Outline

- Introduction
- Background
- Distributed DBMS Architecture
- Distributed Database Design
- Distributed Query Processing
- Distributed Transaction Management
  - Concurrency Control Algorithms
- Building Distributed Database Systems (RAID)
- Mobile Database Systems
- Privacy, Trust, and Authentication
- Peer to Peer Systems

Useful References

- Textbook *Principles of Distributed Database Systems*,

  Chapter 11.3-11.5

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

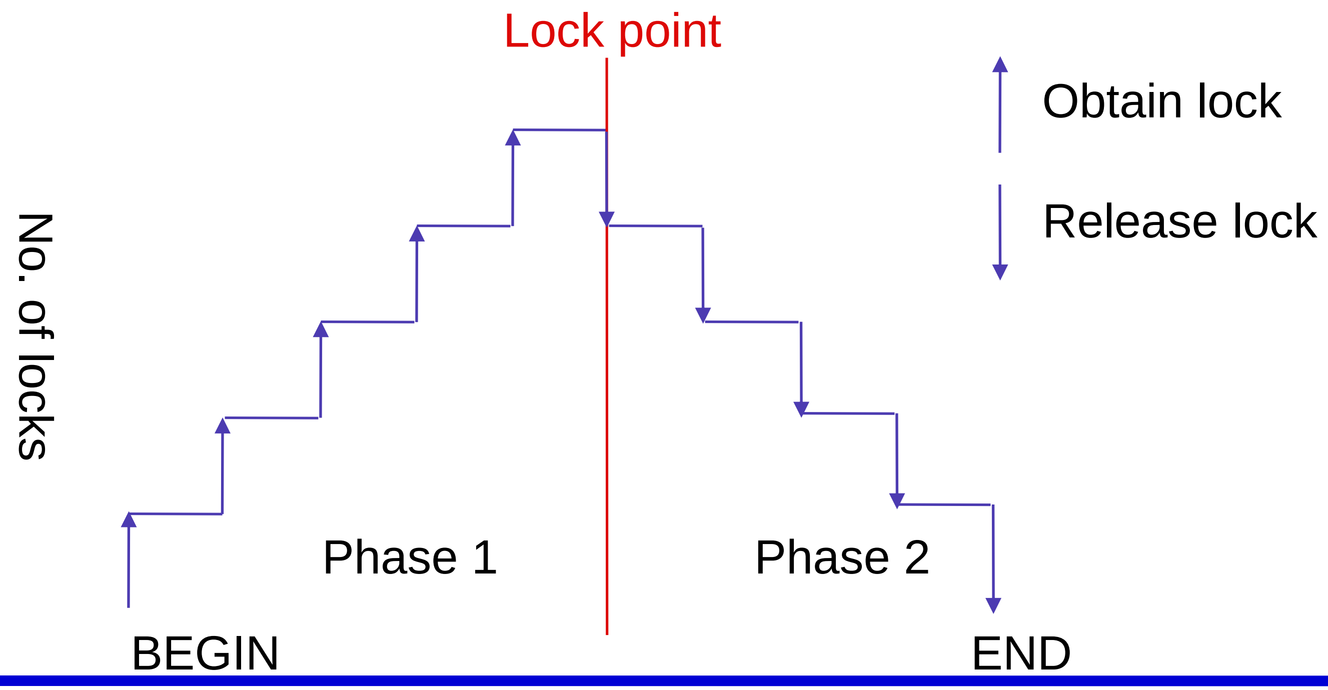
- Transactions indicate their intentions by requesting locks from the scheduler (called lock manager).
- Locks are either read lock (*rl*) [also called shared lock] or write lock (*wl*) [also called exclusive lock]
- Read locks and write locks conflict (because Read and Write operations are incompatible

	rl	*wl*
rl	yes	no
wl	no	no

- Locking works nicely to allow concurrent processing of transactions.

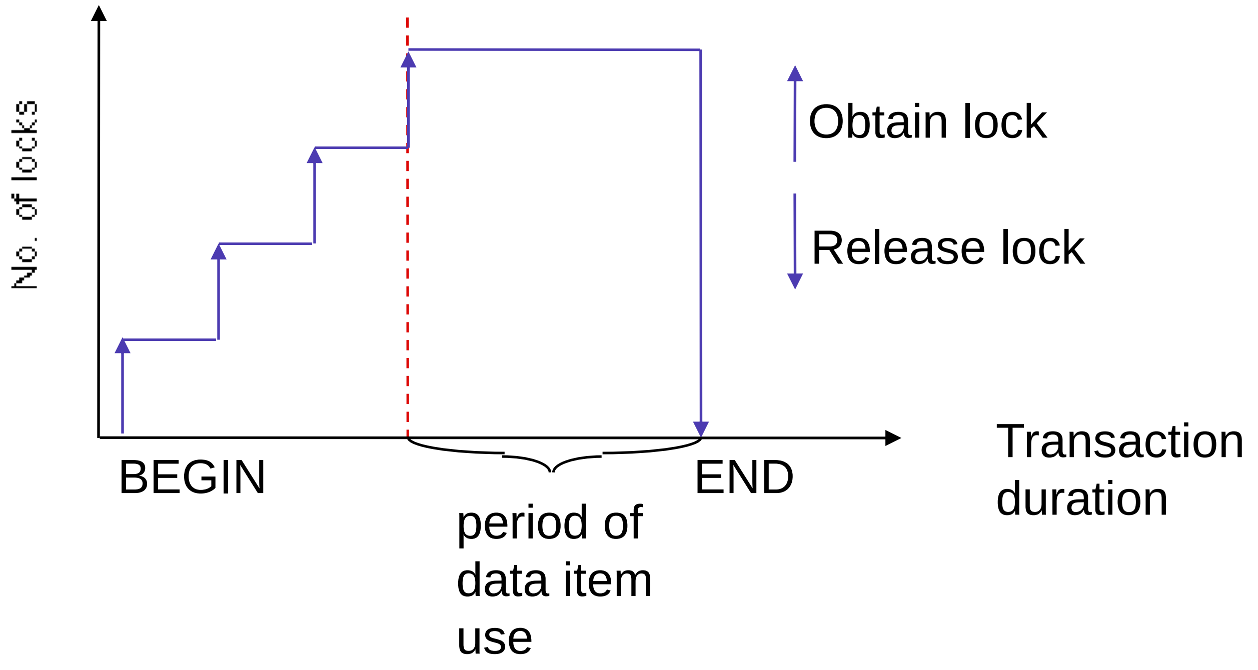
Two-Phase Locking (2PL)

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



Strict 2PL

Hold locks until the end.



Testing for Serializability

Consider transactions T_1, T_2, …, T_k

Create a directed graph (called a conflict graph), whose nodes are transactions. Consider a history of transactions.

If T_1 unlocks an item and T_2 locks it afterwards, draw an edge from T_1 to T_2 implying T_1 must precede T_2 in any serial history

$$T_1 \rightarrow T_2$$

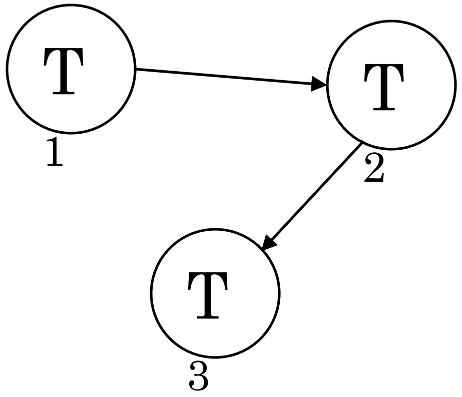
Repeat this for all unlock and lock actions for different transactions.

If graph has a cycle, the history is not serializable.

If graph is a cyclic, a topological sorting will give the serial history.

Example

T_1:	Lock X	
T_1:	Unlock X	$T_1 \rightarrow T_2$
T_2:	Lock X	
T_2:	Lock Y	
T_2:	Unlock X	
T_2:	Unlock Y	$T_2 \rightarrow T_3$
T_3:	Lock Y	
T_3:	Unlock Y	



Theorem

Two phase locking is a sufficient condition to ensure serializablility.

Proof: By contradiction.

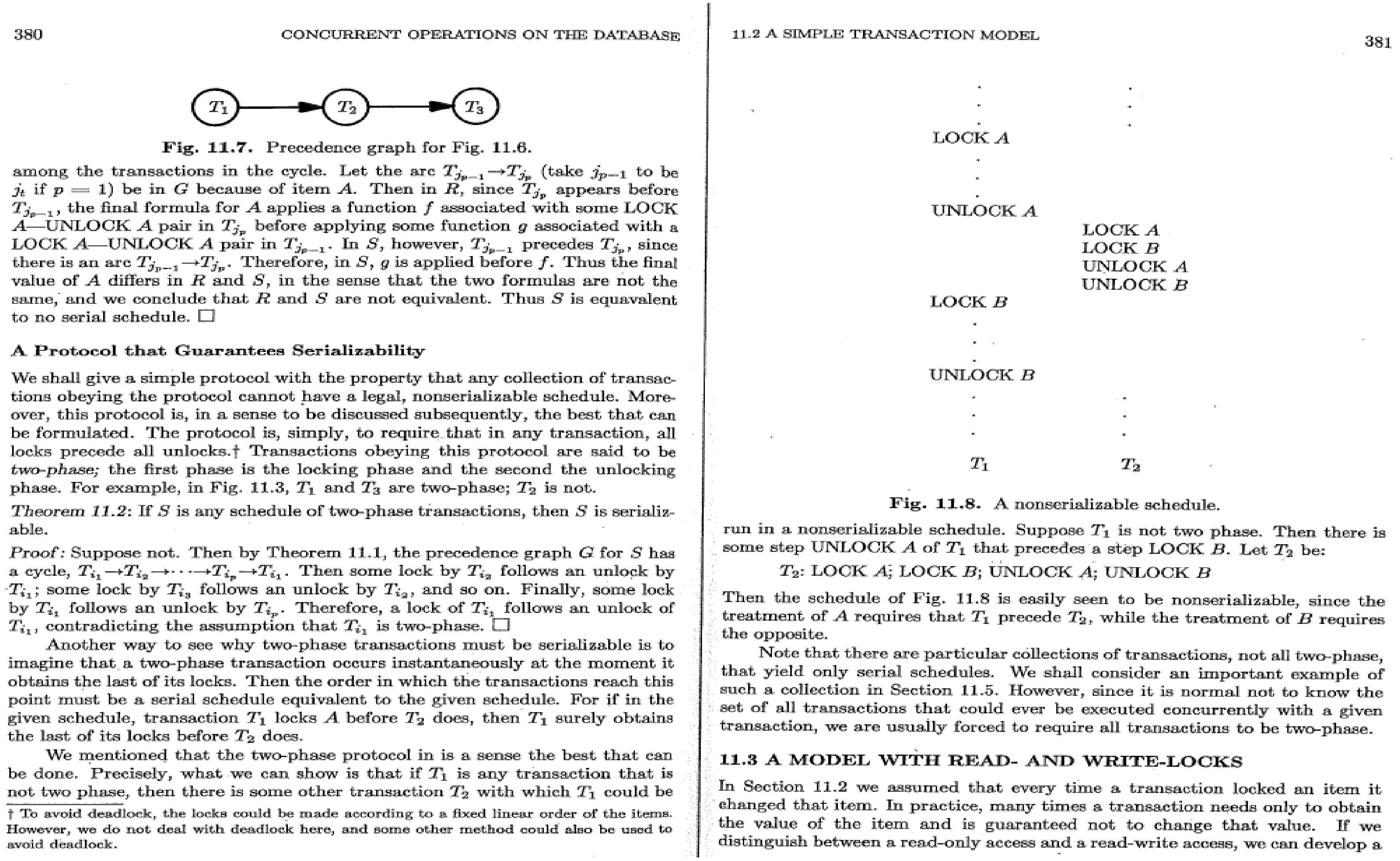
If history is not serializable, a cycle must exist in the conflict graph. This means the existence of a path such as

$$T_1 \rightarrow T_2 \rightarrow T_3 \ldots T_k \rightarrow T_1.$$

This implies T_1 unlocked before T_2 and after T_k.

T_1 requested a lock again. This violates the condition of two phase locking.

2PL from Jeff Ullman's book



$$\begin{tikzcd} T_1 \arrow[r] & T_2 \arrow[r] & T_3 \end{tikzcd}$$

Fig. 11.7. Precedence graph for Fig. 11.6.

among the transactions in the cycle. Let the arc $T_{j_{p-1}} \rightarrow T_{j_p}$ (take j_{p-1} to be j_t if $p = 1$) be in G because of item A. Then in R, since T_{j_p} appears before $T_{j_{p-1}}$, the final formula for A applies a function f associated with some LOCK A—UNLOCK A pair in T_{j_p} before applying some function g associated with a LOCK A—UNLOCK A pair in $T_{j_{p-1}}$. In S, however, $T_{j_{p-1}}$ precedes T_{j_p}, since there is an arc $T_{j_{p-1}} \rightarrow T_{j_p}$. Therefore, in S, g is applied before f. Thus the final value of A differs in R and S, in the sense that the two formulas are not the same, and we conclude that R and S are not equivalent. Thus S is equavalent to no serial schedule. □

A Protocol that Guarantees Serializability

We shall give a simple protocol with the property that any collection of transactions obeying the protocol cannot have a legal, nonserializable schedule. Moreover, this protocol is, in a sense to be discussed subsequently, the best that can be formulated. The protocol is, simply, to require that in any transaction, all locks precede all unlocks.† Transactions obeying this protocol are said to be *two-phase;* the first phase is the locking phase and the second the unlocking phase. For example, in Fig. 11.3, T_1 and T_3 are two-phase; T_2 is not.

Theorem 11.2: If S is any schedule of two-phase transactions, then S is serializable.

Proof: Suppose not. Then by Theorem 11.1, the precedence graph G for S has a cycle, $T_{i_1} \rightarrow T_{i_2} \rightarrow \cdots \rightarrow T_{i_p} \rightarrow T_{i_1}$. Then some lock by T_{i_2} follows an unlock by T_{i_1}; some lock by T_{i_3} follows an unlock by T_{i_2}, and so on. Finally, some lock by T_{i_1} follows an unlock by T_{i_p}. Therefore, a lock of T_{i_1} follows an unlock of T_{i_1}, contradicting the assumption that T_{i_1} is two-phase. □

Another way to see why two-phase transactions must be serializable is to imagine that a two-phase transaction occurs instantaneously at the moment it obtains the last of its locks. Then the order in which the transactions reach this point must be a serial schedule equivalent to the given schedule. For if in the given schedule, transaction T_1 locks A before T_2 does, then T_1 surely obtains the last of its locks before T_2 does.

We mentioned that the two-phase protocol in is a sense the best that can be done. Precisely, what we can show is that if T_1 is any transaction that is not two phase, then there is some other transaction T_2 with which T_1 could be

† To avoid deadlock, the locks could be made according to a fixed linear order of the items. However, we do not deal with deadlock here, and some other method could also be used to avoid deadlock.

```
      .                    .
      .                    .
      .                    .
  LOCK A
      .
      .
      .
  UNLOCK A
                        LOCK A
                        LOCK B
                        UNLOCK A
                        UNLOCK B
  LOCK B
      .
      .
      .
  UNLOCK B
      .                    .
      .                    .
      .                    .
     T1                   T2
```

Fig. 11.8. A nonserializable schedule.

run in a nonserializable schedule. Suppose T_1 is not two phase. Then there is some step UNLOCK A of T_1 that precedes a step LOCK B. Let T_2 be:

T_2: LOCK A; LOCK B; UNLOCK A; UNLOCK B

Then the schedule of Fig. 11.8 is easily seen to be nonserializable, since the treatment of A requires that T_1 precede T_2, while the treatment of B requires the opposite.

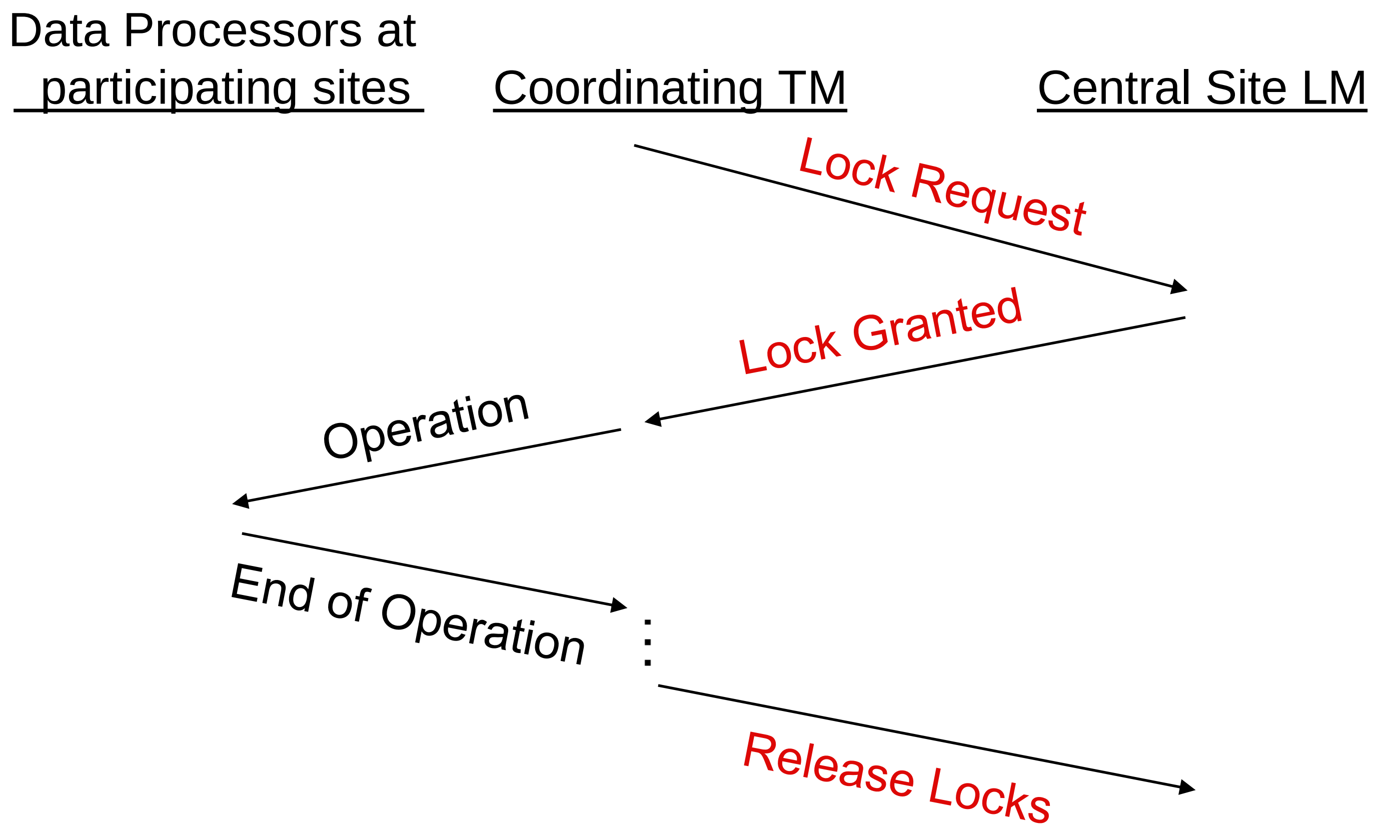
Note that there are particular collections of transactions, not all two-phase, that yield only serial schedules. We shall consider an important example of such a collection in Section 11.5. However, since it is normal not to know the set of all transactions that could ever be executed concurrently with a given transaction, we are usually forced to require all transactions to be two-phase.

11.3 A MODEL WITH READ- AND WRITE-LOCKS

In Section 11.2 we assumed that every time a transaction locked an item it changed that item. In practice, many times a transaction needs only to obtain the value of the item and is guaranteed not to change that value. If we distinguish between a read-only access and a read-write access, we can develop a

Centralized 2PL

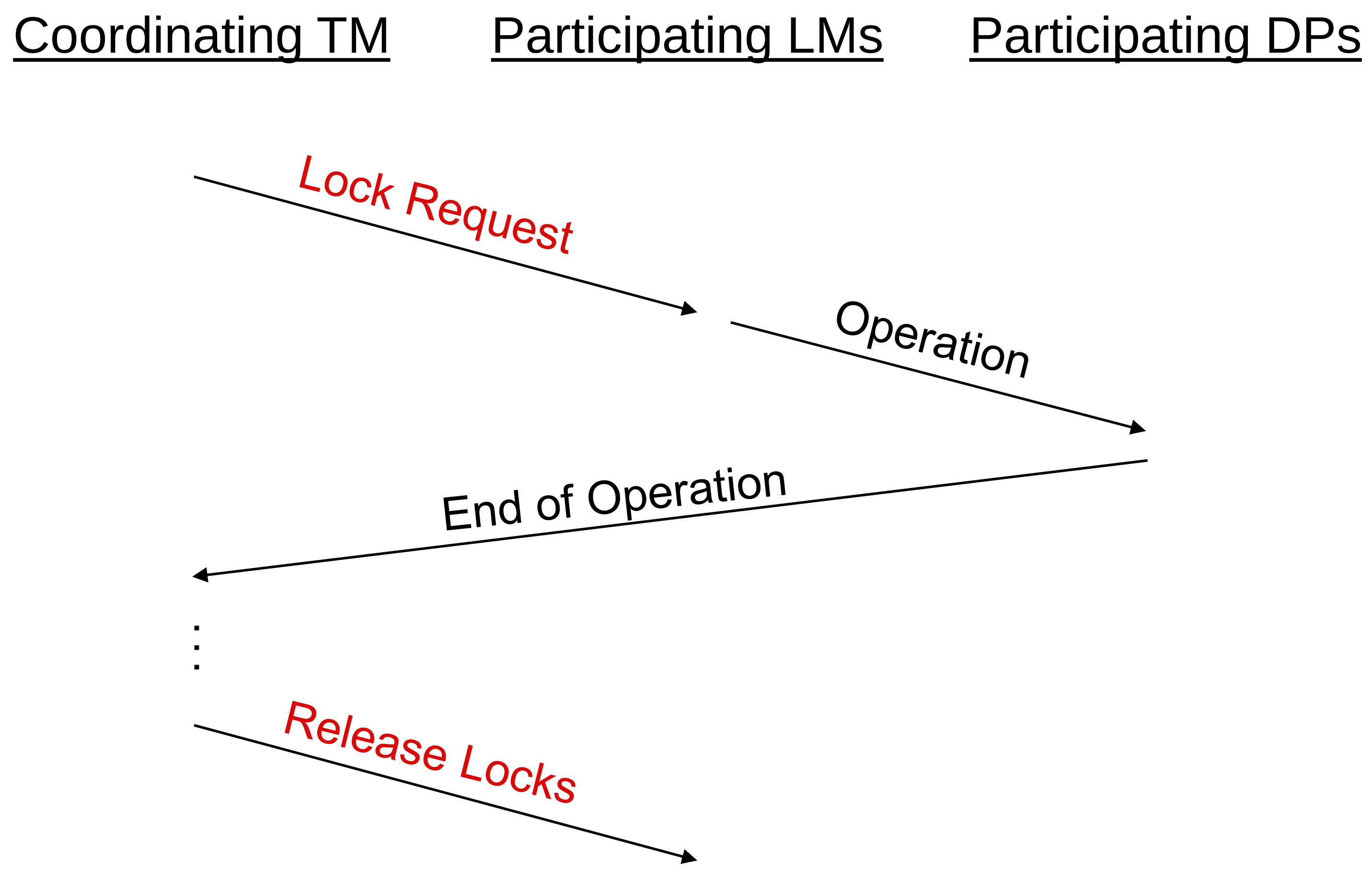
- There is only one 2PL scheduler in the distributed system.
- Lock requests are issued to the central scheduler.



Distributed 2PL

- 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.

Distributed 2PL Execution



Timestamp Ordering

- Transaction (T_i) is assigned a globally unique timestamp $ts(T_i)$.
- 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 read on x
- Conflicting operations are resolved by timestamp order.

Basic T/O:

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

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

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

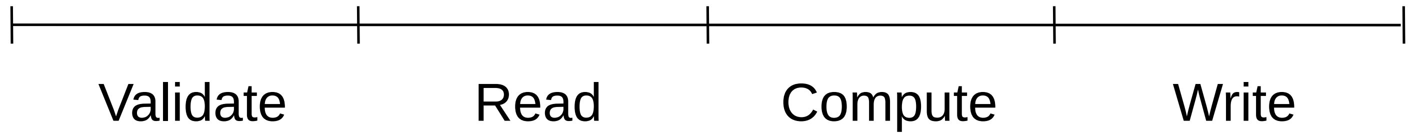
Multiversion Timestamp Ordering

- 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

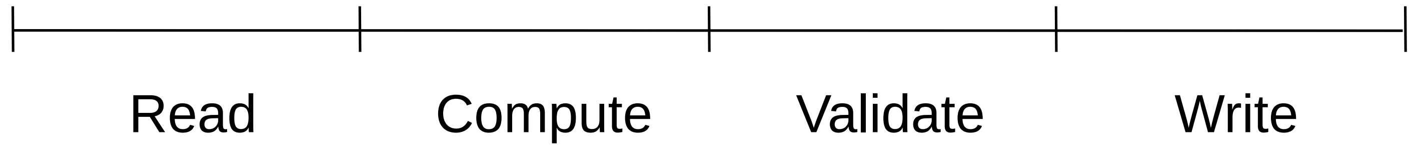
$$ts(T_i) < ts(x_r) < ts(T_j)$$

Optimistic Concurrency Control Algorithms

Pessimistic execution



Optimistic execution


Read
Compute
Validate
Write