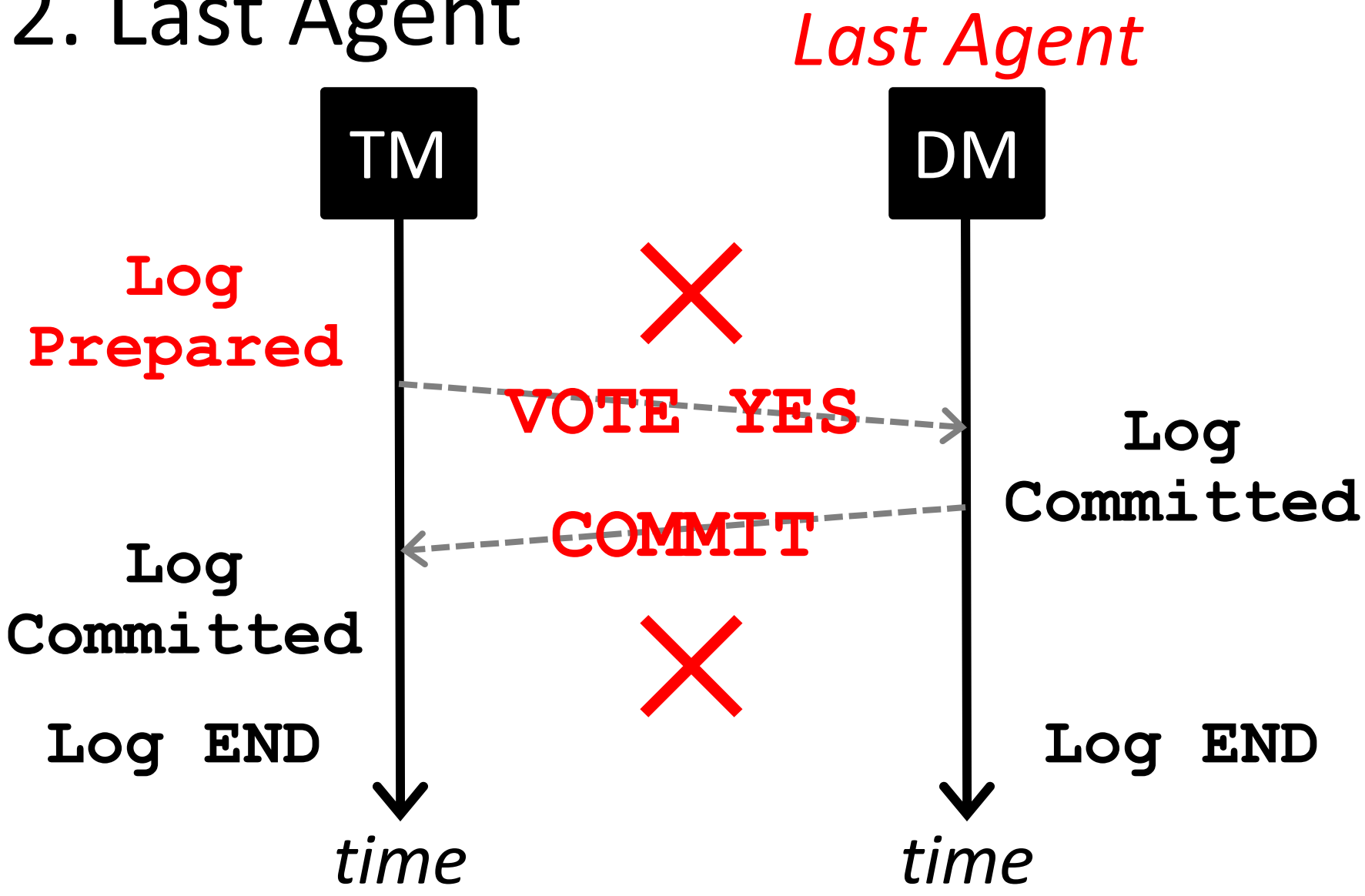
Optimizations to Baseline 2PC

1. **Read Only**
2. **Last Agent**
3. **Unsolicited Vote**
4. **Group Commits**
5. **Commit Acknowledgment**

1. Read Only

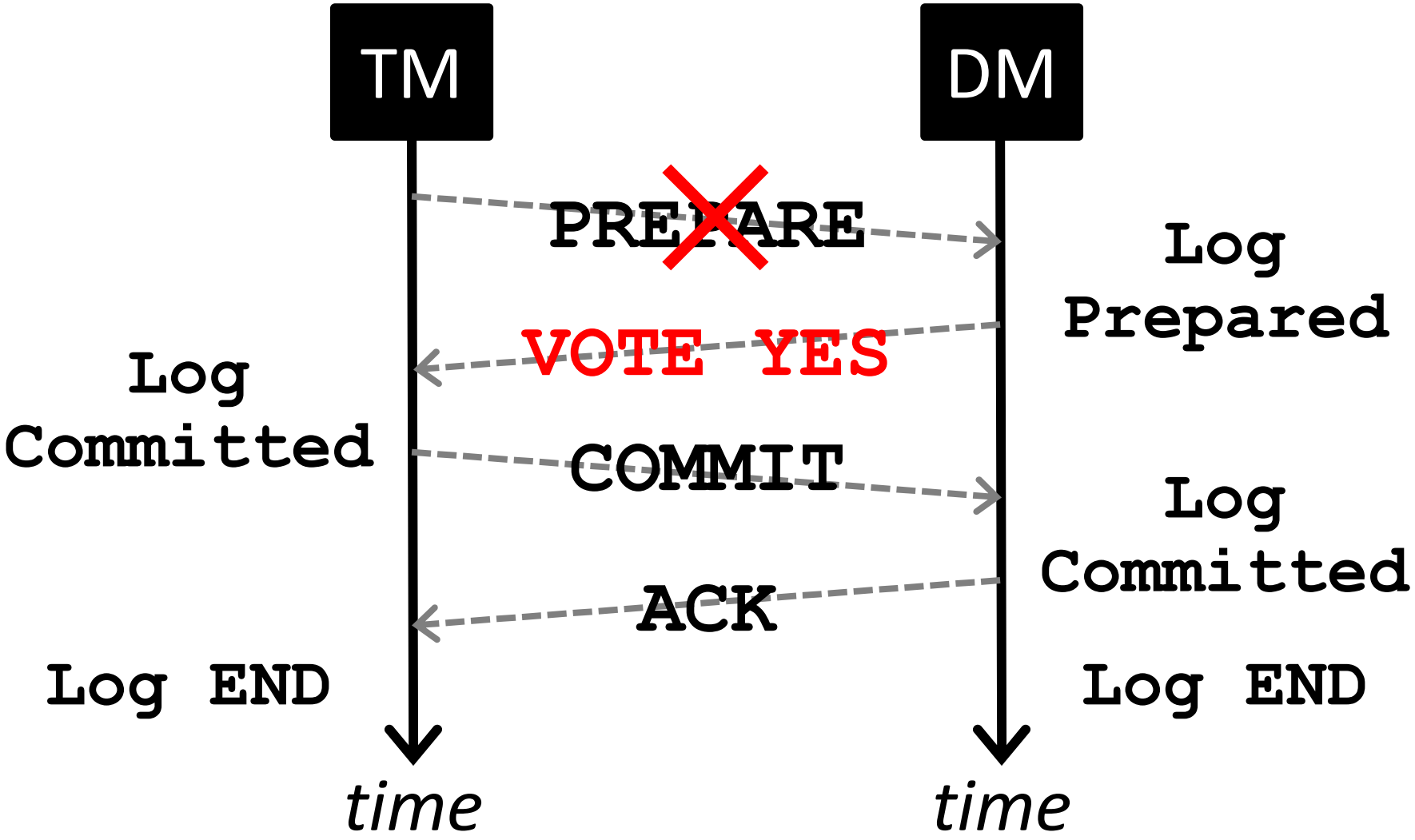

Read Only
TM
DM
PREPARE
Log
Prepared
VOTE RO
Log
Committed
COMMIT
Log
Committed
ACK
Log END
Log END
time
time

2. Last Agent


Last Agent
TM
DM
Log
Prepared
VOTE YES
Log
Committed
COMMIT
Log
Committed
Log END
Log END
time
time

3. Unsolicited Vote

"Smart"



4. Group Commits

- **Forced logs = Blocking I/O**
  - Commit process must wait until writes have been completely flushed out
  - Large number of writes can queue up in the I/O system

  ➔ *Large latency penalty*

- **Idea:** "Batch" multiple writes into one
  - Reduces number of writes

5. Commit Acknowledgment

When to send back an `ACK` message?

1. **Early Acknowledgment:** "Optimistic"
   - I have committed
   - My subordinates (other DMs) may still be committing
2. **Late Acknowledgment:** "Pessimistic"
   - I have committed
   - My subordinates (other DMs) have committed

End

Today's Papers

1. ***Concurrency Control in Distributed Database Systems (1981).***
   Bernstein & Goodman

2. ***Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment (1993).***
   Samaras, Britton, Citron, Mohan