DISTRIBUTED CONCURRENCY CONTROL

UNIT-2

SUB-UNIT-2.3

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.

This is called as concurrency control. It creates 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 Schedule

- An order in which the operations of a set of transactions are executed is called schedule.
- A schedule (history) can be defined as a partial order over the operations of a set of transactions.
- Consider three different transactions T1,T2,T3 as

T1: Read(x)	**T2:** Write(x)	**T3:** Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

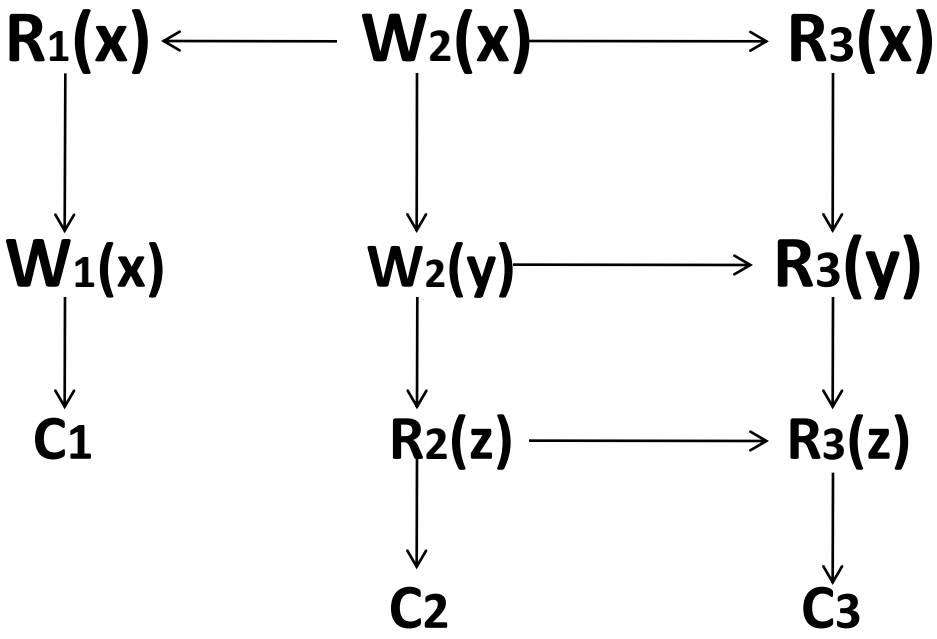
H_1={W2(x),R1(x), R3(x),W1(x),C1,W2(y),R3(y),R2(z),C2,R3(z),C3}

Complete Schedule-Example

Given three transactions

T1: Read(x)	**T2:** Write(x)	**T3:** Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

A possible complete schedule is given as the DAG



Concurrency Control Algorithm

A. Lock based Concurrency Control Algorithm
B. Timestamp Based Concurrency Control Algorithm

A. Lock based Concurrency Control

- 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.

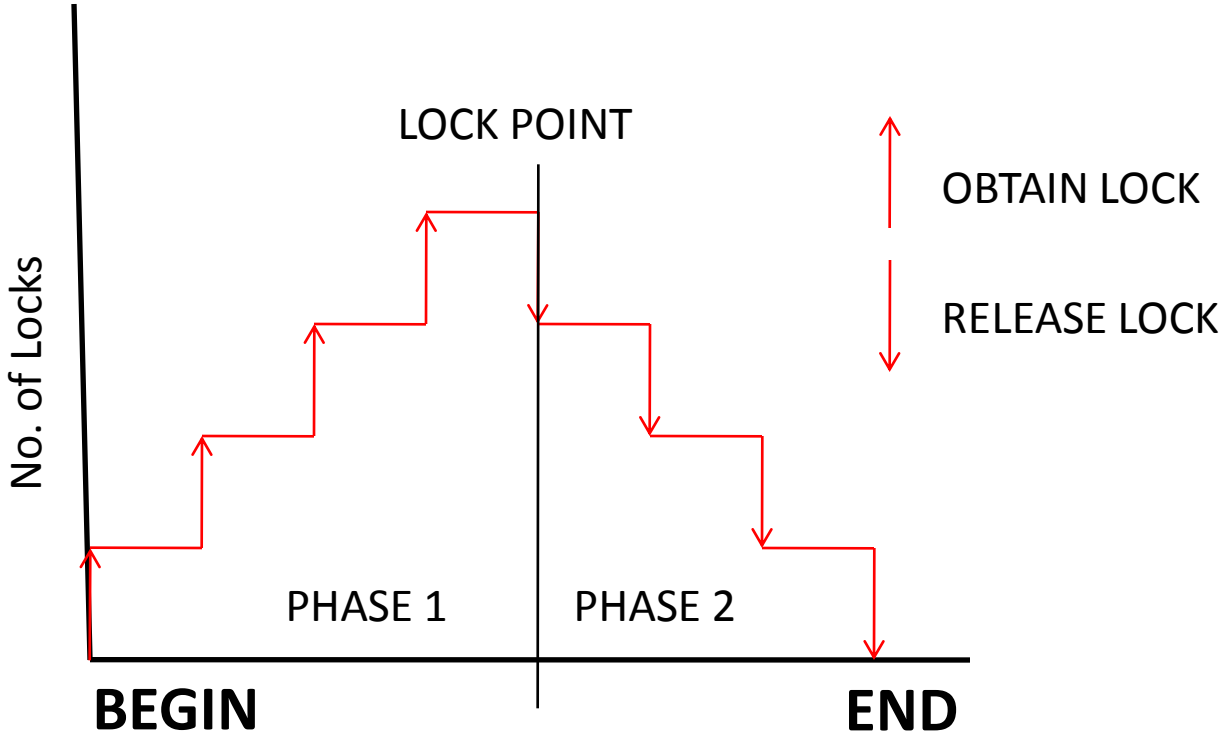
A. Lock based Concurrency Control

Types of lock based concurrency control methods

1. **Two Phase Locking algorithm(2PL)**
2. **Strict Two Phase Locking algorithm(S2PL)**
3. **Centralized Two Phase locking(C2PL)**
4. **Distributed Two Phase Locking(D2PL)**

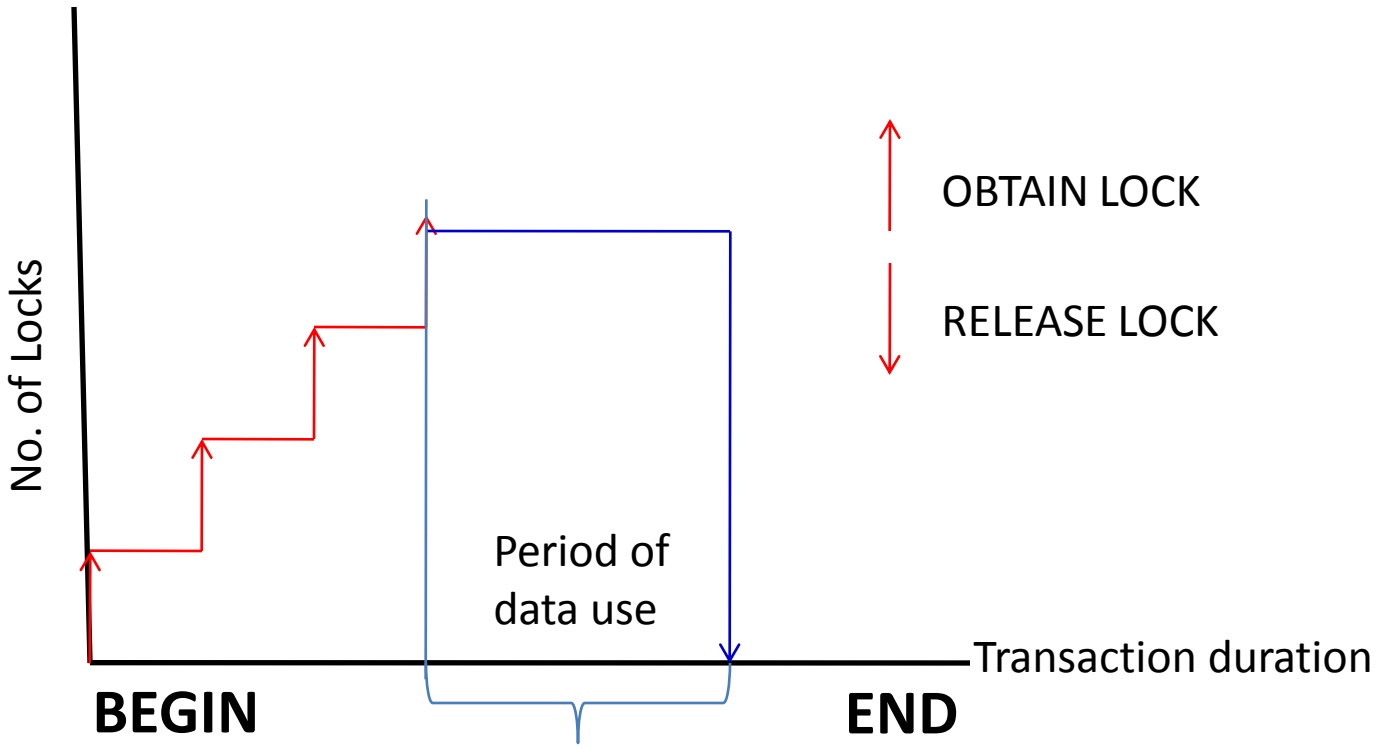
1. Two phase locking (2PL)

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.



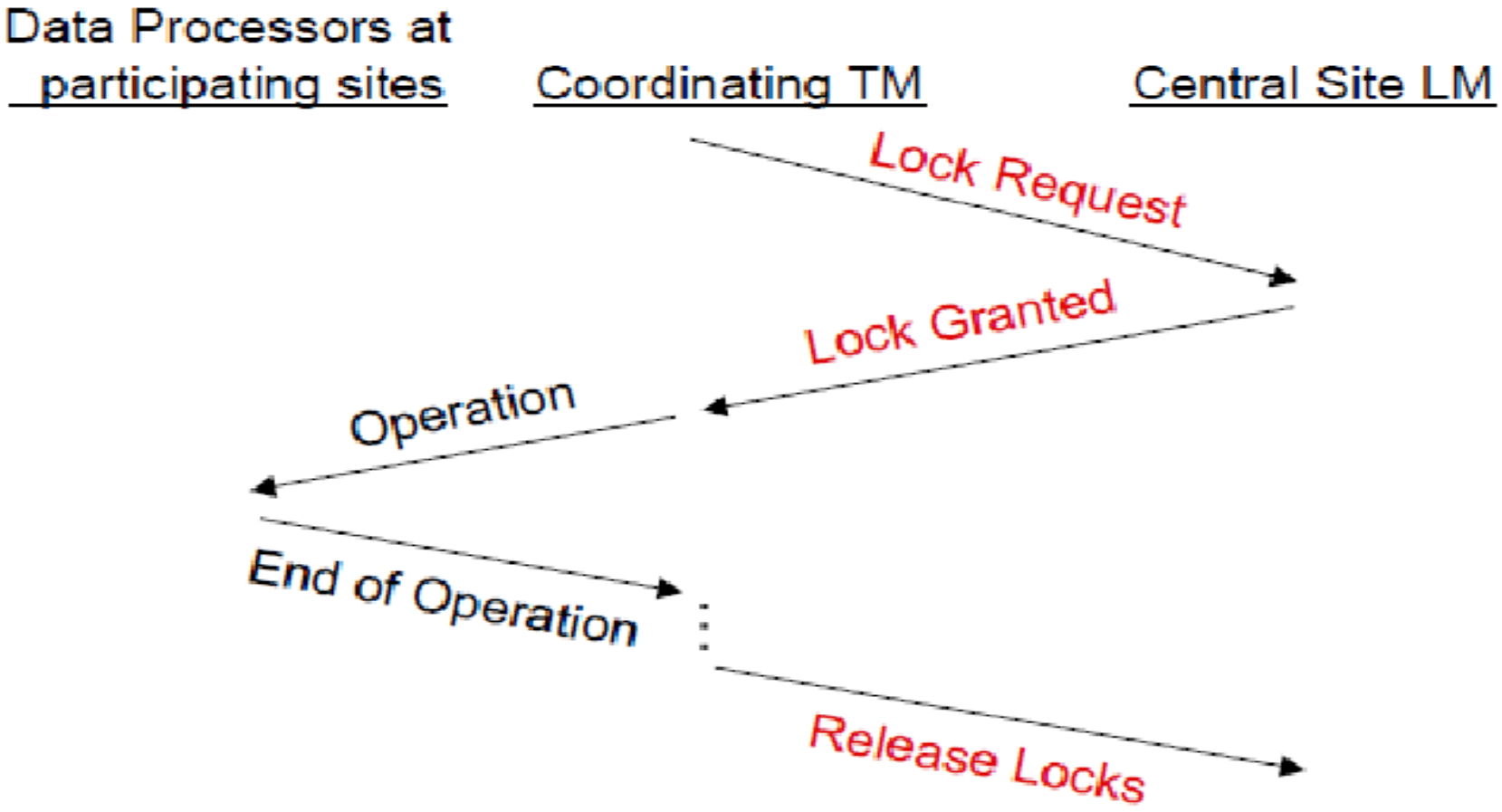
2. Strict Two phase locking (S2PL)

Holds locks till the end of the transaction



3. Centralised Two phase locking (C2PL)

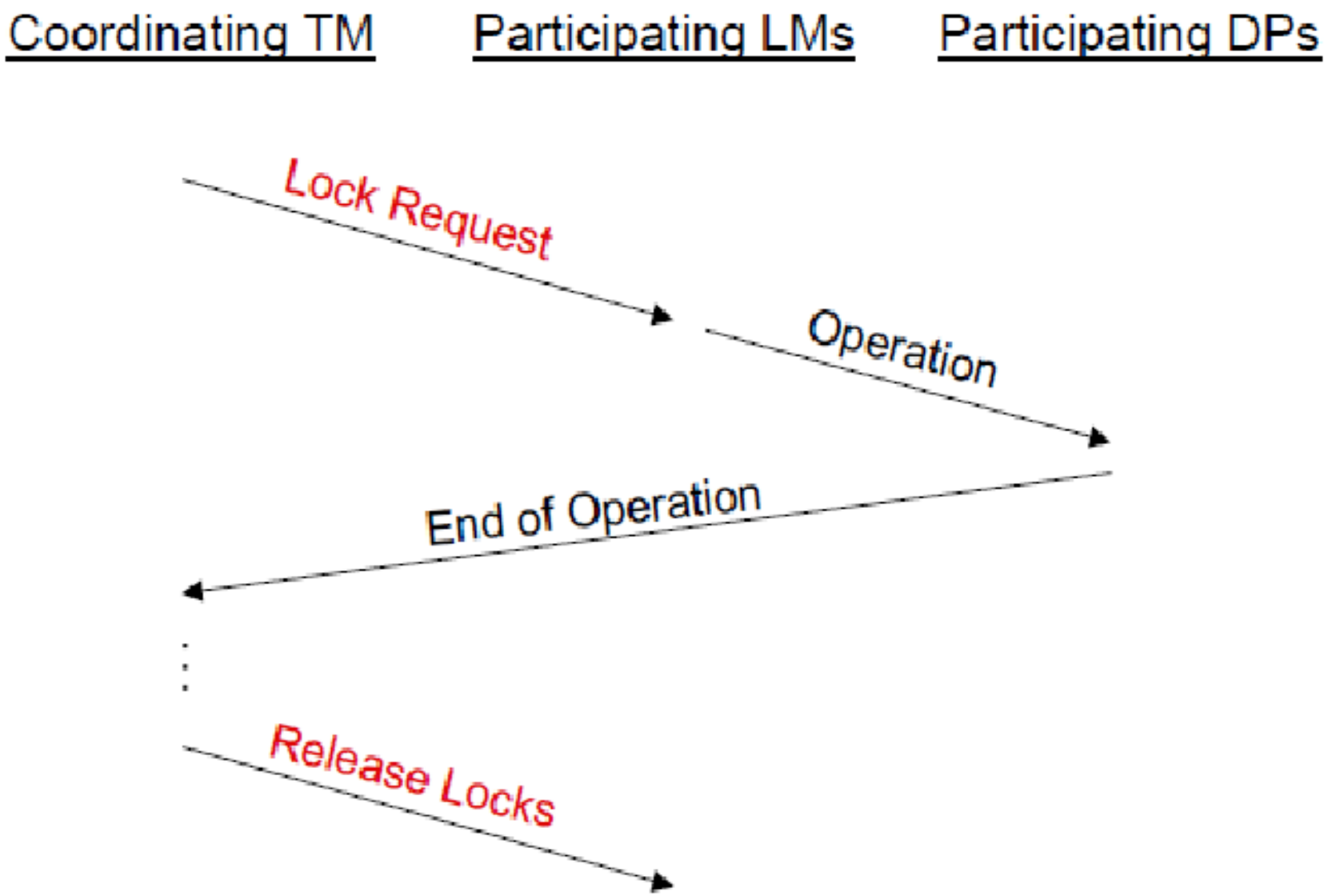
- There is only one 2PL scheduler in the distributed system.
- Lock requests are issued to the central scheduler.



4. Distributed Two phase locking (D2PL)

- 2PL schedulers are placed at each site. Each scheduler handles lock requests for data at that site.
- 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.*

4. Distributed Two phase locking (D2PL)



B. Timestamp Ordering

- Transaction (*Ti) is assigned a* ***globally unique*** **timestamp** *ts(Ti).*
- Transaction manager attaches the timestamp to all operations issued by the transaction.
- 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 write on x*
- Conflicting operations are resolved by timestamp order.

Basic T/O:

for *Ri(x)*

if *ts(Ti) < wts(x)*

then reject *Ri(x)*

else accept *Ri(x)*

rts(x) ←Tts(Ti)

for Wi(x)

if ts(Ti) < rts(x) and ts(Ti) < wts(x)

then reject Wi(x)

else accept Wi(x)

wts(x) ←Tts(Ti)

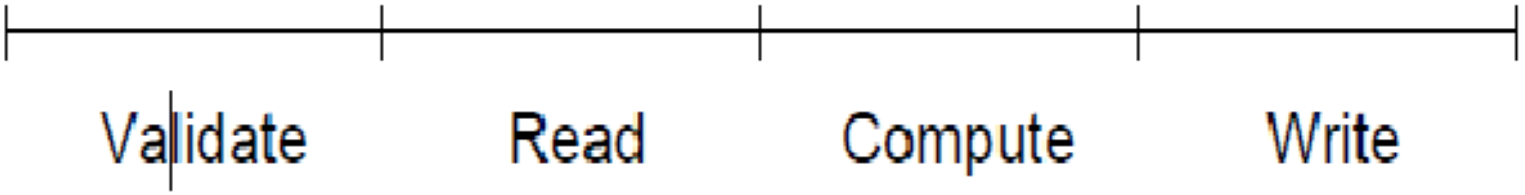
B. Timestamp Ordering

Type

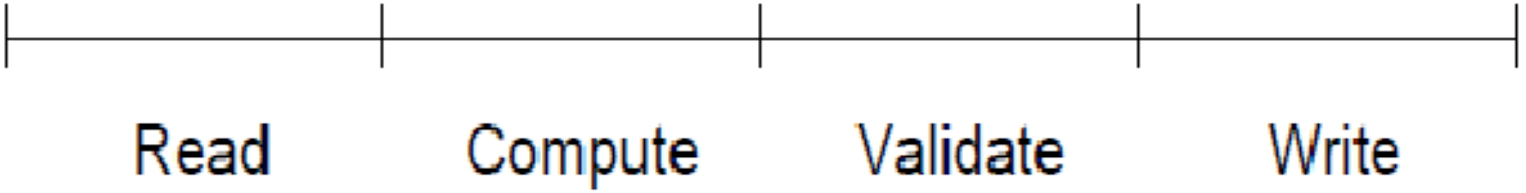
1. **Conservative timestamp ordering**
2. **Multiversion timestamp ordering**

Optimistic method for Concurrency Control

Pessimistic execution



Optimistic execution



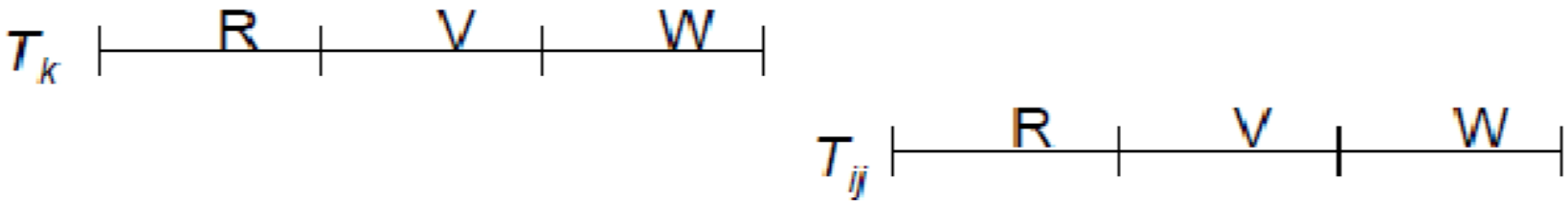
Optimistic method for Concurrency Control

- Transaction execution model: divide into Sub-transactions each of which execute at a site.
  - *Tij: transaction Ti that executes at site j*
- Transactions run independently at each site until they reach the end of their read phases.
- All sub-transactions are assigned a timestamp at the end of their read phase.
- Validation test is performed during validation phase. If one fails, all rejected.

Optimistic CC validation Test

1) If all transactions *Tk where ts(Tk) < ts(Tij)* have completed their write phase before *Tij* has started its read phase, then validation Succeeds.

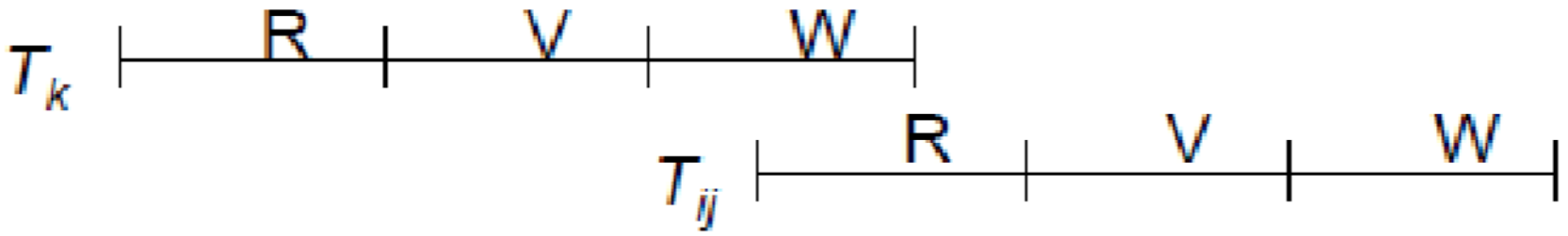
- Transaction are executed in serial order



Optimistic CC validation Test

2) If there is any transaction *Tk such that ts(Tk)<ts(Tij)* and which completes its write phase while *Tij is in* its read phase, then validation succeeds.

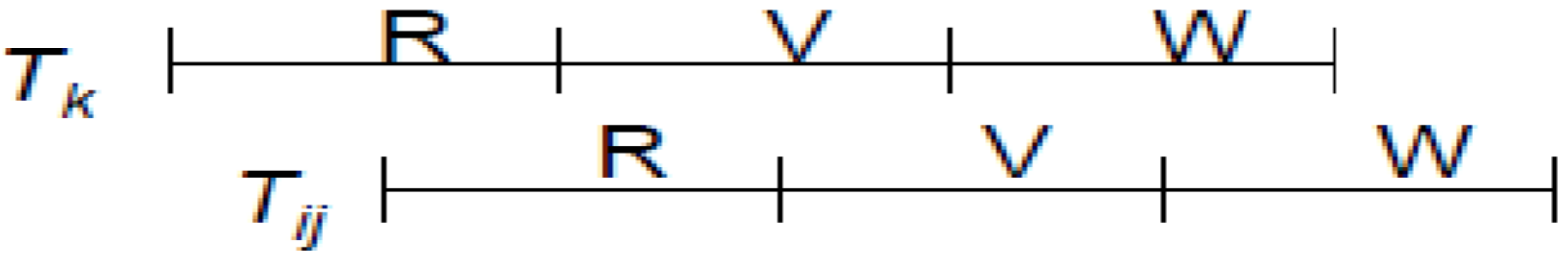
- Read and write phases overlap, but *Tij does not read data* items written by *Tk.*



Optimistic CC validation Test

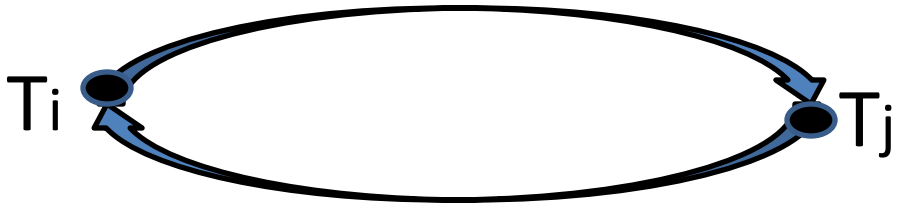
3) If there is any transaction *Tk such that ts(Tk)< ts(Tij)* and which completes its read phase before *Tij* completes its read phase, then validation succeeds.

- 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 *Ti waits for another transaction Tj to release* a lock on an entity, then *Ti → Tj in 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 priority.
  - ➢Unsuitability of the scheme in database environment
  - ➢Suitable for systems that have no provisions for undoing processes.

- **Evaluation:**
  - ➢Reduced concurrency due to pre-allocation
  - ➢ Evaluating whether an allocation is safe leads to added overhead.
  - ➢No transaction rollback or restart is involved.

Deadlock Avoidance

- Transactions are not required to request resources as priority.
- 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 Detection

1. **Wait Die Rule for avoidance:**

- If *Ti requests a lock on a data item* which is already locked by *Tj, then Ti is permitted to* wait iff *ts(Ti)<ts(Tj). If ts(Ti)>ts(Tj), then Ti is aborted*

  and restarted with the same timestamp.
- **if *ts(Ti)<ts(Tj) then Ti waits else Ti dies***
- non-preemptive: *Ti never preempts Tj*
- prefers younger transactions

Deadlock Avoidance Detection

2. **Wound Wait Rule for avoidance:**

- If *Ti requests a lock on a data* item which is already locked by *Tj , then Ti is* permitted to wait iff *ts(Ti)>ts(Tj). If ts(Ti)<ts(Tj), then Tj is aborted and the lock is granted to Ti.*
- **if *ts(Ti)<ts(Tj) then Tj is wounded else Ti waits***
- preemptive: *Ti preempts Tj if it is younger*
- prefers older transactions

Deadlock Detection

- In this the transactions are allowed to wait freely.
- Deadlocks are detected by creating WFG and cycles.
- The topologies for deadlock detection implemented are as

  a)Centralized

  b)Distributed and

  c)Hierarchical

1.Centralized Deadlock Detection

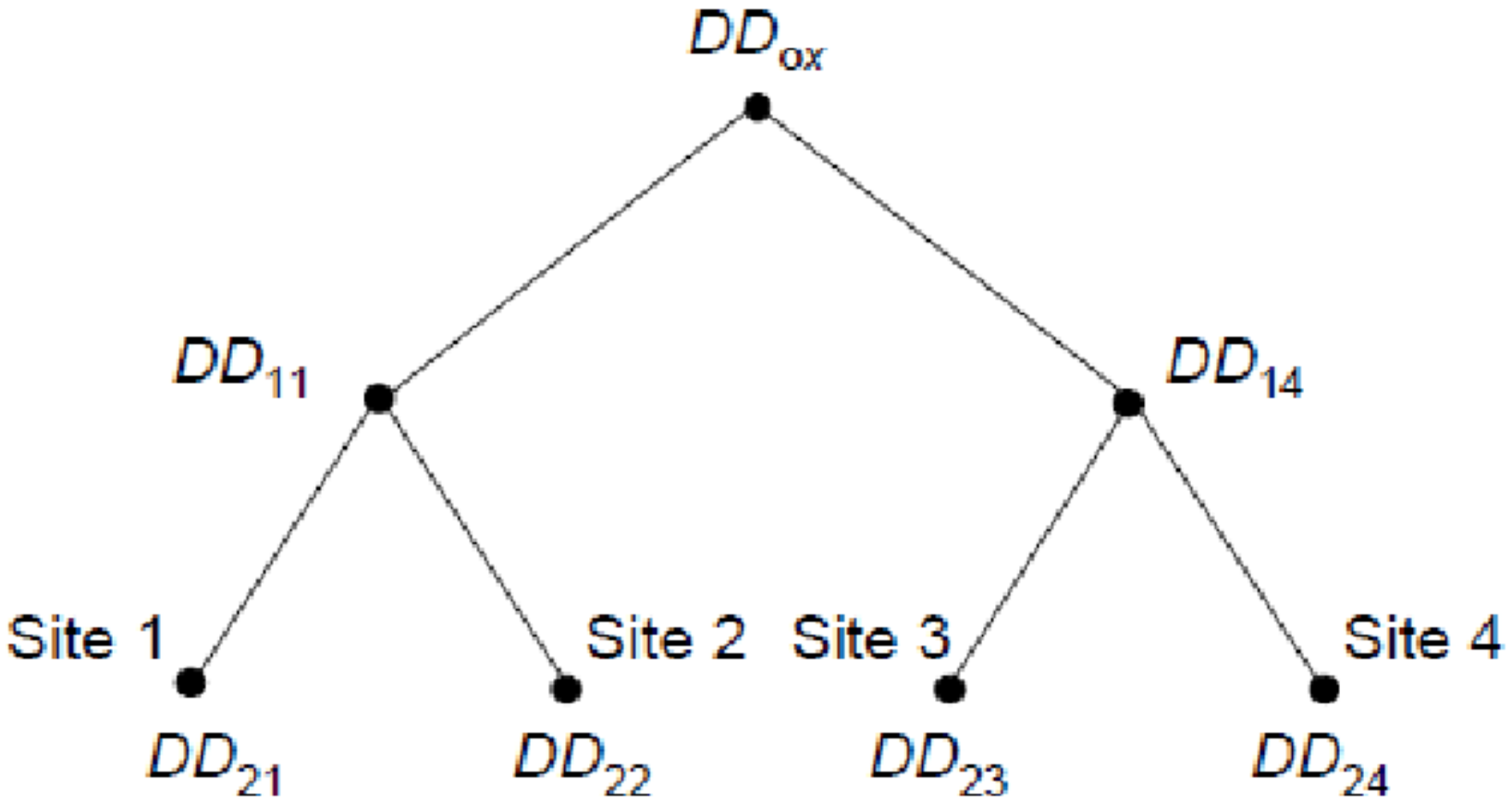
- One site is designated as the deadlock detector for the entire system. Each scheduler periodically sends its local WFG to the central site which merges them to a global WFG to determine cycles.
- Frequency of transmission

  a)Too often ->higher cost, lower delays

  b)Too late->less cost, higher delays
- Proposed for Distributed INGRES

2.Hierarchial Deadlock Detection

- Assign systems as deadlock detector according to the sites in hierarchy as shown



3. Distributed Deadlock Detection

- Sites cooperate in detection of deadlocks.
- The local WFGs are formed at each site and passed on to other sites.
- Since each site receives the potential deadlock cycles from other sites, these edges are added to the local WFGs.
- 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 exits it means global deadlock so pass information to next site.

Schedule Definition

- A sequence of instructions that specify the chronological order in which instructions of concurrent transactions are executed.
- Transactions execute concurrently, but the net effect of the resulting history upon the database is equivalent to some serial history.

Serializable Schedule

- The consistency of database is ensured under concurrent execution by making sure that any schedule that is executed has the same effect as a schedule that could have occurred without any concurrent execution. Such schedules are called **serializable schedules**. A system that ensures this property of transaction is said to ensure serializability.

Non-Serializability

- Consider following two transactions

T1: Read(x)	*T2: Read(x)*
x ←x+5	*x ←x∗15*
Write(*x*)	*Write(x)*
Commit	Commit

- The following two local histories(schedule) are individually serializable (in fact serial), but the two transactions are not globally serializable.
  - *LH1={R1(x),W1(x),C1,R2(x),W2(x),C2}*
  - *LH2={R2(x),W2(x),C2,R1(x),W1(x),C1}*

Nested Transactions

- Transaction inside transaction.
- Have the same properties as their parents
- May themselves have other nested transactions.
- Introduces concurrency control and recovery concepts to within the transaction.
- Type

a) Close nesting: Sub-transactions begin *after their parents and finish before* them.

b) Open nesting:Subtransactions can execute and commit independently.

Nested Transactions(example)

- **Begin_transaction Reservation**

  …

  Begin_transaction Airline

  …

  end. {Airline}

  Begin_transaction Hotel

  …

  end. {Hotel}

- **end. {Reservation)**