# 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($x$) Write($x$) Commit

$T_2$: Write($x$) Write($y$) Read($z$) Commit

$T_3$: Read($x$) Read($y$) Read($z$) 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 $H_c(T) = \{\sum_T, \prec_H\}$ where

1. $\sum_T = \cup_i \sum_i$ , for $i = 1, 2, \ldots, n$
2. $\prec_H \supseteq \cup_i \prec_{T_i}$, for $i = 1, 2, \ldots, n$
3. For any two conflicting operations $O_{ij}, O_{kl} \in \sum_T$, 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

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

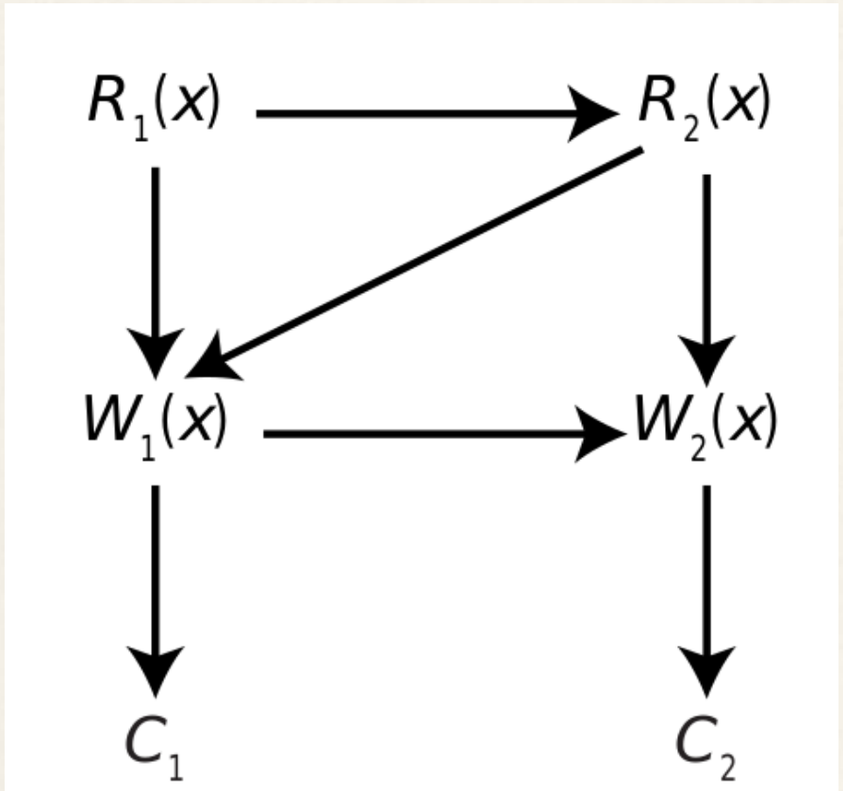
Thus

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

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



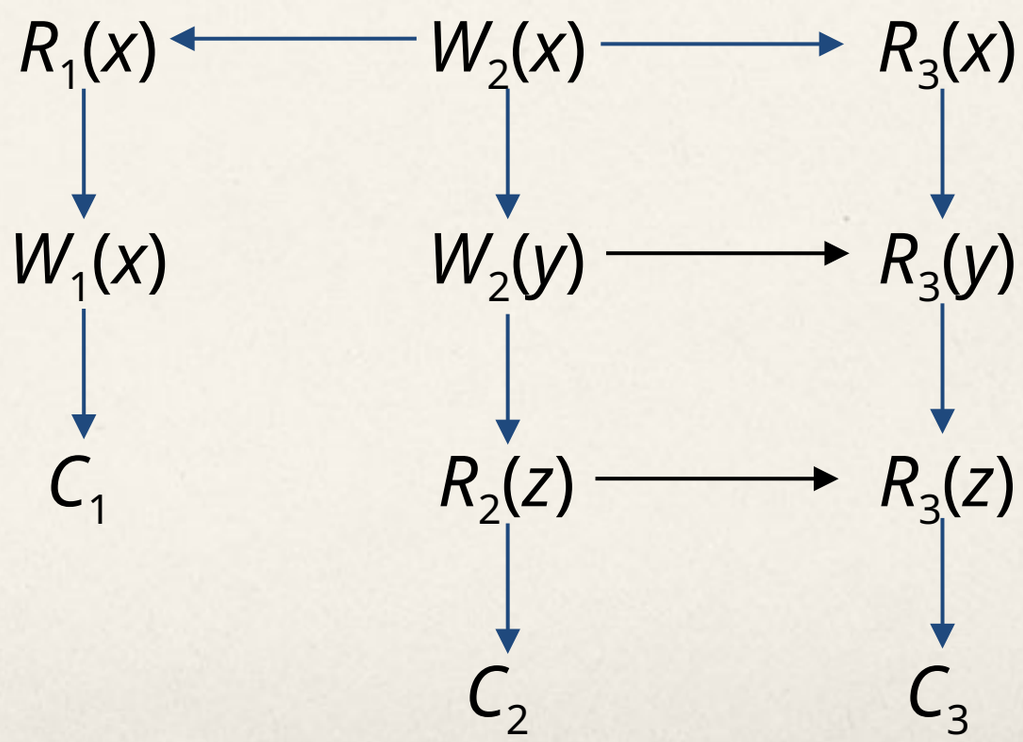

$$H_T^c = \{R_1(x), R_2(x), W_1(x), C_1, W_2(x), C_2\}$$

# Complete Schedule – Example2

Given three transactions

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

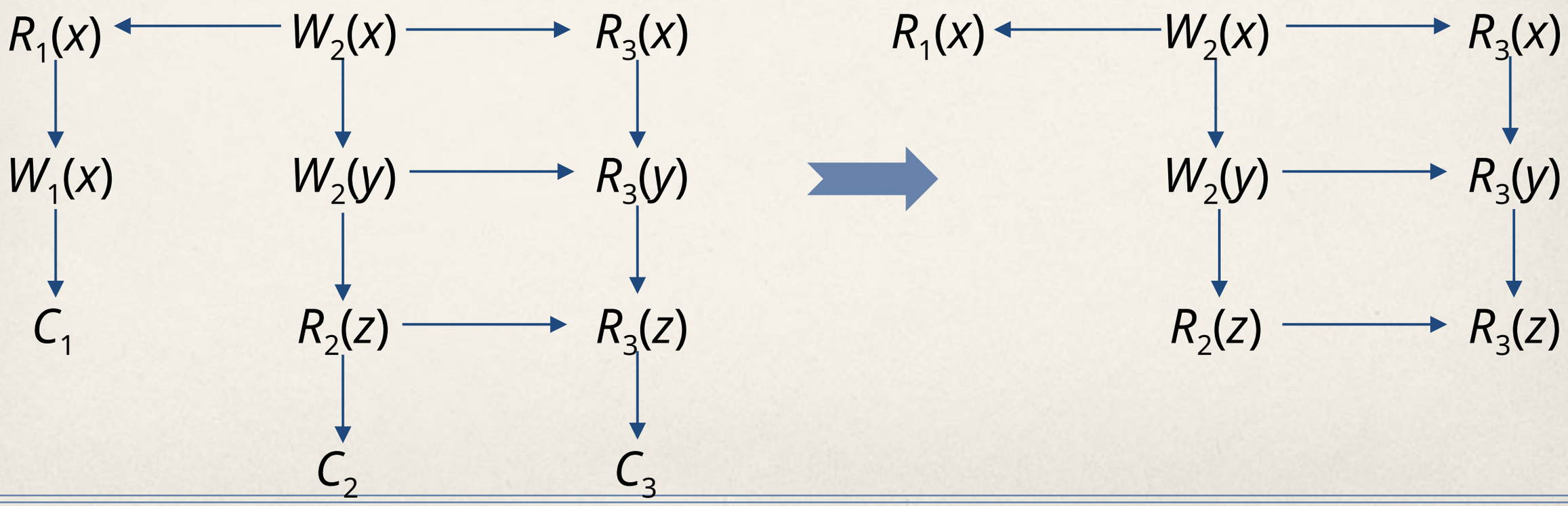
A possible complete schedule is given as the DAG



# 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.

| $T_1$: Read($x$) | $T_2$: Write($x$) | $T_3$: 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

| $T_1$: | $T_2$: | $T_3$: |
|---|---|---|
| Read($x$) | Write($x$) | 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

| $T_1$: | $T_2$: |
|---|---|
| Read($x$) | 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:

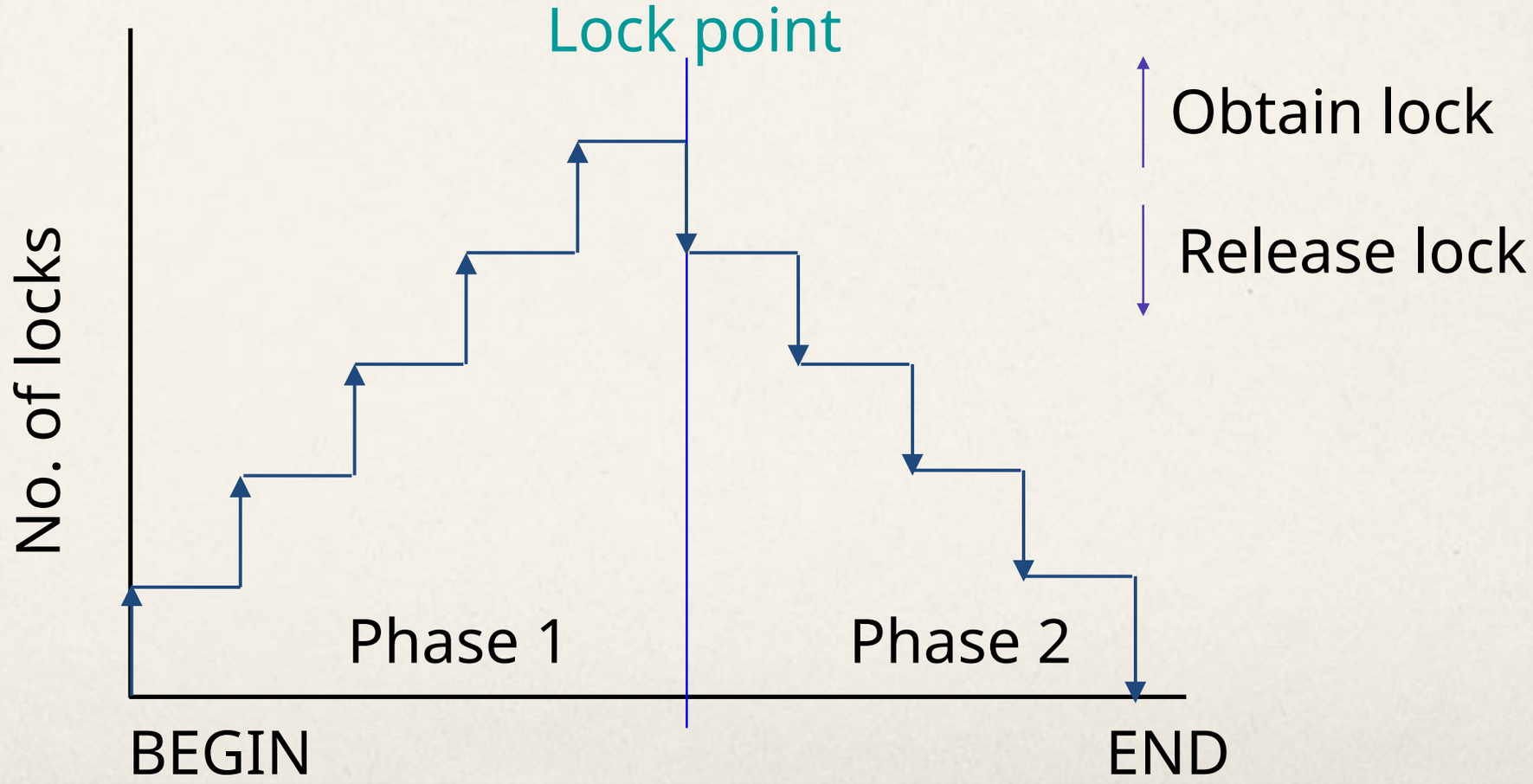
$$H = \{wl_1(x), R_1(x), W_1(x), lr_1(x), wl_2(x), R_2(x), w_2(x), lr_2(x), wl_2(y), R_2(y), W_2(y), lr_2(y), wl_1(y), R_1(y), W_1(y), lr_1(y)\}$$

# Naive Locking Algorithm

- The locking algorithm releases the locks that are held by a transaction (say, Ti ) 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.

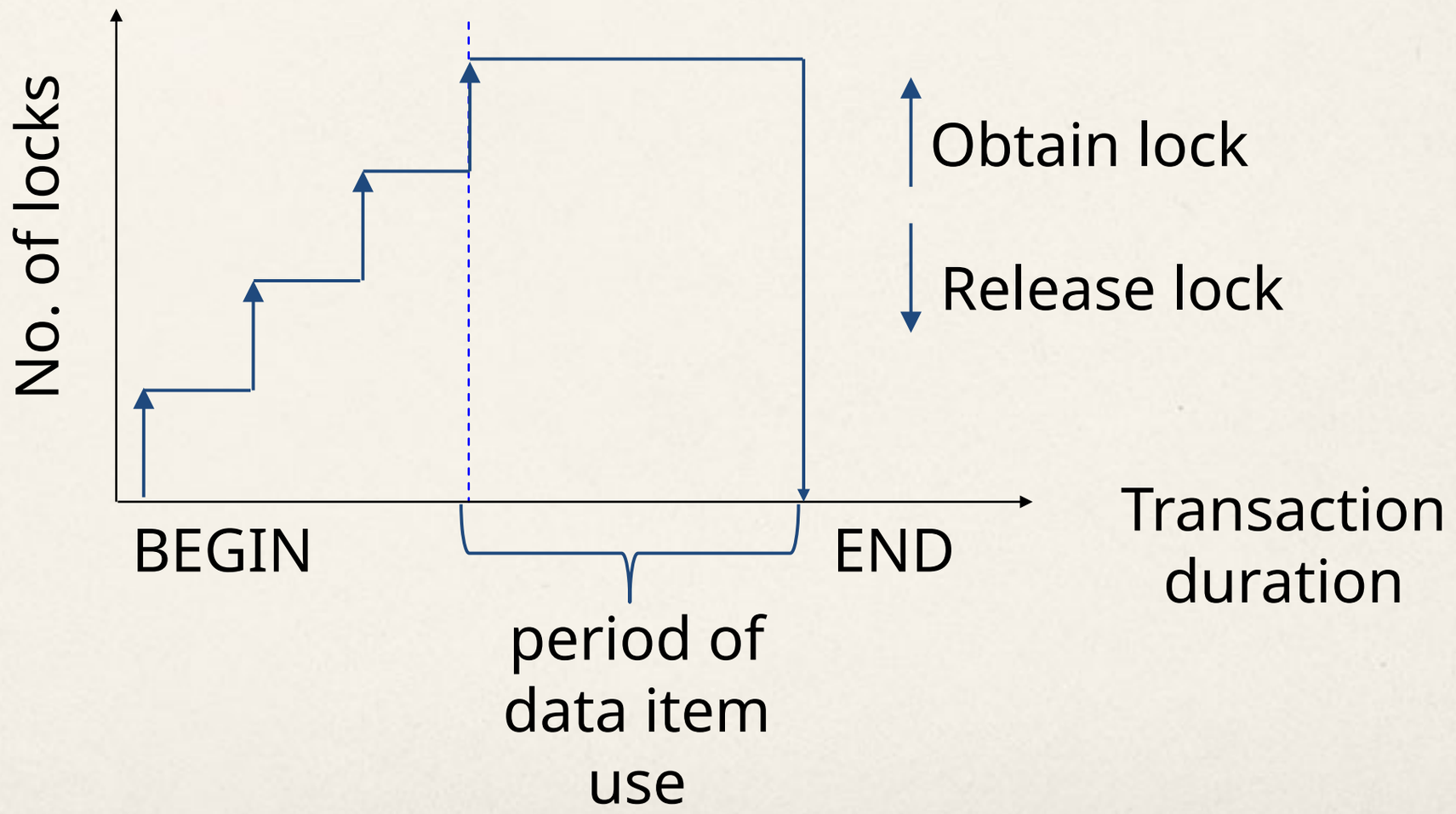


# Two-Phase Locking (2PL)

- 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.



# Distributed Transaction Processing

- Database consistency and transaction consistency
  - DB is in a consistent state if it obeys all of the consistency (integrity) constraints.
    - DB may be inconsistent during the execution of txn but consistent after txn terminates.
  - Txn consistency refers to the ops of concurrent txns.
    - DB remains in a consistent state when a number of txns running concurrently.
- On-Line Transaction Processing (OLTP) and On-Line Analytical Processing (OLAP)
  - OLTP apps (airline reservation or banking systems) are high-throughput transaction-oriented.
    - High-throughput transaction-oriented, need extensive data control and availability, and fast response times.
  - OLAP apps are, for example, trend analysis or forecasting, need to analyze historical, summarized data.
    - They use complex queries over potentially very large tables.

# Concurrency control algorithms

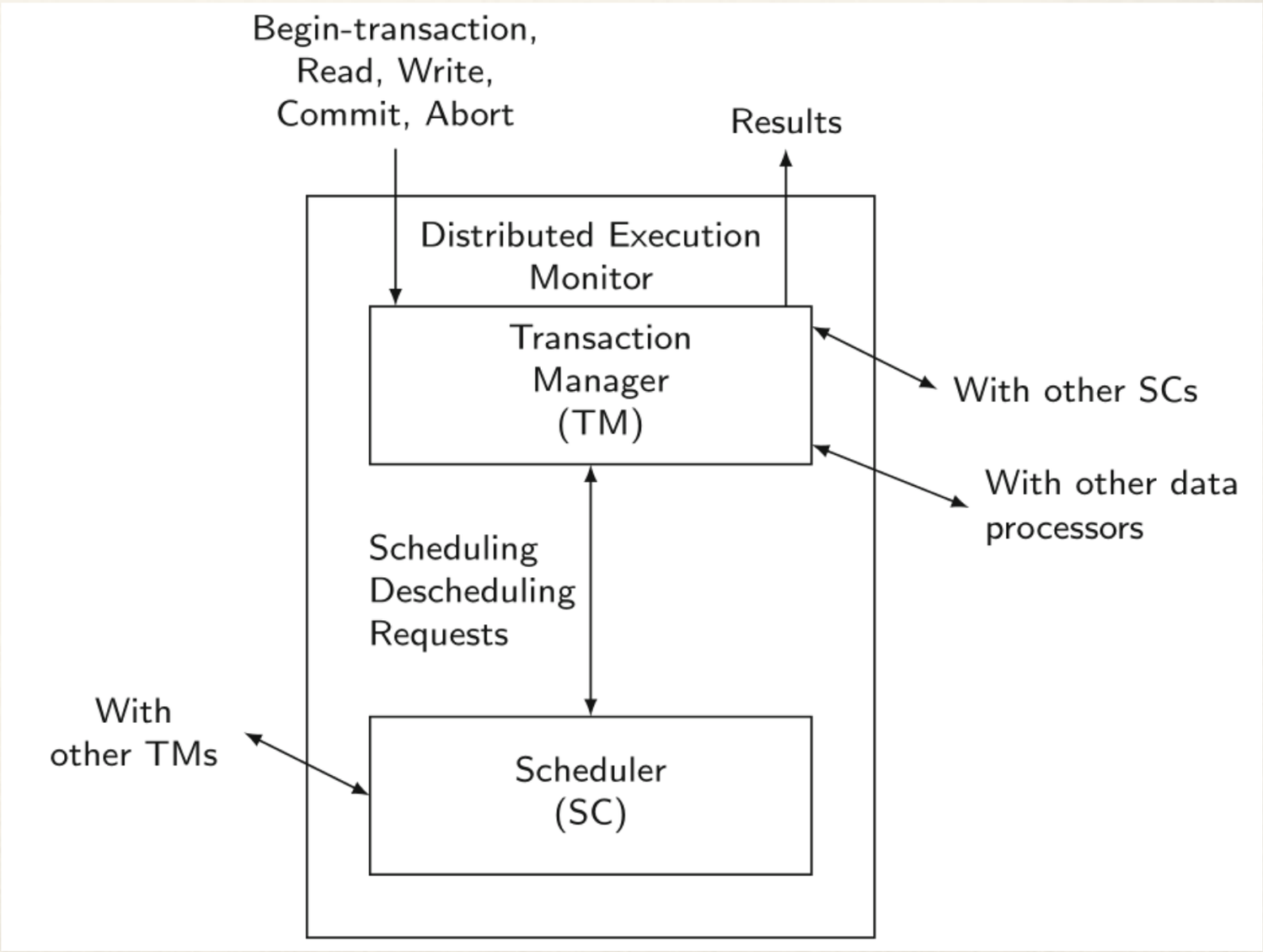
- CC algorithms enforce the **isolation property**
  - No interference among the concurrent txns.
    - Concurrent txn see a consistent db state and leave the db in a consistent state!
- CC algorithms implement a notion of "correct concurrent execution."
  - The most common correctness notion is serializability
    - History generated by concurrent exec of txns ≡ some serial history of txns.
  - Any serial execution order is, by definition, correct.
  - A more relaxed correctness notion called snapshot isolation (SI).
- CC algorithms are basically concerned with enforcing different levels of isolation among concurrent transactions very efficiently.

# Concurrency control algorithms

- When a txn commits, its actions need to be made permanent.
  - Management of txn logs where each action of a txn is recorded.
  - Commit protocols ensure that DB updates as well as logs are saved into persistent storage so that they are made permanent.
  - Abort protocols use the logs to erase all actions of the aborted transaction.
  - Recovery from system crashes uses logs to bring the DB to a consistent state.
- Distributed execution monitor consists of two modules:
  - A transaction manager (TM) and a scheduler (SC).
  - TM coordinates execution of the DB operations on behalf of an application.
  - SC is responsible for synchronizing access to the database.

# Concurrency control algorithms

- TM implements an interface (for the app) to the txn commands:
  - Begin_Transaction, Read, Write, Commit, and Abort. (BT,R,W,C,A)
  - TM can communicate with SCs and data processors (DP) at the same or at different sites.



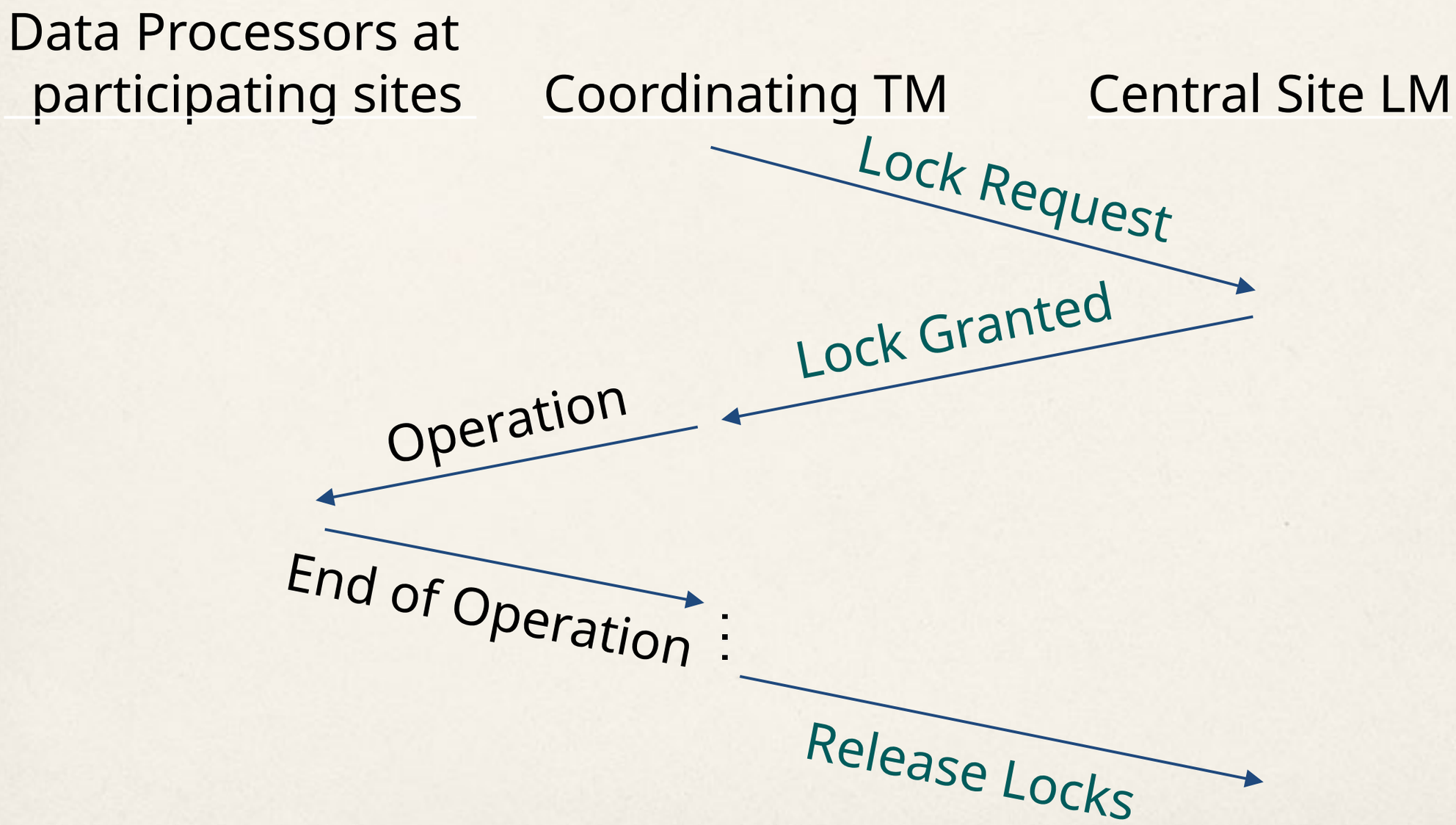
# Locking-Based CC Algorithms

- Prevent isolation violation by requiring each operation to obtain a lock on the data item before it is accessed.
- Lock can either be a read (shared) lock, or a write (exclusive) lock.
- Locks are usually maintained in a lock table.

# Centralized 2PL

- 2PL algorithm can be extended to the distributed DBMS environment
  - Delegating lock management to a single site.
- Each site has a Transaction Manager (TM)
- There is only one 2PL scheduler in the distributed system.
  - All lock requests are issued to the central scheduler.
  - TMs at the other sites communicate with it to obtain locks.
- Coordinating TM coordinates txns on behalf of an application.
  - C2PL-TM is a process that runs forever and waits for messages from an application, a lock manager, or from a data processor.
  - C2PL-LM and data processor (DP) algorithms are procedures that are called when needed (or, processes).
- Common criticism of C2PL algorithms
  - Bottleneck may quickly form around the central site.

# Centralized 2PL



# 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

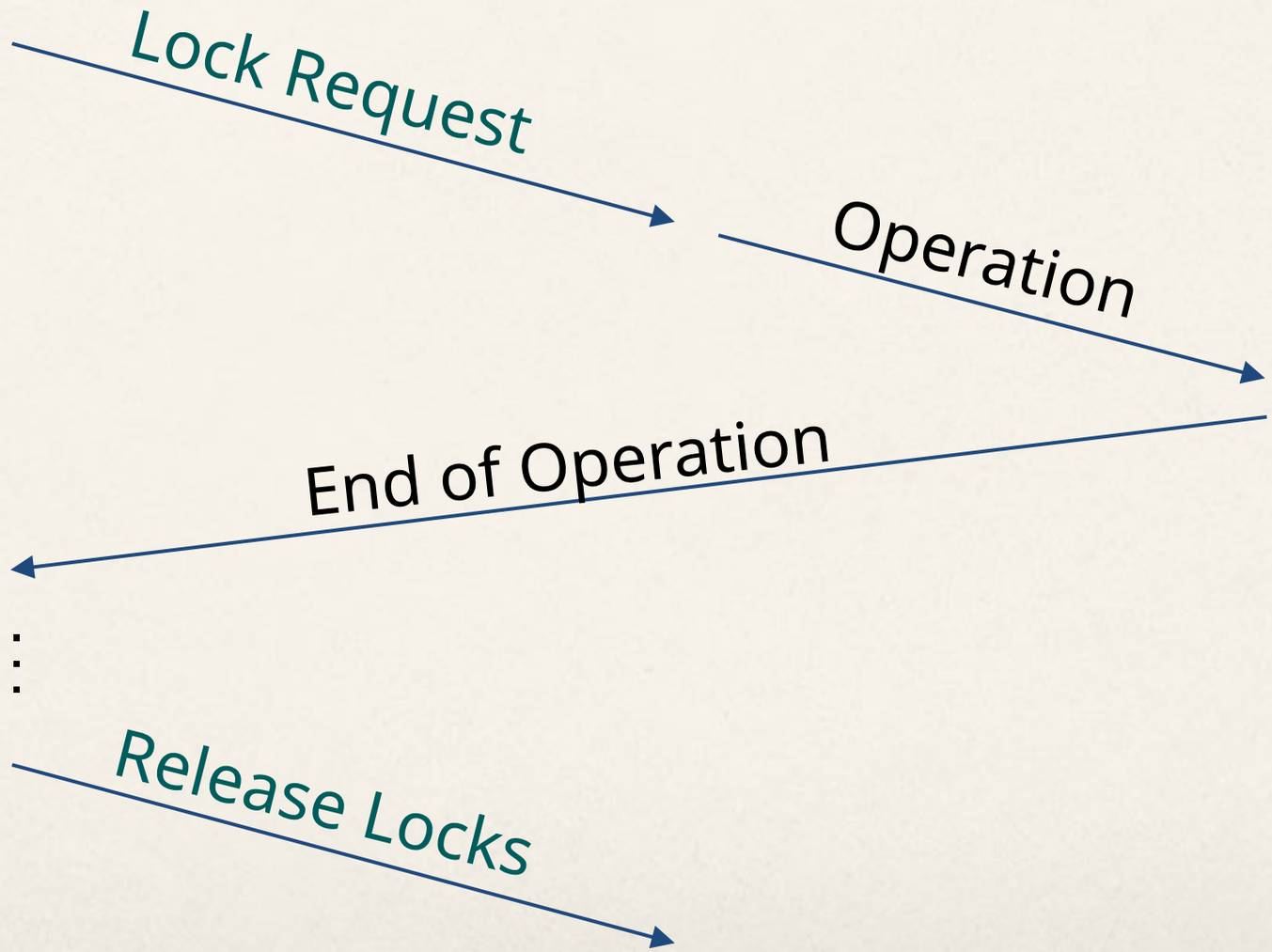
**Algorithm 11.3**: Data Processor (DP) Algorithm

```
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

- 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”
  - 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.

# Distributed 2PL Execution

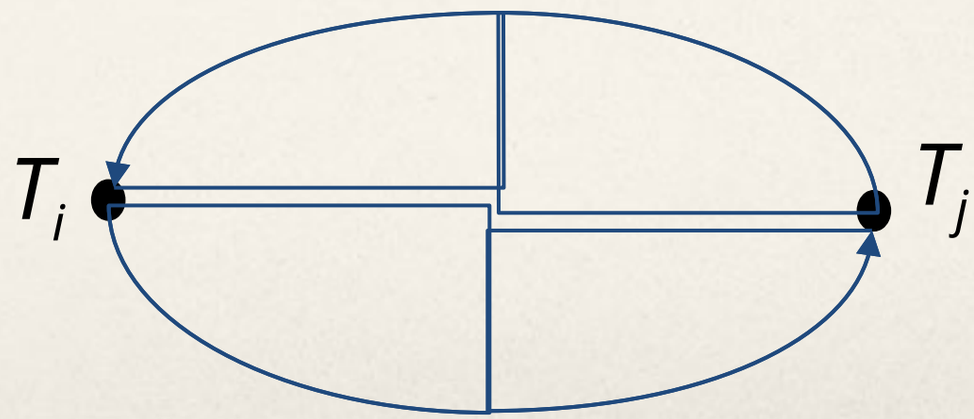


# Distributed 2PL

- 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*.
- Replication will be presented in the next lecture.

# 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.

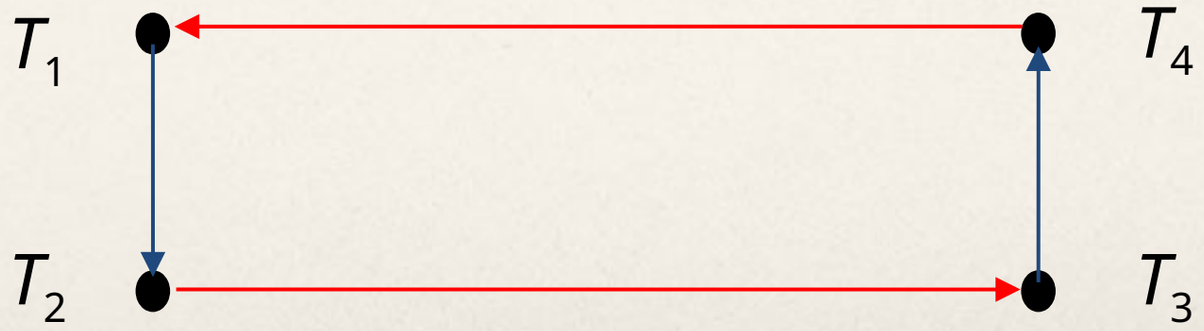


# Local versus Global WFG

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

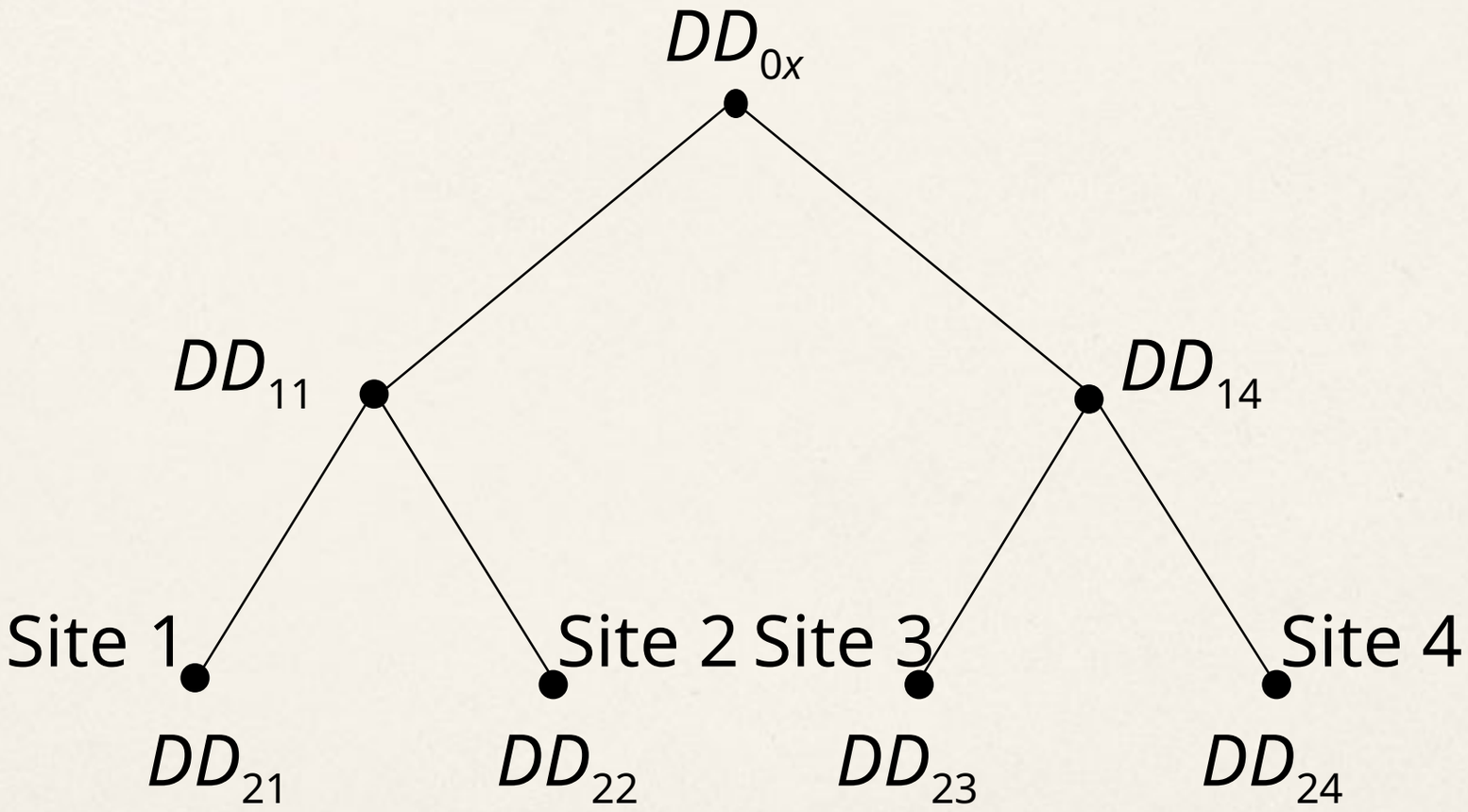
- One site is designated as the deadlock detector for the system. Each scheduler periodically sends its local WFG to the central site which merges them to a global WFG to determine cycles.
- How often to transmit?

  ➡️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.

# Timestamp-Based Algorithms

- Timestamp-based CC algorithms select, a priori, a serialization order.
  - … and txns are executed accordingly.
- Transaction manager (TM) assigns each txn Ti a unique TS, ts(Ti), at its initiation.
  - Maintaining uniqueness and monotonicity of TS-s is not easy.
    - Global (system-wide) monotonically increasing counter. (not realistic)
    - Each site autonomously assigns timestamps based on its local counter.
    - To maintain uniqueness, each site appends its own identifier.
    - TS = ⟨local counter value, site identifier⟩; (note most/least significant positions)
- TM attaches TS to each database operation (passed to scheduler).
  - Site identifiers are used for keeping track of R/W timestamps as well as performing the serializability check.

# Timestamp Ordering

1. Transaction ($T_i$) is assigned a globally unique timestamp $ts(T_i)$.
2. Transaction manager attaches the timestamp to all operations issued by the transaction.
3. 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*
4. Conflicting operations are resolved by timestamp order.

# Basic Timestamp Ordering

Two conflicting operations $O_{ij}$ of $T_i$ and $O_{kl}$ of $T_k \Longrightarrow O_{ij}$ executed before $O_{kl}$ iff $ts(T_i) < ts(T_k)$.

- $T_i$ is called older transaction
- $T_k$ is called younger transaction

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 $W_i(x)$
$wts(x) \leftarrow ts(T_i)$

# Basic TO Algorithm

- Basic TO algorithm runs on coordinating TM:
  - assigns the timestamp to each transaction Ti [ts(Ti )],
  - determines the sites where each data item is stored, and
  - sends the ops to these sites (R/W/C/A ops thru scheduler).
- The data manager is the same as in 2PL algorithm.
- When op is rejected by a scheduler, the corresponding txn is restarted by TM with a new TS.
  - More recent TS $\Longrightarrow$ more chances to execute in its next try.
  - Txns never wait while they hold access rights to data items $\Longrightarrow$ No deadlocks.
  - Penalty of deadlock freedom = potential restart of txn numerous times.
  - See later Conservative timestamp ordering algorithm.
- DP executes ops in the order in which scheduler passes them on.
  - Scheduler does not send new ops to DP until current is finished (accptd+acknw).
  - Scheduler can enforce the ordering by maintaining a queue.
  - In 2P algorithms locks are ordering txn by releasing them at the end of txn.

# Basic TO Algorithm

- Example:
  - TO scheduler first receives Wi(x) and then Wj(x), where ts(Ti) < ts(Tj).
  - The result of these two operations is that wts(x) = ts(Tj). (Wi(x) overwritten)
  - No ordering of ops $\Longrightarrow$ Different (wrong) outcome possible.

# BTO-TM

**Algorithm 5.4:** Basic Timestamp Ordering (BTO-TM)

```
Input: msg : a message
begin
    repeat
        wait for a msg
        switch msg type do
            case transaction operation do          {operation from application program }
                let op be the operation
                switch op.Type do
                    case BT do
                        S ← ∅ ;                    {S: set of sites where transaction executes }
                        assign a timestamp to transaction—call it ts(T)
                        DP(op)                                  {call DP with operation}
                    end case
                    case R, W do
                        find site that stores the requested data item (say S_i)
                        BTO-SC_{S_i}(op, ts(T)) ;          {send op and ts to SC at S_i }
                        S ← S ∪ S_i                {build list of sites where transaction runs}
                    end case
                    case A, C do                   {send op to DPs that execute transaction }
                        DP_S(op)
                    end case
                end switch
            end case
            case SC response do                    {operation must have been rejected by a SC}
                op.Type ← A;                                    {prepare an abort message}
                BTO-SC_S(op, −) ;                               {ask other participating SCs}
                restart transaction with a new timestamp
            end case
            case DP response do                            {operation completed message}
                switch transaction operation type do
                    let op be the operation
                    case R do return op.val to the application
                    case W do inform application of completion of the write
                    case C do
                        if commit msg has been received from all participants then
                            inform application of successful completion of transaction
                        else                       {wait until commit messages come from all}
                            record the arrival of the commit message
                        end if
                    end case
                    case A do
                        inform application of completion of the abort
                        BTO-SC(op)                          {need to reset read and write ts}
                    end case
                end switch
            end case
        end switch
    until forever
end
```

# BTO-SC

**Algorithm 5.5:** Basic Timestamp Ordering Scheduler (BTO-SC)

```
Input: op : Op; ts(T) : Timestamp
begin
    retrieve rts(op.arg) and wts(arg)
    save rts(op.arg) and wts(arg) ;                    {might be needed if aborted }
    switch op.arg do
        case R do
            if ts(T) > wts(op.arg) then
                DP(op) ;                               {operation can be executed; send it to DP}
                rts(op.arg) ← ts(T)
            else
                send "Reject transaction" message to coordinating TM
            end if
        end case
        case W do
            if ts(T) > rts(op.arg) and ts(T) > wts(op.arg) then
                DP(op) ;                               {operation can be executed; send it to DP}
                rts(op.arg) ← ts(T)
                wts(op.arg) ← ts(T)
            else
                send"Reject transaction" message to coordinating TM
            end if
        end case
        case A do
            forall op.arg that has been accessed by transaction do
                reset rts(op.arg) and wts(op.arg) to their initial values
            end forall
        end case
    end switch
end
```

# 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

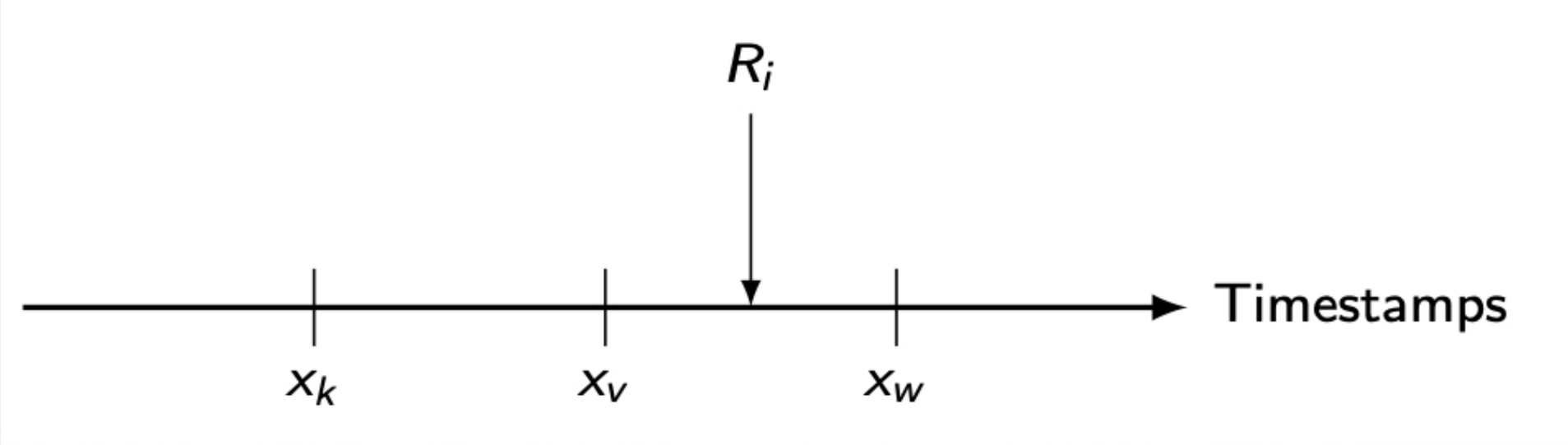
# Multiversion Concurrency Control (MVCC)

- Do not modify the values in the database, create new values.
- Another attempt at eliminating the restart overhead of txn
  - Maintaining multiple versions of data items and
  - Scheduling operations on the appropriate version of the data item
- Time travel queries -- track the change of data item values over time.
  - Proliferation of multiple versions of updated data items $\Longrightarrow$ Purging old versions
- MVCC typically use timestamps to maintain transaction isolation
  - There are also implementations with locking-based CC.
- Original proposal dates back to 1978.
  - Implemented in a number of systems:
  - IBM DB2, Oracle, SQL Server, SAP HANA, BerkeleyDB, PostgreSQL
- We will focus on timestamp-based implementation that enforces serializability

# MVCC Reads

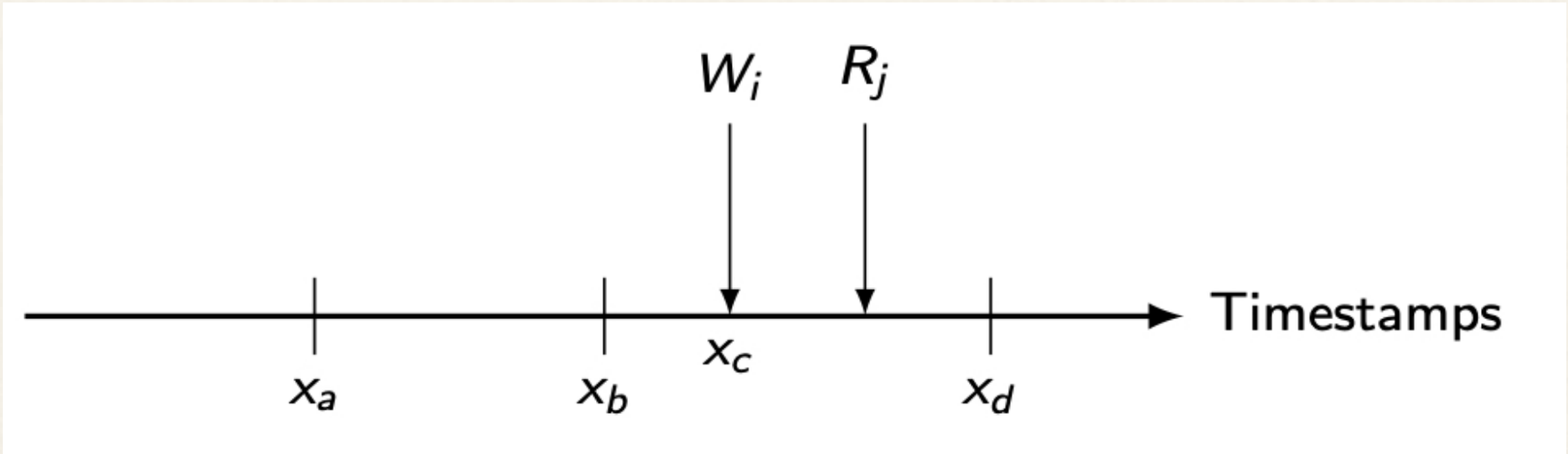
- Each version of a data item is labeled with TS of txn that creates it.
  - Read op accesses the version that is appropriate for its timestamp.
  - Reducing transaction aborts and restarts.
- The use of the following two rules is guaranteeing a serializable history.

1) $R_i(x)$ 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 $ts(T_i)$.



# MVCC Writes

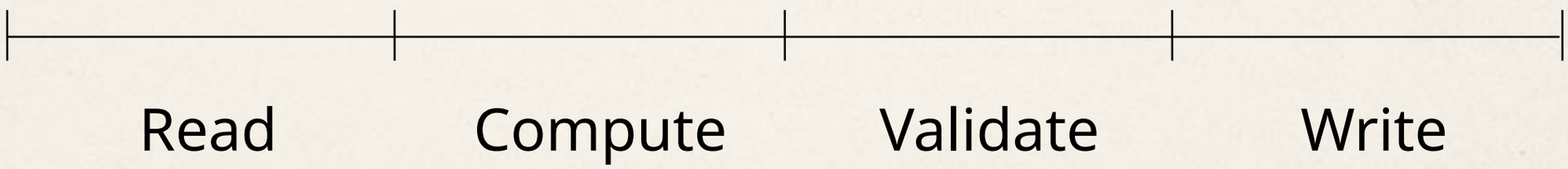
2) $W_i(x)$ is translated into $W_i(x_w)$ and accepted if the scheduler has not yet processed any $R_j(x_r)$ such that: $ts(T_i) < ts(x_r) < ts(T_j)$. Otherwise, txn is rejected.



# 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

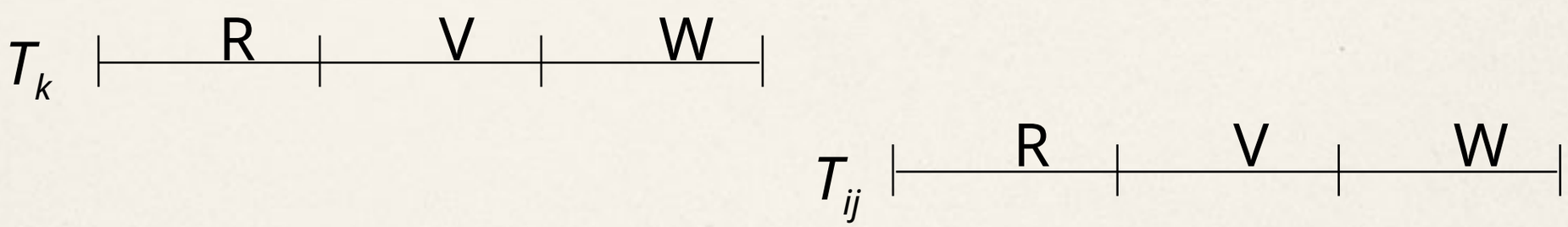
- All subtransactions are assigned a timestamp at the end of their read phase

- Validation test performed during validation phase. If one fails, all rejected.

# Optimistic CC Validation Test

1. 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

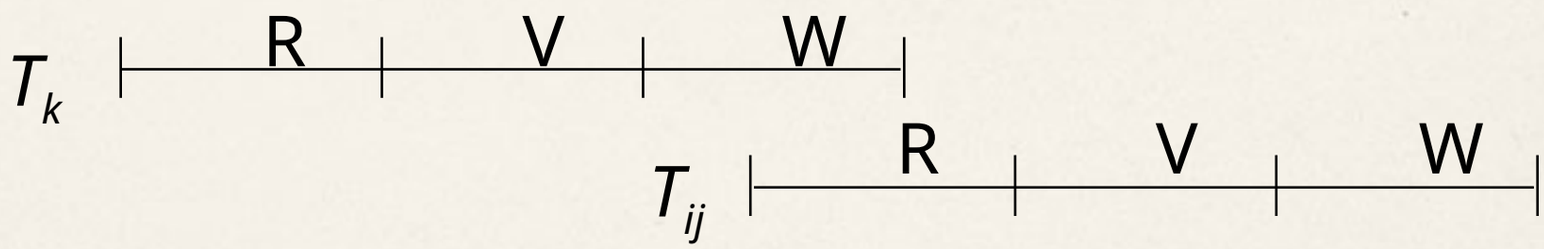
   ➡️ Transaction executions in serial order



# Optimistic CC Validation Test

2. 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 $WS(T_k) \cap RS(T_{ij}) = \emptyset$

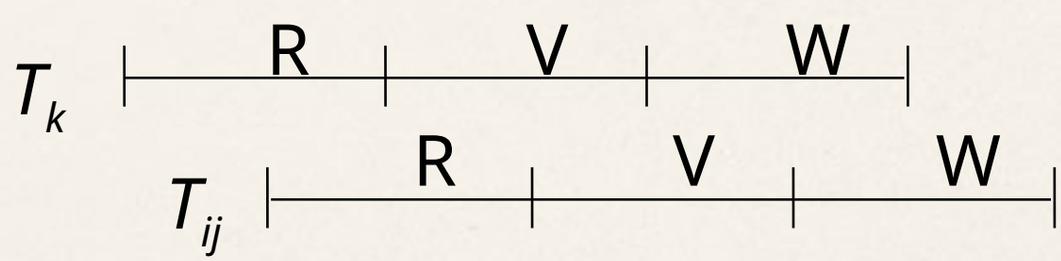
   ➡️ Read and write phases overlap, but $T_{ij}$ does not read data items written by $T_k$



# 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 $WS(T_k) \cap RS(T_{ij}) = \emptyset$ and $WS(T_k) \cap WS(T_{ij}) = \emptyset$

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



# Snapshot Isolation (SI)

- Serializability is the most studied correctness criterion for concurrent txns
  - Too strict for some applications (creates a bottleneck)
  - The main reason: large read queries conflict with updates
- Snapshot isolation (SI) as an alternative
  - Repeatable reads, but not serializable isolation
- Each txn “sees” a consistent snapshot of db when it starts, and R|W this snapshot
  - Reads and writes are performed on this snapshot
  - Read-only txns proceed without significant synchronization overhead
    - Writes are not visible to other txns
    - Txn does not see the writes of other transactions once it starts executing

# Snapshot Isolation (SI)

- MV approach allowing txns to read approp. snapshot
  - RO txns can proceed without significant synchronization overhead
  - For update txns, CC algorithm (in centralized systems) is as follows
- Centralized SI-based CC
  1) $T_i$ starts, obtains a begin timestamp $ts_b(T_i)$
  2) $T_i$ ready to commit, obtains a commit timestamp $ts_c(T_i)$ that is greater than any of the existing $ts_b$ or $ts_c$
  3) $T_i$ commits if no other $T_j$ such that $ts_c(T_j) \in [ts_b(T_i), ts_c(T_i)]$; otherwise aborted (first committer wins)
     - No other transaction has committed since Ti started!
  4) When $T_i$ commits, changes visible to all $T_k$ where $ts_b(T_k) > ts_c(T_i)$
- Problem:
  - How to compute the consistent snapshot (version) on which txn Ti operates

# Distributed CC with SI

- Computing a consistent **distributed snapshot** is hard
- If read and write sets of txn are known up-front
  - May be possible to centrally compute the snapshot (at coordinating TM) by collecting information from the participating sites.
  - Not realistic.
- Similar rules to serializability
  - Each local history should be SI
  - Global history is SI $\Longrightarrow$ commitment orders at each site are the same
- Dependence relationship: $T_i$ at site s ($T_i^s$) is dependent on $T_i^s$ (*dependent(,)*) iff
  - Conditions that need to be satisfied for the above guarantee

$$(RS(T_i^s) \cap WS(T_j^s) \neq \emptyset) \vee (WS(T_i^s) \cap RS(T_j^s) \neq \emptyset) \vee (WS(T_i^s) \cap WS(T_j^s) \neq \emptyset)$$

  - If there is any participating site where this dependence holds, then *dependent* ($T_i$,$T_j$) holds.

# Distributed CC with SI

- Conditions that need to hold to ensure global SI:
  - For a transaction $T_i$ to see a globally consistent snapshot, the following conditions have to hold for each pair of transactions

1) $dependent(T_i, T_j) \wedge ts_b(T_i^s) < ts_c(T_j^s) \Rightarrow ts_b(T_i^t) < ts_c(T_j^t)$ at every site $t$ where $T_i$ and $T_j$ execute together
2) $dependent(T_i, T_j) \wedge ts_c(T_i^s) < ts_c(T_j^s) \Rightarrow ts_c(T_i^t) < ts_b(T_j^t)$ at every site $t$ where $T_i$ and $T_j$ execute together
3) $ts_c(T_i^s) < ts_c(T_j^s) \Rightarrow ts_c(T_i^t) < ts_b(T_j^t)$ at every site $t$ where $T_i$ and $T_j$ execute together

- (1) and (2) ensure that *dependent($T_i$, $T_j$)* is true at all the sites.
- (3) ensures commit order among transactions is the same at all participating sites.
  - Prevents 2 snapshots from including partial incompatible commits.

# Distributed CC with SI

- Before presenting distributed SI CC, we identify the information that each site s maintains:
  - For any active transaction $T_i$, the set of active and committed transactions at s are categorized into two groups:
    1) Those that are concurrent with $T_i$: any $T_j$ where $ts_b(T_i^s) < ts_c(T_j^s)$),
    2)Those that are serial: any $T_j$ where $ts_c(T_j^s) < ts_b(T_i^s)$)
  - A monotonically increasing event clock.
- Basic distributed SI algorithm implements step (3) of the centralized SI CC algorithm
  - It decides whether transaction $T_i$ can be committed or needs to be aborted

# Basic distributed SI algorithm

1) Coordinating TM (CTM) of Ti asks ∀s to send its set of txns concurrent with Ti . (Event clock is piggybacked.)
2) ∀s responds to the CTM with local set of txns concurrent with Ti.
3) CTM merges all local concurrent txns sets into global set for Ti.
4) CTM sends global list of concurrent tran to ∀s.
5) ∀s checks C1 and C2:
   - Check ∀Tj in local history that: Tj executed before Ti, and dependent(Ti, Tj) holds.
   - If that is the case, Ti does not see a consistent snapshot at site s, so it should be aborted. Otherwise Ti is validated at site s.
6) ∀s sends validation to CTM.
   - If validation positive, max(local clock, TM clock) is piggybacked.
7) CTM receives one negative validation: Ti is aborted,
   - At least one site where Ti does not see a consistent snapshot.
   - Otherwise, CTM globally certifies Ti and allows it to commit its updates.
   - Further, CTM sets clock to max(clocks it received, own clock).

# Basic distributed SI algorithm

8) CTM informs all participating sites that Ti is validated and can be committed. (Piggybacks its new event clock $ts_c(Ti)$.
9) Upon receipt of this message, ∀s makes Ti ’s updates persistent, and also updates its own event clock as before.

# Distributed CC with SI – Executing