Outline

- Introduction
- Background
- Distributed Database Design
- Database Integration
- Semantic Data Control
- Distributed Query Processing
- Multidatabase Query Processing
- Distributed Transaction Management
    - Transaction Concepts and Models
    - Distributed Concurrency Control
    - Distributed Reliability
- Data Replication
- Parallel Database Systems
- Distributed Object DBMS
- Peer-to-Peer Data Management
- Web Data Management
- Current Issues

Concurrency Control

- The problem of synchronizing concurrent transactions such that the consistency of the database is maintained while, at the same time, maximum degree of concurrency is achieved.
- Anomalies:
    - Lost updates
        - The effects of some transactions are not reflected on the database.
    - Inconsistent retrievals
        - A transaction, if it reads the same data item more than once, should always read the same value.

Execution History (or Schedule)

- An order in which the operations of a set of transactions are executed.
- A history (schedule) can be defined as a partial order over the operations of a set of transactions.

T_1:	Read(x)	T_2:	Write(x)	T_3:	Read(x)
	Write(x)		Write(y)		Read(y)
	Commit		Read(z)		Read(z)
			Commit		Commit

$$H_1=\{W_2(x),R_1(x), R_3(x),W_1(x),C_1,W_2(y),R_3(y),R_2(z),C_2,R_3(z),C_3\}$$

Formalization of History

A complete history over a set of transactions $T = \{T_1, \ldots, T_n\}$ is a partial order $H_c(T) = \{\Sigma_T, \prec_H\}$ where

❶ $\Sigma_T = \cup_i \Sigma_i$, for $i = 1, 2, \ldots, n$

❷ $\prec_H \supseteq \cup_i \prec_{T_i}$, for $i = 1, 2, \ldots, n$

❸ For any two conflicting operations $O_{ij}, O_{kl} \in \Sigma_T$, either $O_{ij} \prec_H O_{kl}$ or $O_{kl} \prec_H O_{ij}$

Complete Schedule – Example1

T_1: Read(x)
$x \leftarrow x+1$
Write(x)
Commit

T_2: Read(x)
$x \leftarrow x+1$
Write(x)
Commit

A possible complete history H_T^c over $T = \{T_1, T_2\}$ is the partial order $H_T^c = \{\Sigma_T, \prec_T\}$ where

$$\Sigma_1 = \{R_1(x), W_1(x), C_1\}$$
$$\Sigma_2 = \{R_2(x), W_2(x), C_2\}$$

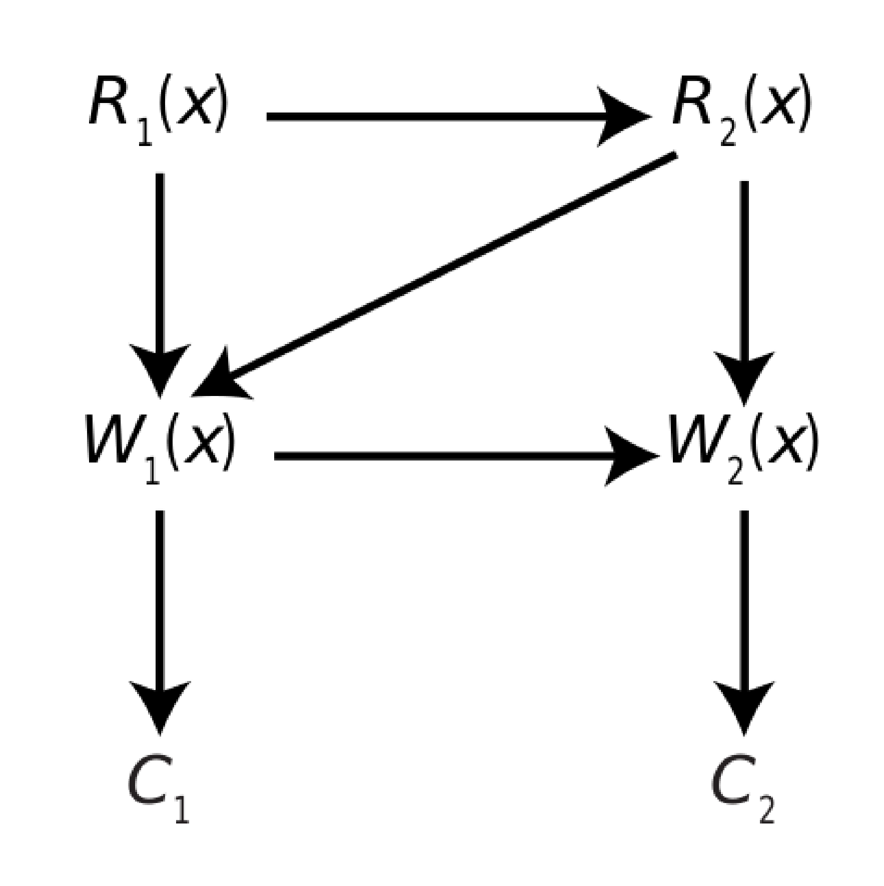
Thus

$$\Sigma_T = \Sigma_1 \cup \Sigma_2 = \{R_1(x), W_1(x), C_1, R_2(x), W_2(x), C_2\}$$

and

$$\prec_H = \{(R_1,R_2),(R_1,W_1),(R_1,C_1),(R_1,W_2),(R_1,C_2),(R_2,W_1),(R_2,C_1),(R_2,W_2),(R_2,C_2),(W_1,C_1),(W_1,W_2),(W_1,C_2),(C_1,W_2),(C_1,C_2),(W_2,C_2)\}$$

Complete Schedule – Example1



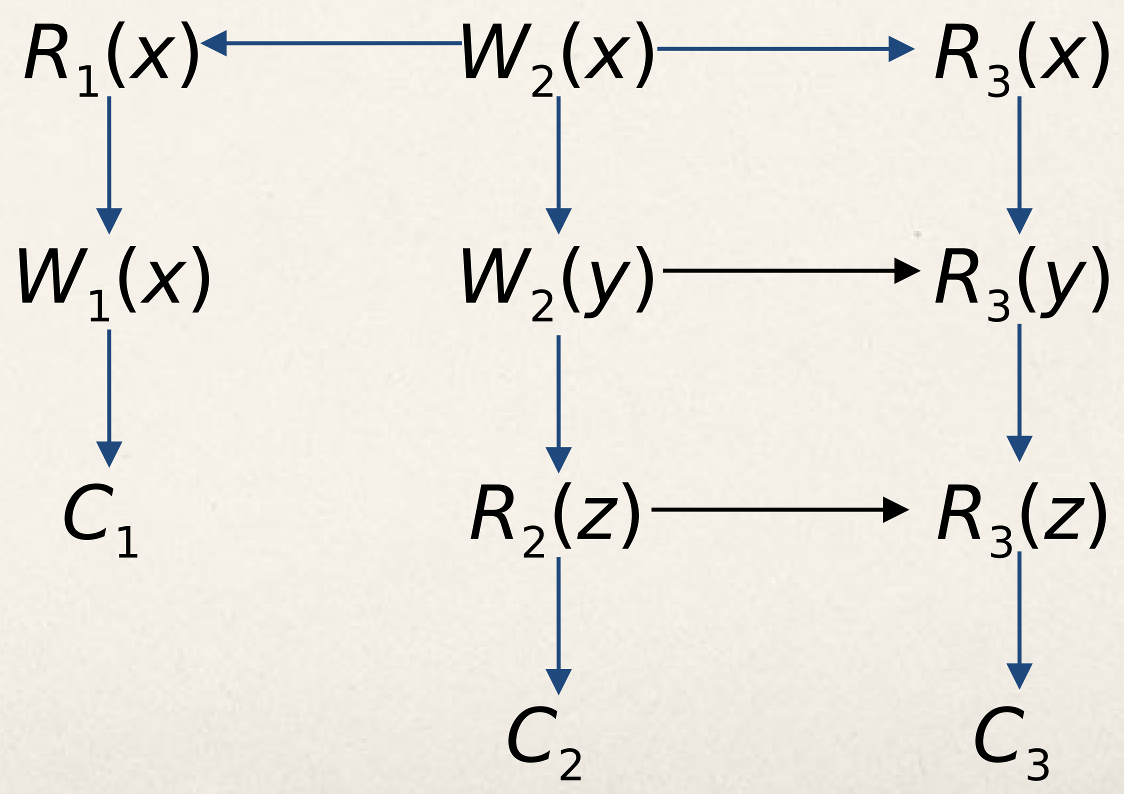
$$H_T^c = \{R_1(x), R_2(x), W_1(x), C_1, W_2(x), C_2\}$$

Complete Schedule – Example2

Given three transactions

T_1:	T_2:	T_3:
Read(x)	Write(x)	Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

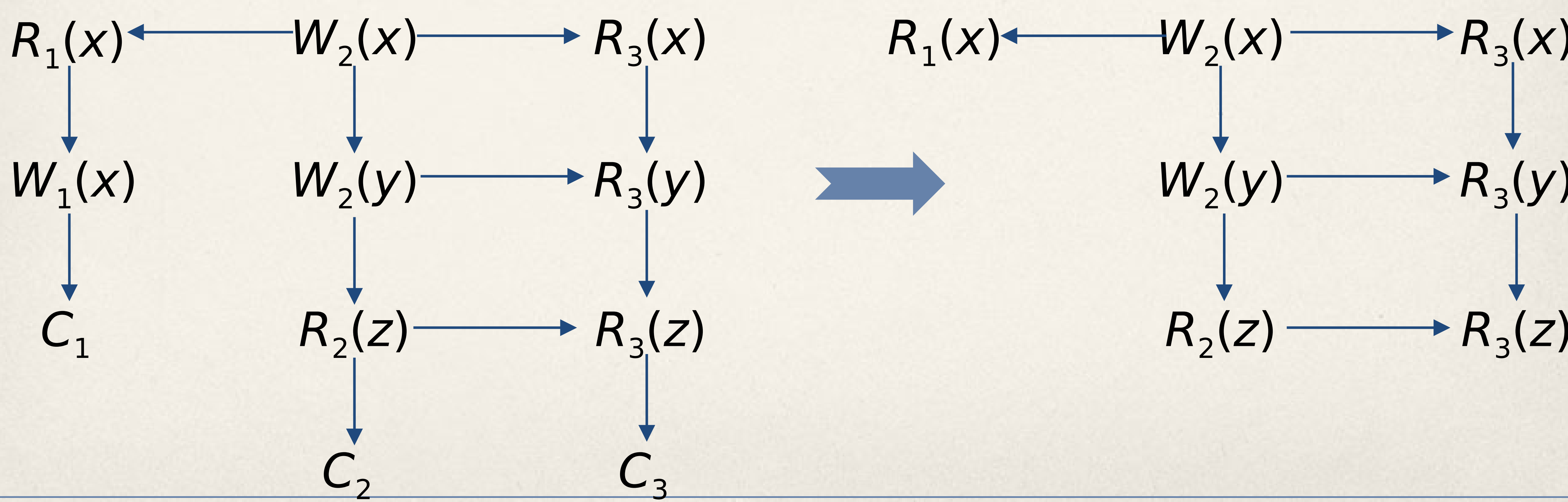
A possible complete schedule is given as the DAG



Schedule Definition

A schedule is a prefix of a complete schedule such that only some of the operations and only some of the ordering relationships are included.

T_1:	Read(x)	T_2:	Write(x)	T_3:	Read(x)
	Write(x)		Write(y)		Read(y)
	Commit		Read(z)		Read(z)
			Commit		Commit



Serial History

- All the actions of a transaction occur consecutively.
- No interleaving of transaction operations.
- If each transaction is consistent (obeys integrity rules), then the database is guaranteed to be consistent at the end of executing a serial history.

T_1:	T_2:	T_3:
Read(x)	Write(x)	Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

$$H = \{\underbrace{W_2(x), W_2(y), R_2(z)}_{T_2}, \underbrace{R_1(x), W_1(x)}_{T_1}, \underbrace{R_3(x), R_3(y), R_3(z)}_{T_3}\}$$

Serializable History

- Transactions execute concurrently, but the net effect of the resulting history upon the database is equivalent to some serial history.
- Equivalent with respect to what?

  Conflict equivalence: the relative order of execution of the conflicting operations belonging to unaborted transactions in two histories are the same.

  Conflicting operations: two incompatible operations (e.g., Read and Write) conflict if they both access the same data item.

  - Incompatible operations of each transaction is assumed to conflict; do not change their execution orders.
  - If two operations from two different transactions conflict, the corresponding transactions are also said to conflict.

Serializable History

T_1: Read(x)	T_2: Write(x)	T_3: Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

The following are not conflict equivalent

$$H_s=\{W_2(x),W_2(y),R_2(z),R_1(x),W_1(x),R_3(x),R_3(y),R_3(z)\}$$

$$H_1=\{W_2(x),R_1(x), R_3(x),W_1(x),W_2(y),R_3(y),R_2(z),R_3(z)\}$$

The following are conflict equivalent; therefore H_2 is *serializable*.

$$H_s=\{W_2(x),W_2(y),R_2(z),R_1(x),W_1(x),R_3(x),R_3(y),R_3(z)\}$$

$$H_2=\{W_2(x),R_1(x),W_1(x),R_3(x),W_2(y),R_3(y),R_2(z),R_3(z)\}$$

Serializability in Distributed DBMS

- Somewhat more involved. Two histories have to be considered:
  - local histories
  - global history
- For global transactions (i.e., global history) to be serializable, two conditions are necessary:
  - Each local history should be serializable.
  - Two conflicting operations should be in the same relative order in all of the local histories where they appear together.

Global Non-serializability

T_1:	Read(x)	T_2:	Read(x)
	$x \leftarrow x$-100		Read(y)
	Write(x)		Commit
	Read(y)		
	$y \leftarrow y$+100		
	Write(y)		
	Commit		

- x stored at Site 1, y stored at Site 2
- LH_1, LH_2 are individually serializable (in fact serial), but the two transactions are not globally serializable.

$$LH_1=\{R_1(x),W_1(x), R_2(x)\}$$

$$LH_2=\{R_2(y), R_1(y),W_1(y)\}$$

Concurrency Control Algorithms

- Pessimistic
  - Two-Phase Locking-based (2PL)
    - Centralized (primary site) 2PL
    - Primary copy 2PL
    - Distributed 2PL
  - Timestamp Ordering (TO)
    - Basic TO
    - Multiversion TO
    - Conservative TO
  - Hybrid
- Optimistic
  - Locking-based
  - Timestamp ordering-based

Locking-Based Algorithms

- Transactions indicate their intentions by requesting locks from the scheduler (called lock manager).
- Locks are either read lock (*rl*) [also called shared lock] or write lock (*wl*) [also called exclusive lock]
- Read locks and write locks conflict (because Read and Write operations are incompatible

	rl	*wl*
rl	yes	no
wl	no	no

- Locking works nicely to allow concurrent processing of transactions.

Naive Locking Algorithm

T_1: Read(x)	T_2: Read(x)
$x \leftarrow x+1$	$x \leftarrow x*2$
Write(x)	Write(x)
Read(y)	Read(y)
$y \leftarrow y-1$	$y \leftarrow y*2$
Write(y)	Write(y)
Commit	Commit

The following is a valid history that a lock manager employing the locking algorithm may generate:

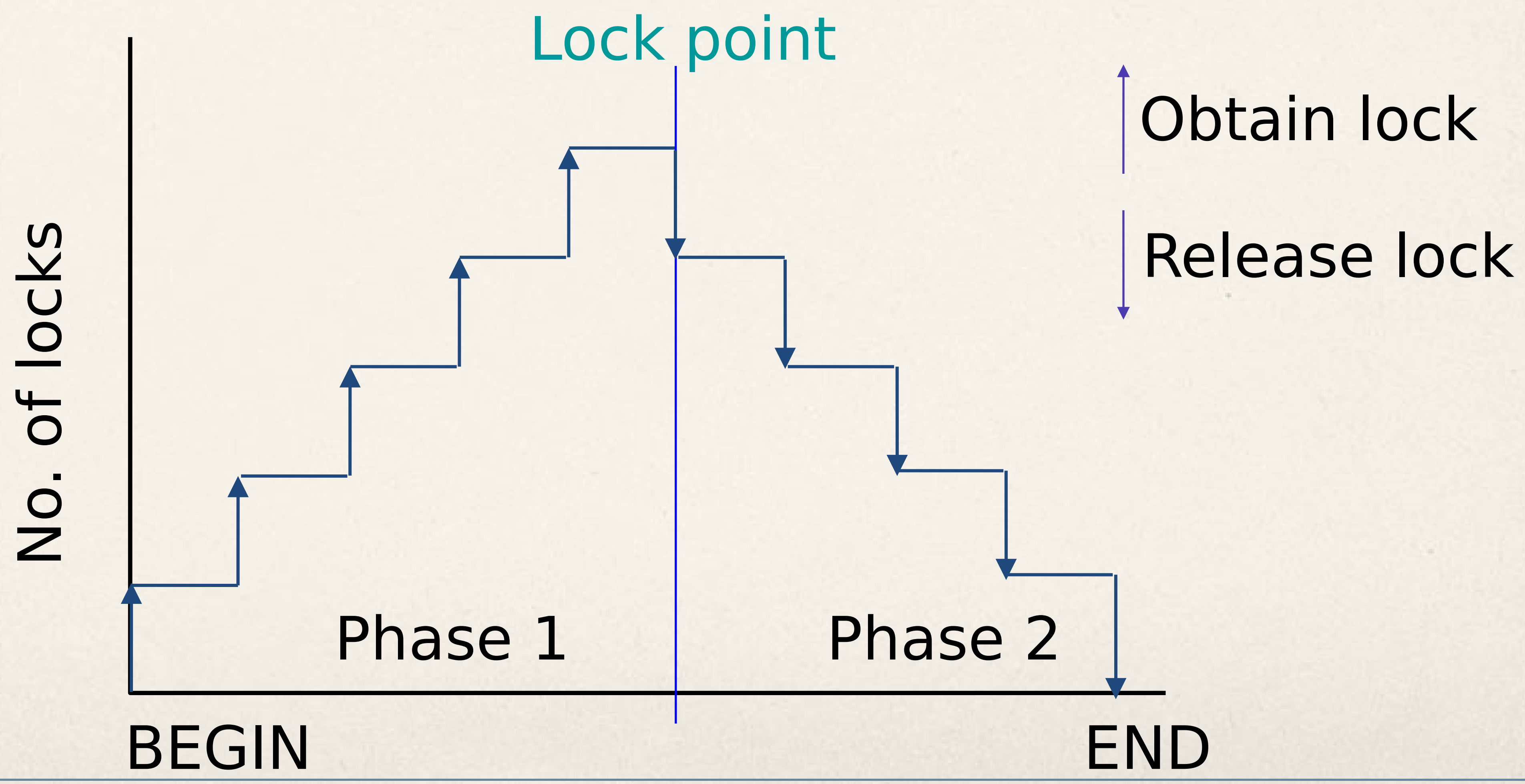
$$H = \{wl_1(x), R_1(x), W_1(x), lr_1(x), wl_2(x), R_2(x), w_2(x), lr_2(x), wl_2(y), R_2(y), W_2(y), lr_2(y), wl_1(y), R_1(y), W_1(y), lr_1(y)\}$$

Naive Locking Algorithm

- The locking algorithm releases the locks that are held by a transaction (say, T_i) as soon as the associated database command (read or write) is executed.
  - The transaction itself is locking other items (say, y), after it releases its lock on x.
- This may seem to be advantageous from the viewpoint of increased concurrency
  - It permits transactions to interfere with one another
  - Loss of isolation and atomicity

Two-Phase Locking (2PL)

1. A Transaction locks an object before using it.
2. When an object is locked by another transaction, the requesting transaction must wait.
3. When a transaction releases a lock, it may not request another lock.

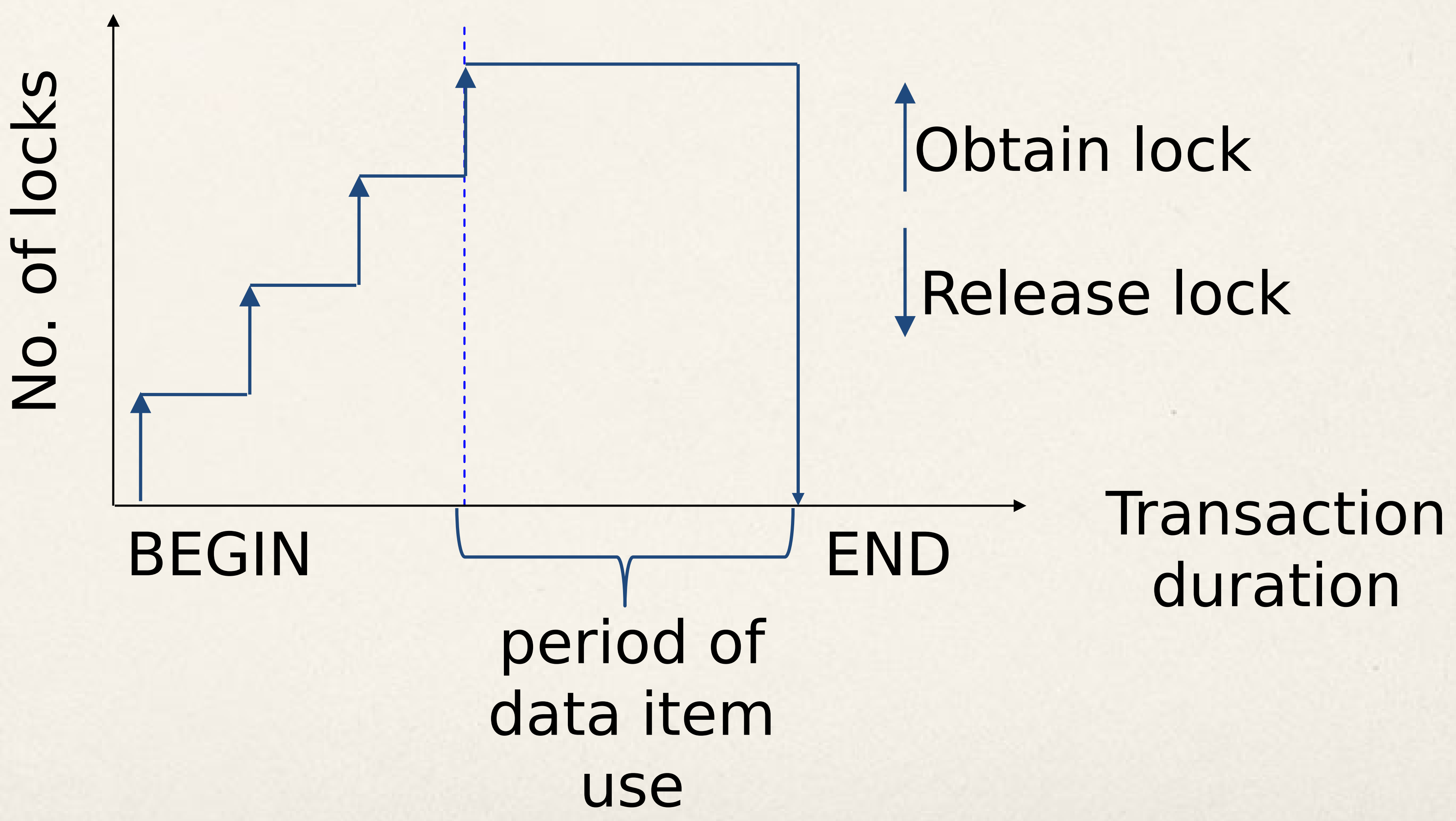


Two-Phase Locking (2PL)

- Two-phase locking rule simply states that no transaction should request a lock after it releases one of its locks!
- 2PL algorithms execute transactions in two phases.
  - growing phase: it obtains locks and accesses data items, and
  - a shrinking phase, during which it releases locks
- Lock point
  - when the transaction has achieved all its locks
  - End of the growing phase, beginning of the shrinking phase of a transaction.
- It has been proven that any history generated by a concurrency control algorithm that obeys the 2PL rule is serializable

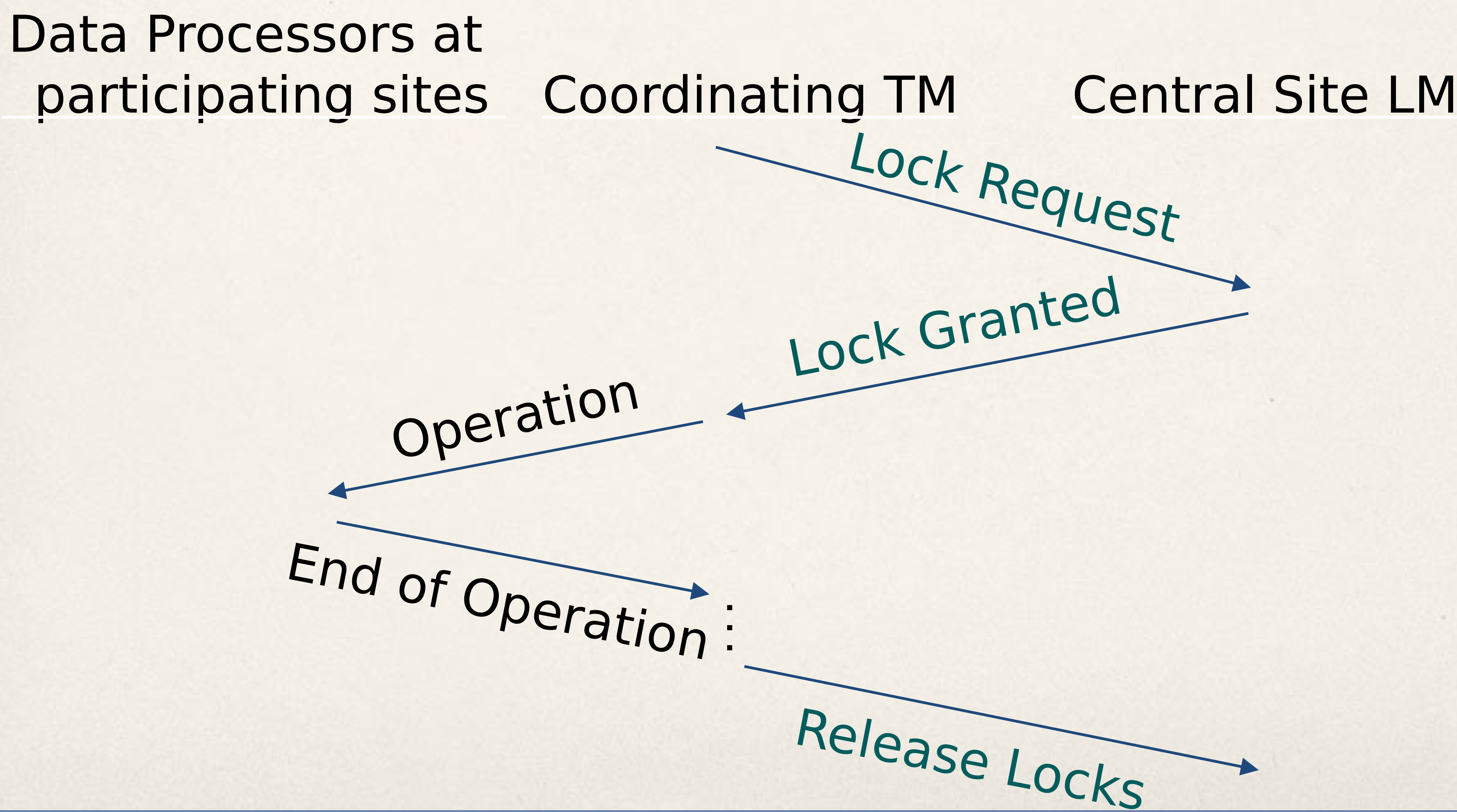
Strict 2PL

Hold locks until the end.



Centralized 2PL

- There is only one 2PL scheduler in the distributed system.
- Lock requests are issued to the central scheduler.



C2PL-TM

Algorithm 11.1: Centralized 2PL Transaction Manager (C2PL-TM) Algorithm

```
Input: msg : a message
begin
    repeat
        wait for a msg ;
        switch msg do
            case transaction operation
                let op be the operation ;
                if op.Type = BT then DP(op)          {call DP with operation}
                else C2PL-LM(op)                     {call LM with operation}
            case Lock Manager response          {lock request granted or locks
            released}
                if lock request granted then
                    find site that stores the requested data item (say H_i) ;
                    DP_Si(op)                   {call DP at site S_i with operation}
                else                            {must be lock release message}
                    inform user about the termination of transaction

            case Data Processor response        {operation completed message}
                switch transaction operation do
                    let op be the operation ;
                    case R
                        return op.val (data item value) to the application
                    case W
                        inform application of completion of the write
                    case C
                        if commit msg has been received from all participants
                        then
                            inform application of successful completion of
                            transaction ;
                            C2PL-LM(op)                 {need to release locks}
                        else        {wait until commit messages come from all}
                            record the arrival of the commit message

                    case A
                        inform application of completion of the abort ;
                        C2PL-LM(op)                     {need to release locks}
    until forever ;
end
```

C2PL-LM

Algorithm 11.2: Centralized 2PL Lock Manager (C2PL-LM) Algorithm

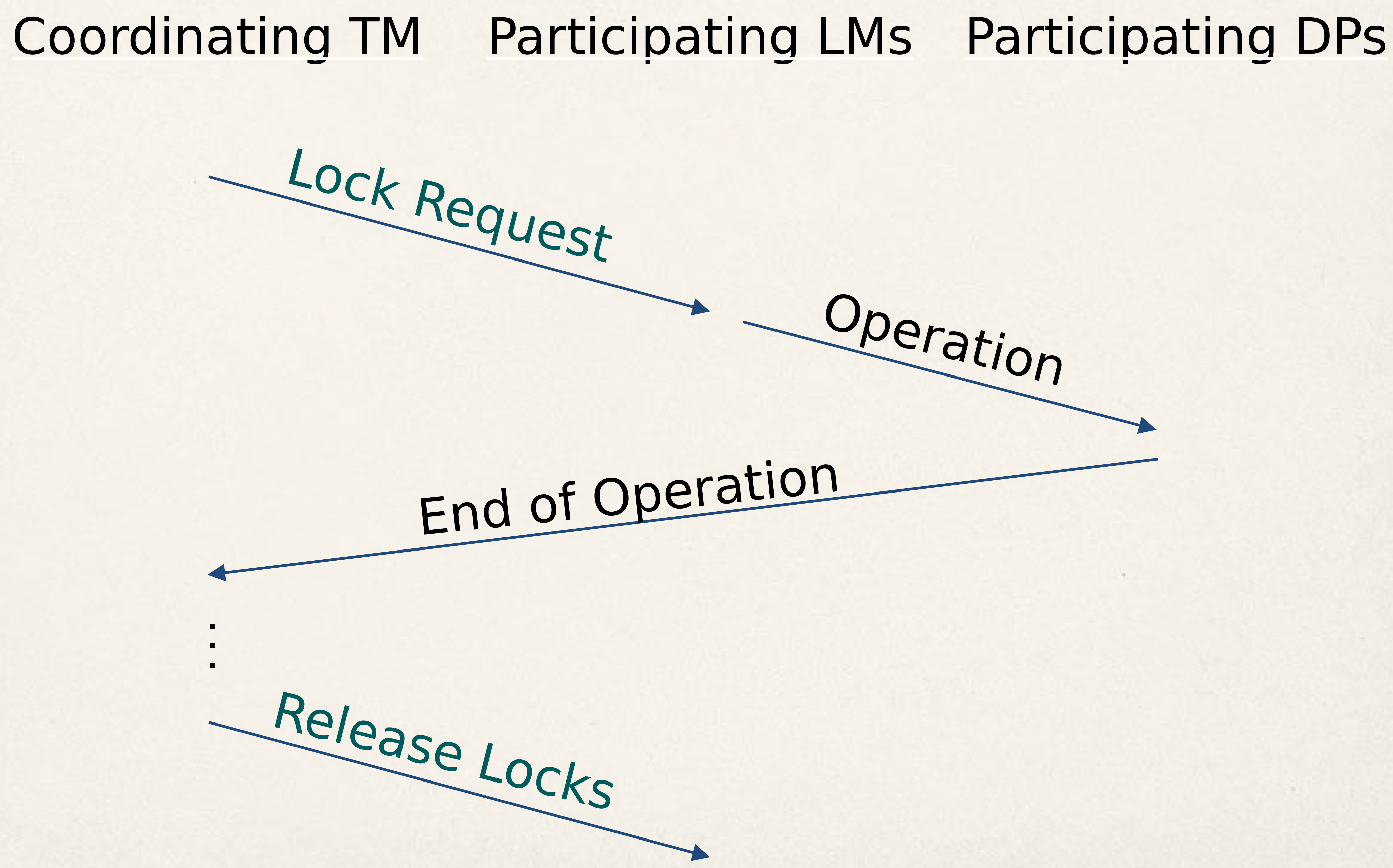
```
Input: op : Op
begin
    switch op.Type do
        case R or W                                {lock request; see if it can be granted}
            find the lock unit lu such that op.arg ⊆ lu ;
            if lu is unlocked or lock mode of lu is compatible with op.Type
            then
                set lock on lu in appropriate mode on behalf of transaction
                op.tid ;
                send "Lock granted" to coordinating TM of transaction
            else
                put op on a queue for lu

        case C or A                                {locks need to be released}
            foreach lock unit lu held by transaction do
                release lock on lu held by transaction ;
                if there are operations waiting in queue for lu then
                    find the first operation O on queue ;
                    set a lock on lu on behalf of O ;
                    send "Lock granted" to coordinating TM of transaction
                    O.tid
            send "Locks released" to coordinating TM of transaction
end
```

Data Processor

Algorithm 11.3: Data Processor (DP) Algorithm

```
Input: op : Op
begin
    switch op.Type do                                    {check the type of operation}
        case BT                              {details to be discussed in Chapter 12}
            do some bookkeeping
        case R
            op.res ← READ(op.arg) ;                          {database READ operation}
            op.res ← "Read done"
        case W                        {database WRITE of val into data item arg}
            WRITE(op.arg, op.val) ;
            op.res ← "Write done"
        case C
            COMMIT ;                                              {execute COMMIT }
            op.res ← "Commit done"
        case A
            ABORT ;                                                {execute ABORT }
            op.res ← "Abort done"
    return op
end
```

Distributed 2PL Execution



Distributed 2PL

- The distributed 2PL TM algorithm is similar to the C2PL-TM
- TM-s and 2PL schedulers are placed at each site.
  - Each scheduler handles lock requests for data at that site.
- Major modifications:
  - The messages that are sent to the central site LM in C2PL-TM
    - Sent to LM-s at all participating sites in D2PL-TM
  - Operations are not passed to the DP-s by the coordinating TM
    - Set to DP-s by the participating lock managers
    - Coordinating TM does not wait for a “lock request granted”

Distributed 2PL

- The participating DP-s send the “end of operation” messages to the coordinating TM
  - The alternative is for each DP to send it to its own lock manager who can then release the locks and inform the coordinating TM
- In case of replication:
  - A transaction may read any of the replicated copies of item *x*, by obtaining a read lock on one of the copies of *x*.
  - Writing into *x* requires obtaining write locks for all copies of *x*.

Timestamp Ordering

❶ Transaction (T_i) is assigned a globally unique timestamp $ts(T_i)$.

❷ Transaction manager attaches the timestamp to all operations issued by the transaction.

❸ Each data item is assigned a write timestamp (*wts*) and a read timestamp (*rts*):

$rts(x)$ = largest timestamp of any read on x

$wts(x)$ = largest timestamp of any read on x

❹ Conflicting operations are resolved by timestamp order.

Basic T/O:

for $R_i(x)$	for $W_i(x)$
if $ts(T_i) < wts(x)$	**if** $ts(T_i) < rts(x)$ **and** $ts(T_i) < wts(x)$
then reject $R_i(x)$	**then** reject $W_i(x)$
else accept $R_i(x)$	**else** accept $W_i(x)$
$rts(x) \leftarrow ts(T_i)$	$wts(x) \leftarrow ts(T_i)$

Conservative Timestamp Ordering

- Basic timestamp ordering tries to execute an operation as soon as it receives it
  - progressive
  - too many restarts since there is no delaying
- Conservative timestamping delays each operation until there is an assurance that it will not be restarted
- Assurance?
  - No other operation with a smaller timestamp can arrive at the scheduler
  - Note that the delay may result in the formation of deadlocks

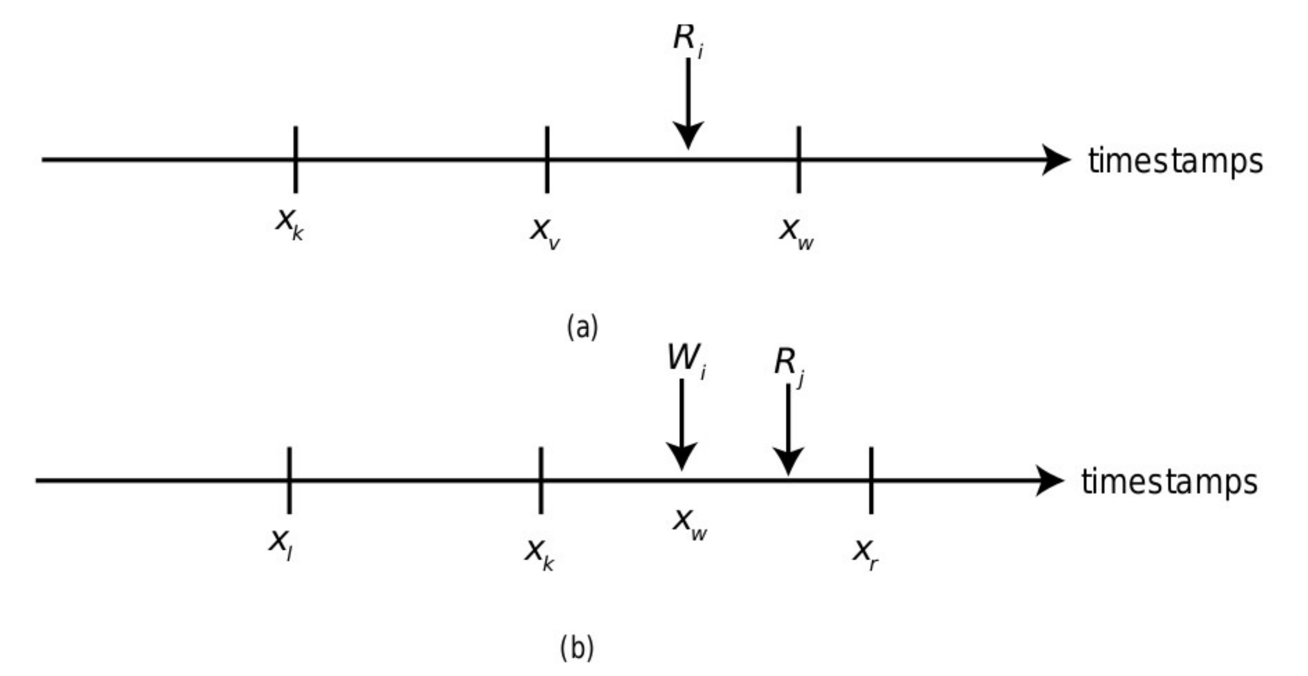
Multiversion Timestamp Ordering

- Do not modify the values in the database, create new values.
- A $R_i(x)$ is translated into a read on one version of x.

  Find a version of x (say x_v) such that $ts(x_v)$ is the largest timestamp less than $ts(T_i)$.

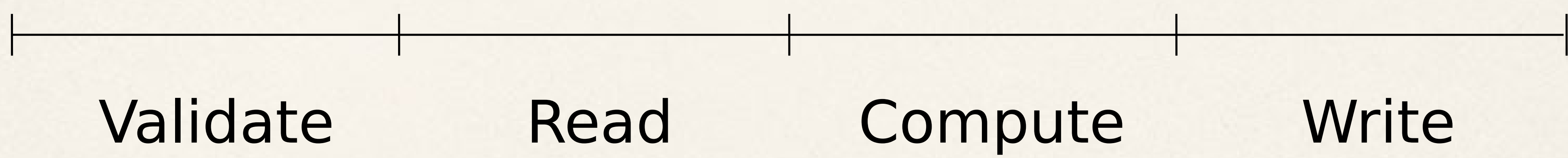
- A $W_i(x)$ is translated into $W_i(x_w)$ and accepted if the scheduler has not yet processed any $R_j(x_r)$ such that

$$ts(T_i) < ts(x_r) < ts(T_j)$$

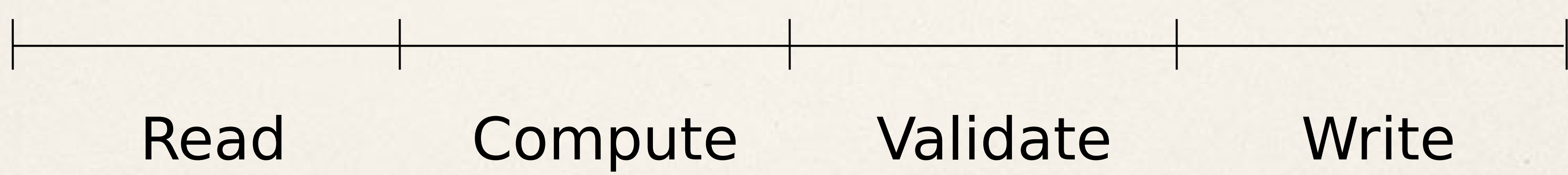


Optimistic Concurrency Control Algorithms

Pessimistic execution



Optimistic execution



Optimistic Concurrency Control Algorithms

- Transaction execution model: divide into subtransactions each of which execute at a site

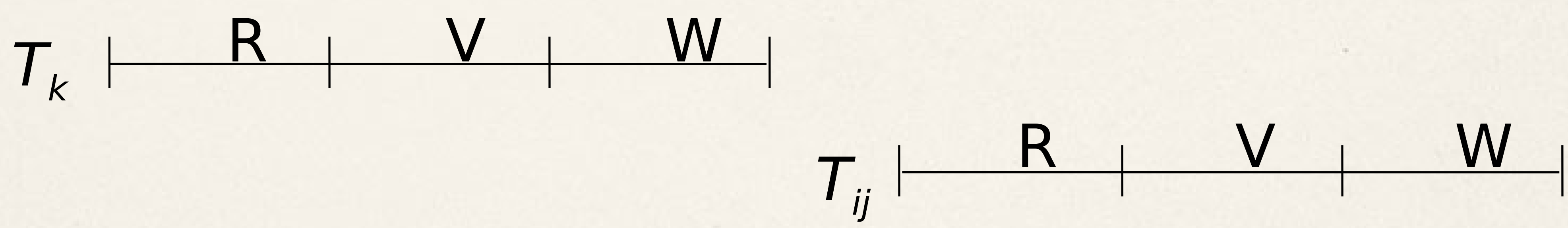
  T_{ij}: transaction T_i that executes at site j

- Transactions run independently at each site until they reach the end of their read phases
- All subtransactions are assigned a timestamp at the end of their read phase
- Validation test performed during validation phase. If one fails, all rejected.

Optimistic CC Validation Test

❶ If all transactions T_k where $ts(T_k) < ts(T_{ij})$ have completed their write phase before T_{ij} has started its read phase, then validation succeeds

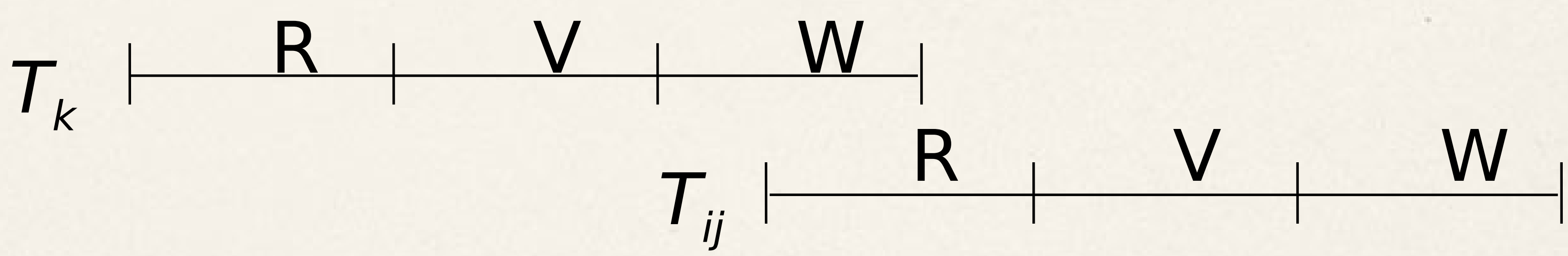
Transaction executions in serial order



Optimistic CC Validation Test

- ❷ If there is any transaction T_k such that $ts(T_k)<ts(T_{ij})$ and which completes its write phase while T_{ij} is in its read phase, then validation succeeds if $WS(T_k) \cap RS(T_{ij}) = \emptyset$

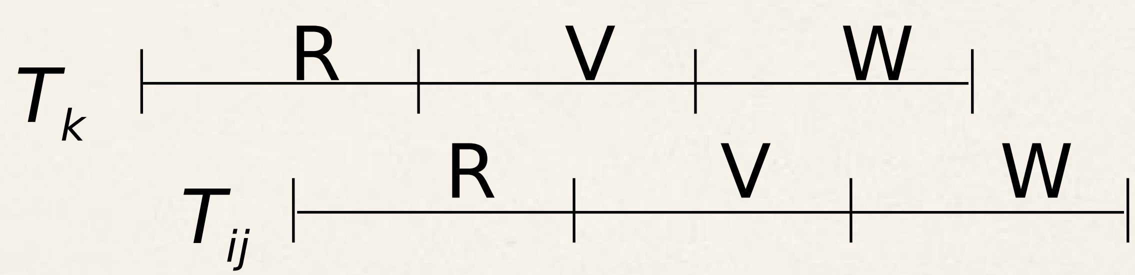
    Read and write phases overlap, but T_{ij} does not read data items written by T_k



Optimistic CC Validation Test

❸ If there is any transaction T_k such that $ts(T_k) < ts(T_{ij})$ and which completes its read phase before T_{ij} completes its read phase, then validation succeeds if $WS(T_k) \cap RS(T_{ij}) = Ø$ and $WS(T_k) \cap WS(T_{ij}) = Ø$

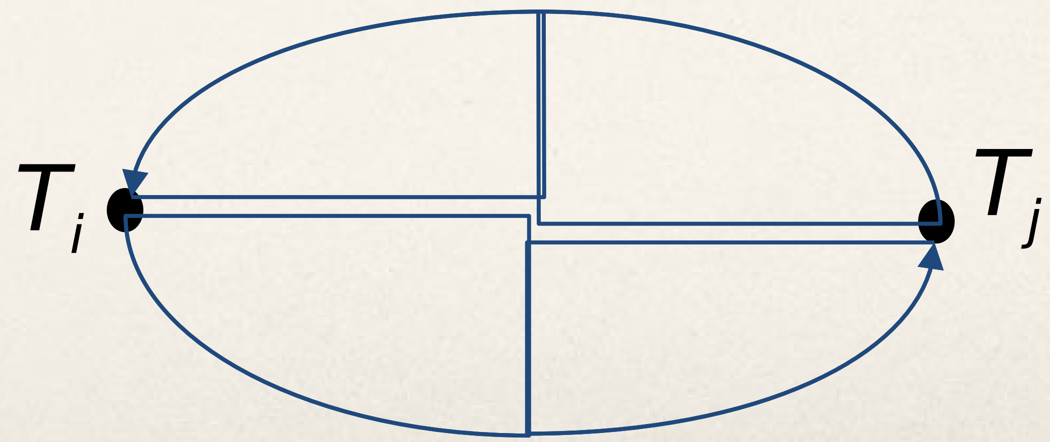
They overlap, but don't access any common data items.



Deadlock

- A transaction is deadlocked if it is blocked and will remain blocked until there is intervention.
- Locking-based CC algorithms may cause deadlocks.
- TO-based algorithms that involve waiting may cause deadlocks.
- Wait-for graph

    If transaction T_i waits for another transaction T_j to release a lock on an entity, then $T_i \rightarrow T_j$ in WFG.



Local versus Global WFG

Assume T_1 and T_2 run at site 1, T_3 and T_4 run at site 2. Also assume T_3 waits for a lock held by T_4 which waits for a lock held by T_1 which waits for a lock held by T_2 which, in turn, waits for a lock held by T_3.

Local WFG

Global WFG

Deadlock Management

- Ignore

  Let the application programmer deal with it, or restart the system

- Prevention

  Guaranteeing that deadlocks can never occur in the first place. Check transaction when it is initiated. Requires no run time support.

- Avoidance

  Detecting potential deadlocks in advance and taking action to insure that deadlock will not occur. Requires run time support.

- Detection and Recovery

  Allowing deadlocks to form and then finding and breaking them. As in the avoidance scheme, this requires run time support.

Deadlock Prevention

- All resources which may be needed by a transaction must be predeclared.
  - The system must guarantee that none of the resources will be needed by an ongoing transaction.
  - Resources must only be reserved, but not necessarily allocated a priori
  - Unsuitability of the scheme in database environment
  - Suitable for systems that have no provisions for undoing processes.
- Evaluation:
  - – Reduced concurrency due to preallocation
  - – Evaluating whether an allocation is safe leads to added overhead.
  - – Difficult to determine (partial order)
  - + No transaction rollback or restart is involved.

Deadlock Avoidance

- Transactions are not required to request resources a priori.
- Transactions are allowed to proceed unless a requested resource is unavailable.
- In case of conflict, transactions may be allowed to wait for a fixed time interval.
- Order either the data items or the sites and always request locks in that order.
- More attractive than prevention in a database environment.

Deadlock Avoidance – Wait-Die Algorithm

If T_i requests a lock on a data item which is already locked by T_j, then T_i is permitted to wait iff $ts(T_i)<ts(T_j)$. If $ts(T_i)>ts(T_j)$, then T_i is aborted and restarted with the same timestamp.

- **if** $ts(T_i)<ts(T_j)$ **then** T_i waits **else** T_i dies
- non-preemptive: T_i never preempts T_j
- prefers younger transactions

Deadlock Avoidance – Wound-Wait Algorithm

If T_i requests a lock on a data item which is already locked by T_j , then T_i is permitted to wait iff $ts(T_i)>ts(T_j)$. If $ts(T_i)<ts(T_j)$, then T_j is aborted and the lock is granted to T_i.

- **if** $ts(T_i)<ts(T_j)$ **then** T_j is wounded **else** T_i waits
- preemptive: T_i preempts T_j if it is younger
- prefers older transactions

Deadlock Detection

- Transactions are allowed to wait freely.
- Wait-for graphs and cycles.
- Topologies for deadlock detection algorithms
  - Centralized
  - Distributed
  - Hierarchical

Centralized Deadlock Detection

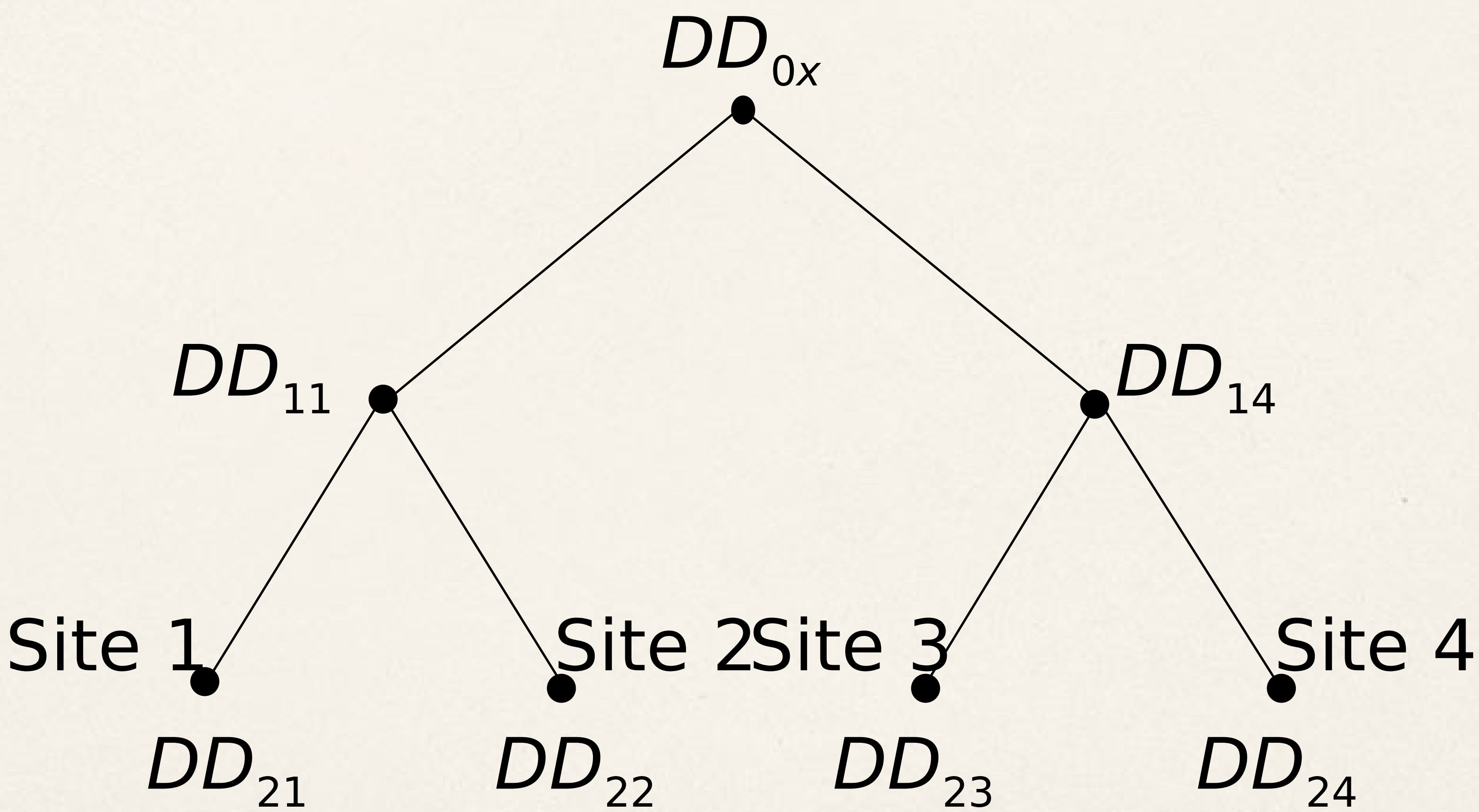
- One site is designated as the deadlock detector for the system. Each scheduler periodically sends its local WFG to the central site which merges them to a global WFG to determine cycles.
- How often to transmit?

  Too often ⇒ higher communication cost but lower delays due to undetected deadlocks

  Too late ⇒ higher delays due to deadlocks, but lower communication cost
- Would be a reasonable choice if the concurrency control algorithm is also centralized.
- Proposed for Distributed INGRES

Hierarchical Deadlock Detection

Build a hierarchy of detectors



Distributed Deadlock Detection

- Sites cooperate in detection of deadlocks.
- One example:

  The local WFGs are formed at each site and passed on to other sites. Each local WFG is modified as follows:

  1. Since each site receives the potential deadlock cycles from other sites, these edges are added to the local WFGs
  2. The edges in the local WFG which show that local transactions are waiting for transactions at other sites are joined with edges in the local WFGs which show that remote transactions are waiting for local ones.

  Each local deadlock detector:

  - looks for a cycle that does not involve the external edge. If it exists, there is a local deadlock which can be handled locally.
  - looks for a cycle involving the external edge. If it exists, it indicates a potential global deadlock. Pass on the information to the next site.

"Relaxed" Concurrency Control

- Non-serializable histories
  - E.g., ordered shared locks
  - Semantics of transactions can be used
    - Look at semantic compatibility of operations rather than simply looking at reads and writes
- Nested distributed transactions
  - Closed nested transactions
  - Open nested transactions
  - Multilevel transactions

Multilevel Transactions

Consider two transactions

T_1: Withdraw(o,x) T_2: Withdraw(o,y)

Deposit(p,x) Deposit(p,y)