Distributed concurrency control

Dario Della Monica

These slides are a modified version of the slides provided with the book

Özsu and Valduriez, *Principles of Distributed Database Systems* (3rd Ed.), 2011

The original version of the slides is available at: extras.springer.com

Outline (distributed DB)

- Introduction (Ch. 1) ⋆
- Distributed Database Design (Ch. 3) ⋆
- Distributed Query Processing (Ch. 6-8) ⋆
- Distributed Transaction Management (Ch. 10-12) ⋆
    - ➡ Introduction to transaction management (Ch. 10) ⋆
    - ➡ **Distributed Concurrency Control (Ch. 11)** ⋆
    - ➡ Distributed DBMS Reliability (Ch. 12) ⋆

⋆ Özsu and Valduriez, *Principles of Distributed Database Systems* (3rd Ed.), 2011

Outline (today)

- Distributed Concurrency Control (Ch. 11) [★]
  - Serializability Theory
    - Formalization/Abstraction of Transactions
    - Formalization/Abstraction of Concurrent Transactions (Histories)
    - Serial Histories
  - Locking-based
    - (strict) 2-phase Locking (2PL)
  - Deadlock management

[★] Özsu and Valduriez, *Principles of Distributed Database Systems* (3rd Ed.), 2011

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 has to do with C(onsistency) and I(solation) from the ACID properties
- *Consistency*: assuming that each transaction is internally consistent (no integrity constraint violations) it is obtained by guaranteeing the right level of *isolation* (serializability)
- *Isolation*: isolating transactions from one another in terms of their effects on the DB. More precisely, in terms of the effect on the DB of intermediate operations (before commit)
- Tradeoff between isolation and parallel execution (concurrency)
- Assumptions
    - ➙ System is fully reliable (no failures) – we deal with reliability in Ch. 12[★]
    - ➙ No data replication – discussion on data replication is in Ch. 13[★] (we do not cover this chapter)
- Possible 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

---

[★] Özsu and Valduriez, *Principles of Distributed Database Systems* (3rd Ed.), 2011

POSET's to Model Transactions

- We treat a transaction as a POSET (aka partially ordered set, partial order)
- POSET's are pairs $< \Sigma, \prec >$ where
  - Σ is a set (domain)
  - $\prec$ is a binary relation over Σ ($\prec \subseteq \Sigma \times \Sigma$) that is
    - irreflexive (not $a \prec a$, for all a)
    - asymmetric ($a \prec b$ implies not $b \prec a$, for all a,b)
    - transitive ($a \prec b$ and $b \prec c$ implies $a \prec c$, for all a,b,c)
- Operations are
  - DB operations (read or write): $R(x)$, $W(x)$ (where x is a data entity, e.g., a tuple) or
  - termination conditions (abort or commit): A, C
- A transaction is modeled as a partially ordered set of operations containing exactly one termination condition

Formalization/Abstraction of Transactions

- A transition is a POSET $T = < \sum , \prec >$ where
  - ➞ $\sum$ is finite: it is the set of operations of T
    - ✦ O is the set of DB operations in $\sum$ (operation that are not termination conditions)
      - ✓ i.e., elements of O are of the kind $R(x)$, $W(x)$ where x is a data entity
    - ✦ Thus, $\sum = O \cup \{ N \}$, where $N \in \{ A, C \}$ (exactly 1 termination condition)
    - ✦ 2 DB operations (elements of O) conflict iff they act on the same data entity x and one of them is a write W operation
      - ✓ $W(x)$, $R(x)$ are in conflict, $W(x)$, $W(x)$ are in conflict
      - ✓ $W(x)$, $R(y)$ are NOT in conflict, $W(x)$, $W(y)$ are NOT in conflict, $R(x)$, $R(x)$ are NOT in conflict
    - ✦ NOTICE: it is possible to have 2 distinct $W(x)$ operations
      - ✓ We assume implicit indices to make every operation unique
    - ✦ Operations are atomic (indivisible units)
  - ➞ $\prec$ is s.t.
    - ✦ order of conflicting operation is specified
      - ✓ for all $o_1, o_2 \in O$: if o_1 and o_2 conflict, then either $o_1 \prec o_2$ or $o_2 \prec o_1$
    - ✦ all DB operations precede the unique termination condition
      - ✓ for all $o \in O$: $o \prec N$

Formalization/Abstraction of Transactions – cont’d

- POSET’s are DAG (directed acyclic graphs)
- We represent a transaction either way (as a POSET or as a DAG)
- The order of 2 conflicting operations is important and **MUST** be specified
  - → it specify the execution order between the 2 operations
- Operations that are not related can be executed in parallel
- A transaction might force other precedence order relations besides the ones between conflicting operations
- These depend on application semantics

Transaction *T*:

Read(x)
Read(y)
$x \leftarrow x + y$
Write(x)
Commit

POSET representation of *T*:

- $\Sigma = \{R(x), R(y), W(x), C\}$
- $\prec = \{ (R(x), W(x)), (R(y), W(x)), (W(x), C), (R(x), C), (R(y), C) \}$

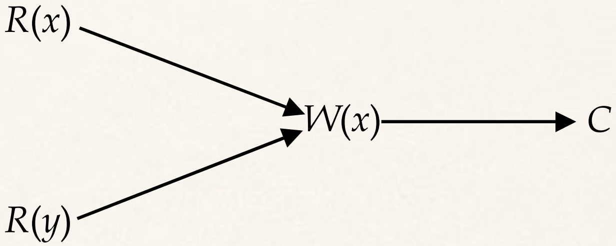
POSET representation abstracts away application (non-DB) operations (e.g., $x \leftarrow x + y$)

DAG Representation

Let $T = \{ R(x) \prec W(x)\ ,\ R(y) \prec W(x)\ ,\ W(x) \prec C\ ,\ R(x) \prec C\ ,\ R(y) \prec C \}$

(compact representation of a POSET)

Corresponding ***DAG representation***:



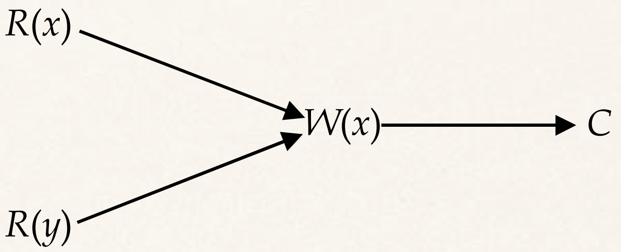
Transactions are DAG's (directed acyclic graphs)

DAG Representation

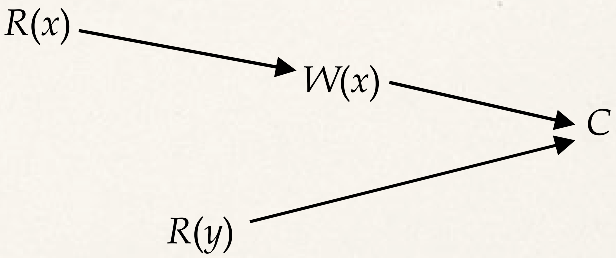
Let $T = \{ R(x) \prec W(x) \ , \ R(y) \prec W(x) \ , \ W(x) \prec C \ , \ R(x) \prec C \ , \ R(y) \prec C \}$

(compact representation of a POSET)

Corresponding ***DAG representation***:



Let $T = \{ R(x) \prec W(x) \ ,$ ~~$R(y) \prec W(x)$~~ $, \ W(x) \prec C \ , \ R(x) \prec C \ , \ R(y) \prec C \}$

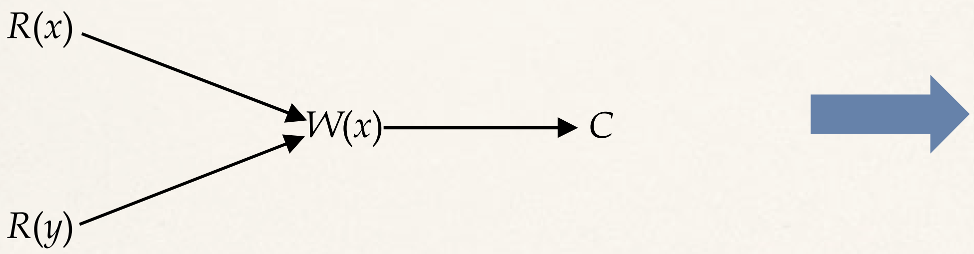


Order of $R(y)$ and $W(x)$ is irrelevant
(order of 2 read operations is always irrelevant)

Transactions are DAG's (directed acyclic graphs)

Transactions as Executions

- There is at least one (possibly several), at least one, linear orders (aka total orders) *compatible with* (i.e., *extending*) any given partial order
- Each of them is a possible execution of the transaction
- Therefore, a transaction that is a **linear order** is a transaction execution



$$\left\{\begin{array}{l} R(x) \prec R(y) \prec W(x) \prec C \\ \\ R(y) \prec R(x) \prec W(x) \prec C \end{array}\right.$$

From Transactions to Histories

- **Informal definition:** A **history** is defined over a set of transactions and specifies possible interleaved executions of transactions in such set
- The formalization of transaction as POSET's can be extended to sets of transactions to define histories
- **Formal definition:**
  - Extend the notion of *conflicting operations* to pairs of operations O_i , O_j belonging to different transactions
  - Given a set $T = \{ T_1, \ldots, T_n \}$ of transactions
    - (where $T_i = < \Sigma_i , \prec_i >$ for all i – we assume $\Sigma_i \cap \Sigma_j = \varnothing$ for all $i \neq j$)

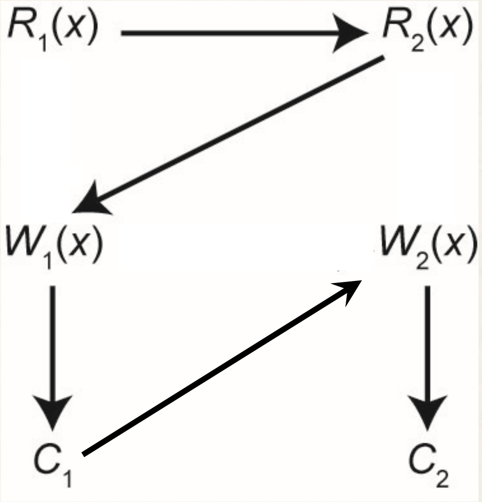
    A history H over T is a pair $H = < \Sigma , \prec >$ where
    - $\Sigma = \cup_{T_i \in T} \Sigma_I$ is a finite set of read/write operations plus one termination condition (C or A) for each transaction)
    - $\prec \supseteq \cup_{T_i \in T} \prec_i$ is a partial order that extends $\prec_i$ by including precedence constraints for conflicting operations belonging to different transactions (and, possibly, more precedence constraints for pairs of operations)
    - Still, the order of 2 conflicting operations is important and **MUST** be specified. Therefore
      - for all $o_i \in \Sigma_i$ and all $o_j \in \Sigma_j$ $(i \neq j)$: if o_i and o_j conflict, then either $o_i \prec o_j$ or $o_j \prec o_i$
- A history that is a **linear order** is a concurrent transaction execution
- In the book [★] the term *complete history* is used for what we call here *history*
  - because they use *histories* to refer to prefixes of histories (partial histories)

[★] Özsu and Valduriez, *Principles of Distributed Database Systems* (3rd Ed.), 2011

History – Example

T_1: $R(x)$
$x \leftarrow x + 1$
$W(x)$
C

T_2: $R(x)$
$x \leftarrow x + 1$
$W(x)$
C



A history over $T = \{ T_1 , T_2 \}$ is the partial order:

$H = \{ R_1(x) \prec R_2(x) \prec W_1(x) \prec C_1 \prec W_2(x) \prec C_2 \}$

H is actually a linear order, so it is a *concurrent execution* of transactions T_1 and T_2

Serial History

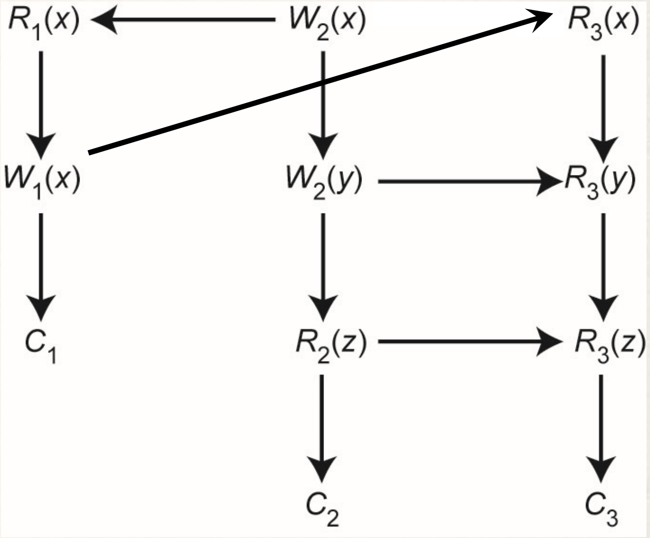
- A **serial history** (or **serial execution of concurrent transactions** or **serial execution**) is a concurrent transaction execution where operations of different transaction do not interleave
- A serial history defines a **linear order over transactions**, too (**serialization order**)

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 serial history over $T = \{ T_1, T_2 \}$ compatible with H is the linear order (we omit the termination conditions C_i):

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

$$T_2 \prec T_1 \prec T_3$$



A history H over $T = \{ T_1, T_2, T_3 \}$

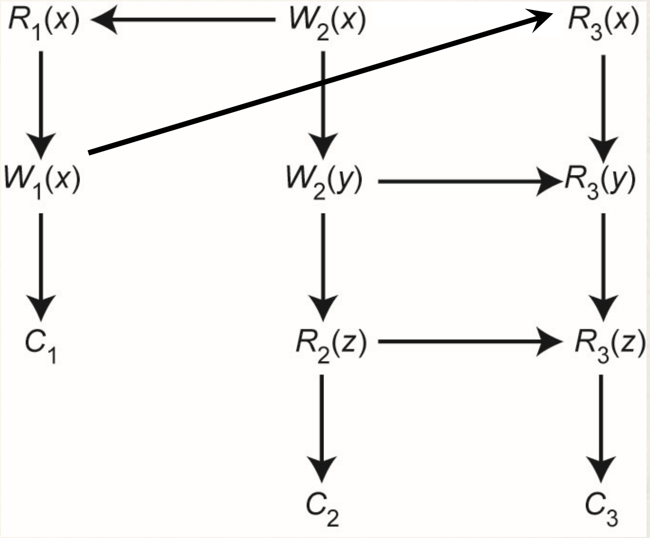
Serial History

- A **serial history** (or **serial execution of concurrent transactions** or **serial execution**) is a concurrent transaction execution where operations of different transaction do not interleave
- A serial history defines a **linear order over transactions**, too (**serialization order**)

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 serial history over $T = \{ T_1, T_2 \}$ compatible with H is the linear order (we omit the termination conditions C_i):

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



A history H over $T = \{ T_1, T_2, T_3 \}$

A serial history preserves DB consistency (as transactions, when executed singularly, brings DB from a consistent state to another consistent state)

Conflict equivalence

- **Definition.** Two histories over the same set of transactions are **conflict equivalent** (or, simply **equivalent**) iff they agree on the execution order of the conflicting operations
    - $H_1 = < \Sigma_1 , \prec_1 >$ and $H_2 = < \Sigma_2 , \prec_2 >$ with O and O′ conflicting operations
        - then $O \prec_1 O'$ if and only if $O \prec_2 O'$

  (we are ignoring abort transaction to keep definition simpler)
- $H' = \{ W_2(x) \prec_{H'} W_2(y) \prec_{H'} R_2(z) \prec_{H'} R_1(x) \prec_{H'} W_1(x) \prec_{H'} R_3(x) \prec_{H'} R_3(y) \prec_{H'} R_3(z) \}$
    - is NOT equivalent to $H_1 = \{ W_2(x) \prec_{H_1} R_1(x) \prec_{H_1} R_3(x) \prec_{H_1} W_1(x) \prec_{H_1} R_3(y) \prec_{H_1} R_3(z) \prec_{H_1} W_2(y) \prec_{H_1} R_2(z) \}$
        - because $W_1(x) \prec_{H'} R_3(x)$ in H' **but** $R_3(x) \prec_{H_1} W_1(x)$ in H_1
    - is equivalent to $H_2 = \{ W_2(x) \prec_{H_2} R_1(x) \prec_{H_2} W_1(x) \prec_{H_2} R_3(x) \prec_{H_2} W_2(y) \prec_{H_2} R_3(y) \prec_{H_2} R_2(z) \prec_{H_2} R_3(z) \}$

  (actually H', H_1, H_2 are all concurrent transaction executions)
- **Definition.** A history is **serializable** iff it is equivalent to a serial execution
    - Therefore, H_2 is *serializable* (because H_2 is equivalent to H', which is a serial history)

Conflict equivalence

- **Definition.** Two histories over the same set of transactions are **conflict equivalent** (or, simply **equivalent**) iff they agree on the execution order of the conflicting operations
  - $H_1 = < \Sigma_1 , \prec_1 >$ and $H_2 = < \Sigma_2 , \prec_2 >$ with O and O′ conflicting operations
    - then $O \prec_1 O'$ if and only if $O \prec_2 O'$

  (we are ignoring abort transaction to keep definition simpler)
- $H' = \{ W_2(x) \prec_{H'} W_2(y) \prec_{H'} R_2(z) \prec_{H'} R_1(x) \prec_{H'} W_1(x) \prec_{H'} R_3(x) \prec_{H'} R_3(y) \prec_{H'} R_3(z) \}$
  - is NOT equivalent to $H_1 = \{ W_2(x) \prec_{H_1} R_1(x) \prec_{H_1} R_3(x) \prec_{H_1} W_1(x) \prec_{H_1} R_3(y) \prec_{H_1} R_3(z) \prec_{H_1} W_2(y) \prec_{H_1} R_2(z) \}$
    - because $W_1(x) \prec_{H'} R_3(x)$ in H' **but** $R_3(x) \prec_{H_1} W_1(x)$ in H_1
  - is equivalent to $H_2 = \{ W_2(x) \prec_{H_2} R_1(x) \prec_{H_2} W_1(x) \prec_{H_2} R_3(x) \prec_{H_2} W_2(y) \prec_{H_2} R_3(y) \prec_{H_2} R_2(z) \prec_{H_2} R_3(z) \}$

  (actually H', H_1, H_2 are all concurrent transaction executions)
- **Definition.** A history is **serializable** iff it is equivalent to a serial execution
  - Therefore, H_2 is *serializable* (because H_2 is equivalent to H', which is a serial history)

Primary function of a **concurrency controller** is to **produce a serializable history** over the set of pending transactions

Serializability in Distributed DBMS

- Somewhat more involved. Two histories have to be considered:
  - local histories: histories over sets of transactions at the same site
  - global history: union of local histories
- For global transactions (i.e., global history) to be serializable, two conditions are necessary:
  - Each local history should be serializable
  - Identical local serialization order

★ Özsu and Valduriez, *Principles of Distributed Database Systems* (3rd Ed.), 2011

Example

- T_1 transfers 100 € from bank account x to y
- T_2 reads balance of x and y
- x is stored at site s_1
- y is stored at site s_2
- site s_1: $T_{1,s_1} = \{ R_1(x) \prec W_1(x) \}$ $\quad T_{2,s_1} = \{ R_2(x) \}$
  site s_2: $T_{1,s_2} = \{ R_1(y) \prec W_1(y) \}$ $\quad T_{2,s_2} = \{ R_2(y) \}$
- Local history at site s_1 : $H_{s_1} = \{ R_1(x) \prec W_1(x) \prec R_2(x) \}$
  - H_{s_1} is **serial** with serialization order $\boldsymbol{T_1 \prec T_2}$
- Local history at site s_2 : $H_{s_2} = \{ R_1(y) \prec W_1(y) \;,\; R_2(y) \prec W_1(y) \}$
  - H_{s_2} is **locally serializable**: $R_2(y) \prec R_1(y) \prec W_1(y)$
  - but not globally (serialization order $\boldsymbol{T_2 \prec T_1}$)

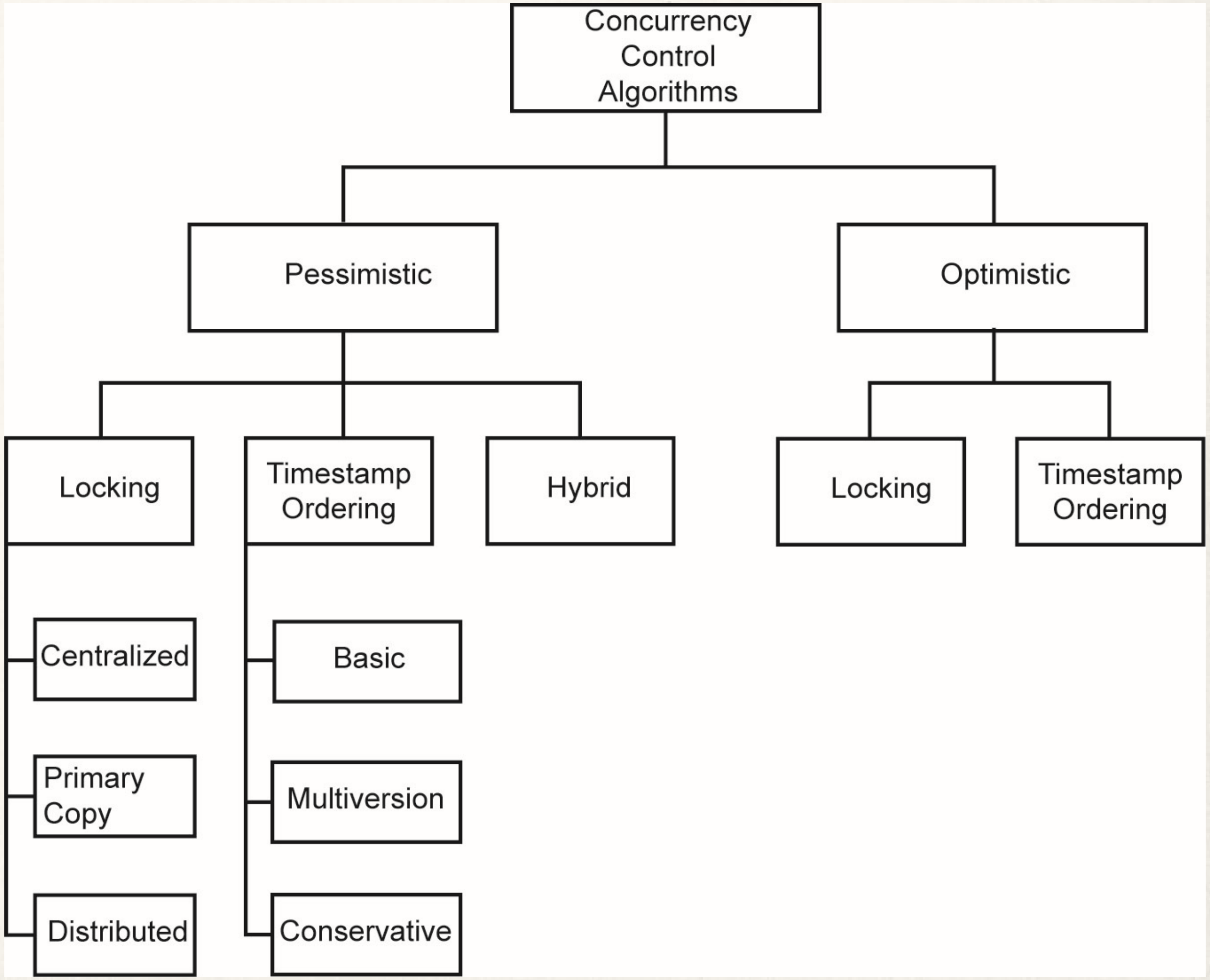
T_1:		T_2:	
	Read(x)		Read(x)
	$x \leftarrow x$-100		Read(y)
	Write(x)		Commit
	Read(y)		
	$y \leftarrow y$+100		
	Write(y)		
	Commit		

★ Özsu and Valduriez, *Principles of Distributed Database Systems* (3rd Ed.), 2011

Taxonomy of concurrency control mechanism

Classification

- based on synchronization primitives
  - locking vs. timestamp vs. hybrid
- pessimistic vs. optimistic

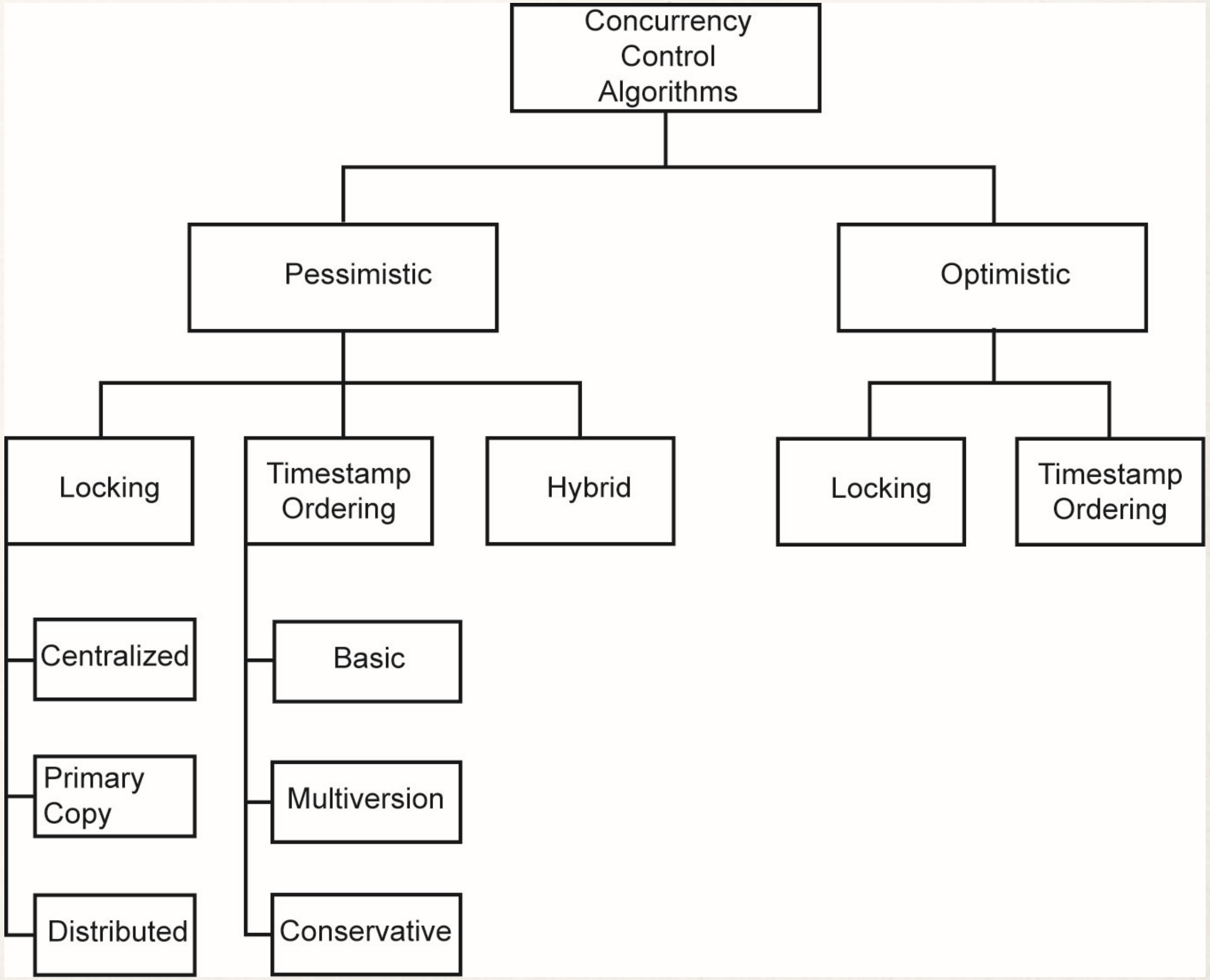


Taxonomy of concurrency control mechanism

Classification

- based on synchronization primitives
  - locking vs. timestamp vs. hybrid
- pessimistic vs. optimistic

We consider locking-based algorithm in the pessimistic scenario



- Concurrency Control Algorithms
  - Pessimistic
    - Locking
      - Centralized
      - Primary Copy
      - Distributed
    - Timestamp Ordering
      - Basic
      - Multiversion
      - Conservative
    - Hybrid
  - Optimistic
    - Locking
    - Timestamp Ordering

Locking-Based Algorithms

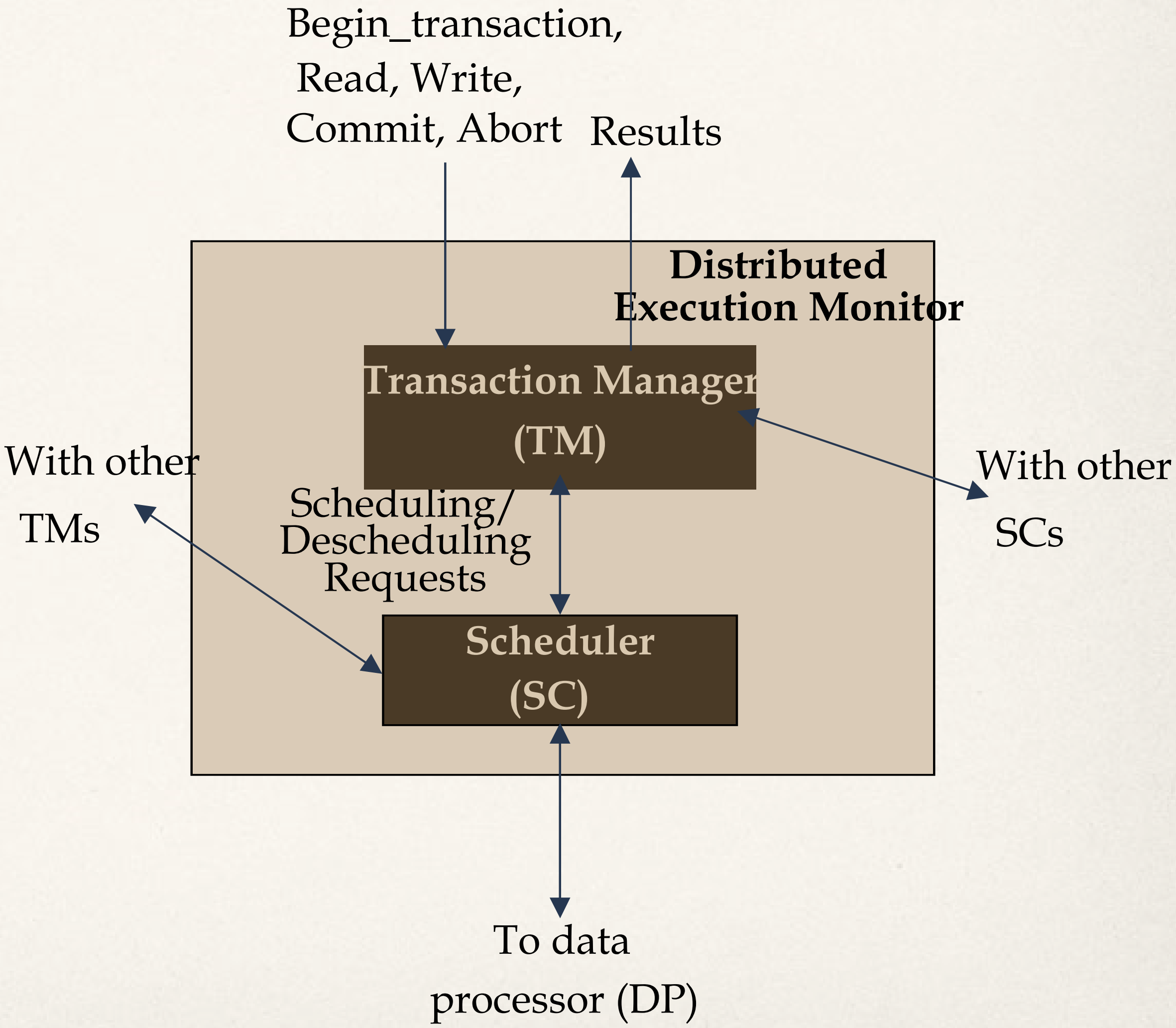
- Goal: to generate serializable histories
- Transactions indicate their intentions to read/write data item x by requesting suitable locks from the scheduler (called lock manager)
- Locks are either read lock (*rl*) [also called shared lock] or write lock (*wl*) [also called exclusive lock]
- Read locks and write locks conflict (because so do Read and Write operations)

	rl	*wl*
rl	no	yes
wl	yes	yes

- Locking works nicely to allow concurrent processing of transactions

Locking-Based Mechanism

- TM handles R/W requests coming from applications and passes them to SC (op. type R/W, transaction id., data unit)
- SC decides when to grant the lock according to compatibility access rules
- When SC grants lock, the DP executes the operation (R/W)
- SC is asked to release the lock
- TM is informed of successful operation
- Locking mechanism does not guarantee serializability (serial/serializable histories)
  - ➙ 2-phase locking (2PL) is the solution



Locking-Based Mechanism - Example

$$
\begin{array}{ll}
T_1: & R_1(x) \\
 & x \leftarrow x+1 \\
 & W_1(x) \\
 & R_1(y) \\
 & y \leftarrow y\text{-}1 \\
 & W_1(y) \\
 & C_1 \\
 & \downarrow
\end{array}
$$

$$
T_1 = \{ R_1(x) \prec W_1(x),\ R_1(y) \prec W_1(y) \}
$$

$$
\begin{array}{ll}
T_2: & R_2(x) \\
 & x \leftarrow x*2 \\
 & W_2(x) \\
 & R_2(y) \\
 & y \leftarrow y*2 \\
 & W_2(y) \\
 & C_2 \\
 & \downarrow
\end{array}
$$

$$
T_1 = \{ R_2(x) \prec W_2(x),\ R_2(y) \prec W_2(y) \}
$$

$$
H = \{ \underbrace{R_1(x) \prec W_1(x)}_{wl_1(x)} \prec \underbrace{R_2(x) \prec W_2(x)}_{wl_2(x)} \prec \underbrace{R_2(y) \prec W_2(y)}_{wl_2(y)} \prec \underbrace{R_1(y) \prec W_1(y)}_{wl_1(y)} \}
$$

Locking-Based Mechanism - Example

T_1: $R_1(x)$
$x \leftarrow x+1$
$W_1(x)$
$R_1(y)$
$y \leftarrow y\text{-}1$
$W_1(y)$
C_1
$\downarrow$

$$T_1 = \{ R_1(x) \prec W_1(x), \; R_1(y) \prec W_1(y) \}$$

T_2: $R_2(x)$
$x \leftarrow x*2$
$W_2(x)$
$R_2(y)$
$y \leftarrow y*2$
$W_2(y)$
C_2
$\downarrow$

$$T_1 = \{ R_2(x) \prec W_2(x), \; R_2(y) \prec W_2(y) \}$$

Expected results (starting with x=50, y = 20)

- x=102 and y = 38 (T_1 before T_2)
- x=101 and y = 39 (T_2 before T_1)

Actual results

- x=102 and y = 39 (T_1 and T_2 interleave)

$$H = \{ \underbrace{R_1(x) \prec W_1(x)}_{wl_1(x)} \prec \underbrace{R_2(x) \prec W_2(x)}_{wl_2(x)} \prec \underbrace{R_2(y) \prec W_2(y)}_{wl_2(y)} \prec \underbrace{R_1(y) \prec W_1(y)}_{wl_1(y)} \}$$

Locking-Based Mechanism - Example

T_1: $R_1(x)$
$x \leftarrow x+1$
$W_1(x)$
$R_1(y)$
$y \leftarrow y\text{-}1$
$W_1(y)$
C_1

↓

$$T_1 = \{ R_1(x) \prec W_1(x), \; R_1(y) \prec W_1(y) \}$$

T_2: $R_2(x)$
$x \leftarrow x*2$
$W_2(x)$
$R_2(y)$
$y \leftarrow y*2$
$W_2(y)$
C_2

↓

$$T_1 = \{ R_2(x) \prec W_2(x), \; R_2(y) \prec W_2(y) \}$$

Expected results (starting with x=50, y = 20)

- x=102 and y = 38 (T_1 before T_2)
- x=101 and y = 39 (T_2 before T_1)

Actual results

- x=102 and y = 39 (T_1 and T_2 interleave)

(simple) locking grants exclusive access to data item…

… but it does not grant isolation (T_1 and T_2 interleave)

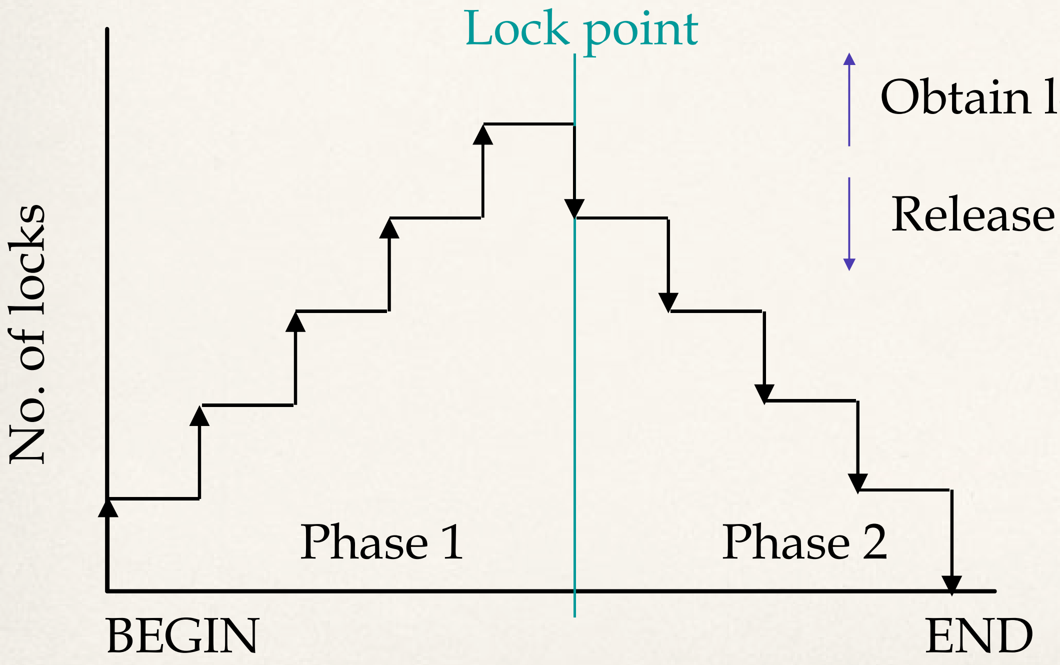
$$H = \{ \underbrace{R_1(x) \prec W_1(x)}_{wl_1(x)} \prec \underbrace{R_2(x) \prec W_2(x)}_{wl_2(x)} \prec \underbrace{R_2(y) \prec W_2(y)}_{wl_2(y)} \prec \underbrace{R_1(y) \prec W_1(y)}_{wl_1(y)} \}$$

Two-Phase Locking (2PL)

❶ A Transaction locks an object before using it

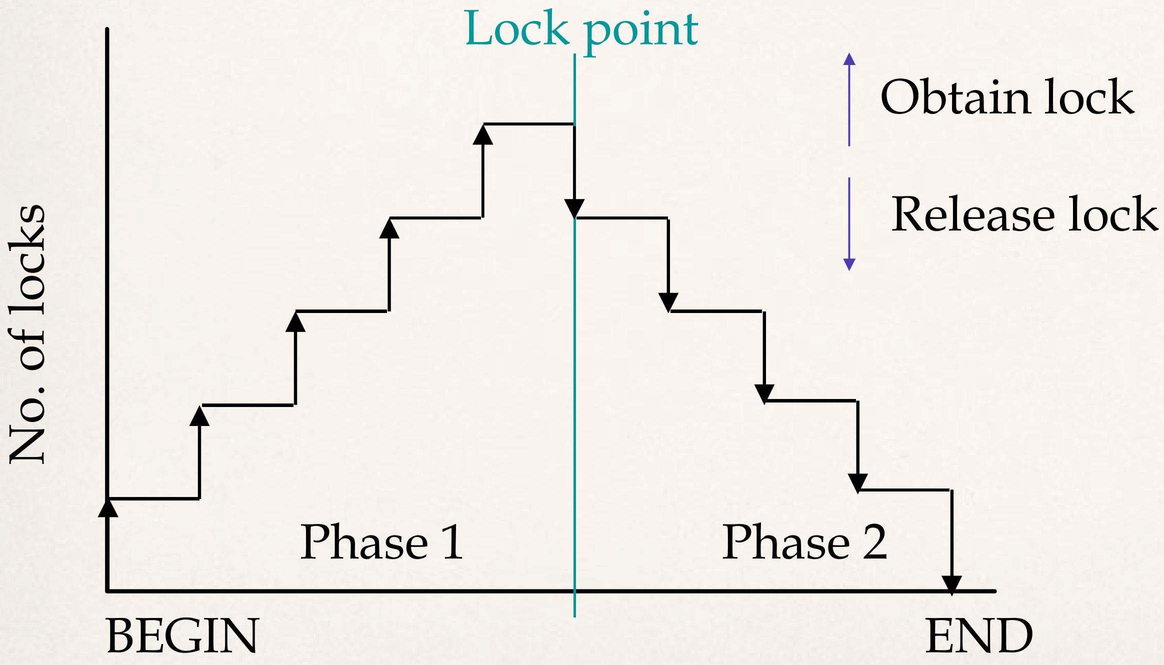
❷ When an object is locked by another transaction, the requesting transaction must wait (if conflicting)

❸ When a transaction releases a lock, it may not request another lock.



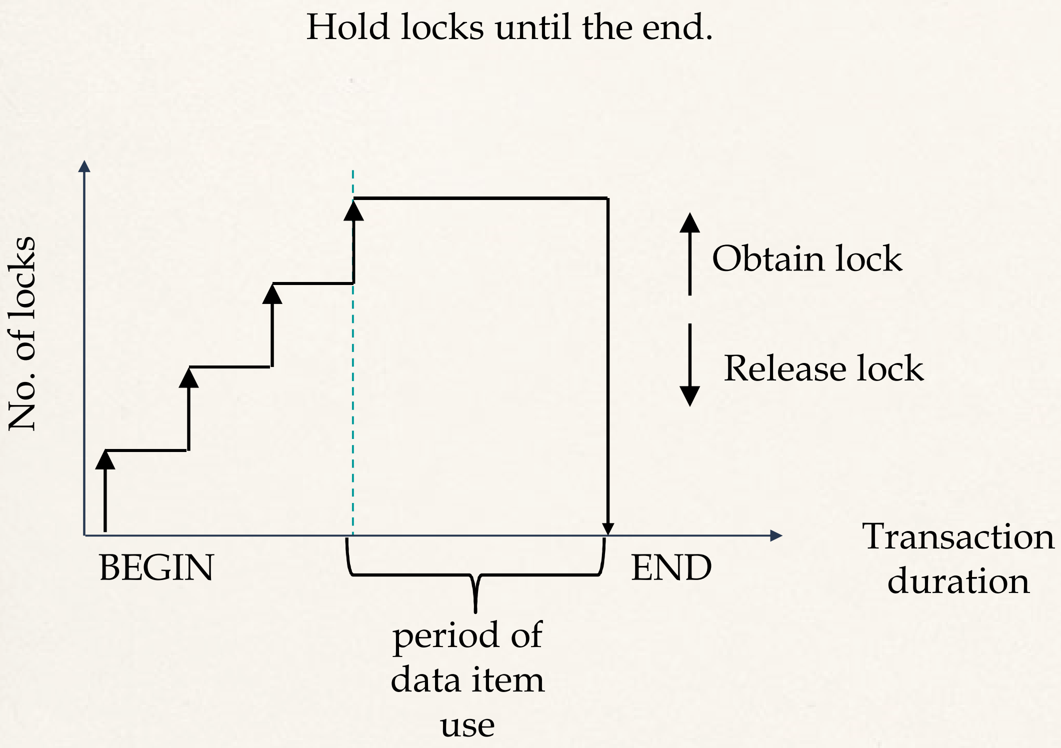
Two-Phase Locking (2PL)

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



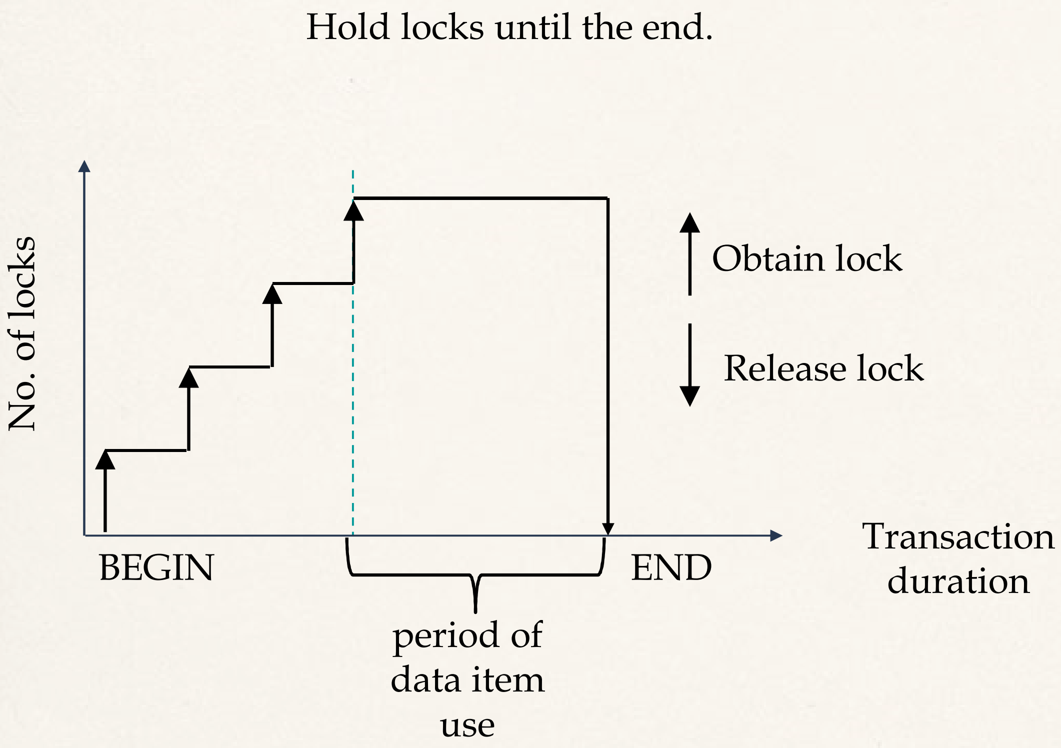
- 2PL guarantees serializable histories
- Implementation issues
  - TM must know not only when data item *x* will not be used anymore…
  - … also when no more locks will be requested
  - Moreover, during descending phase other transactions get lock (dirty read)
    - Possibility of cascade aborts

Strict 2PL



- Strict 2PL: all locks are released together after commit
- Higher degree of isolation
- Easier to implement
- Less concurrency

Strict 2PL



- Strict 2PL: all locks are released together after commit
- Higher degree of isolation
- Easier to implement
- Less concurrency

Locking-based mechanisms can cause deadlocks

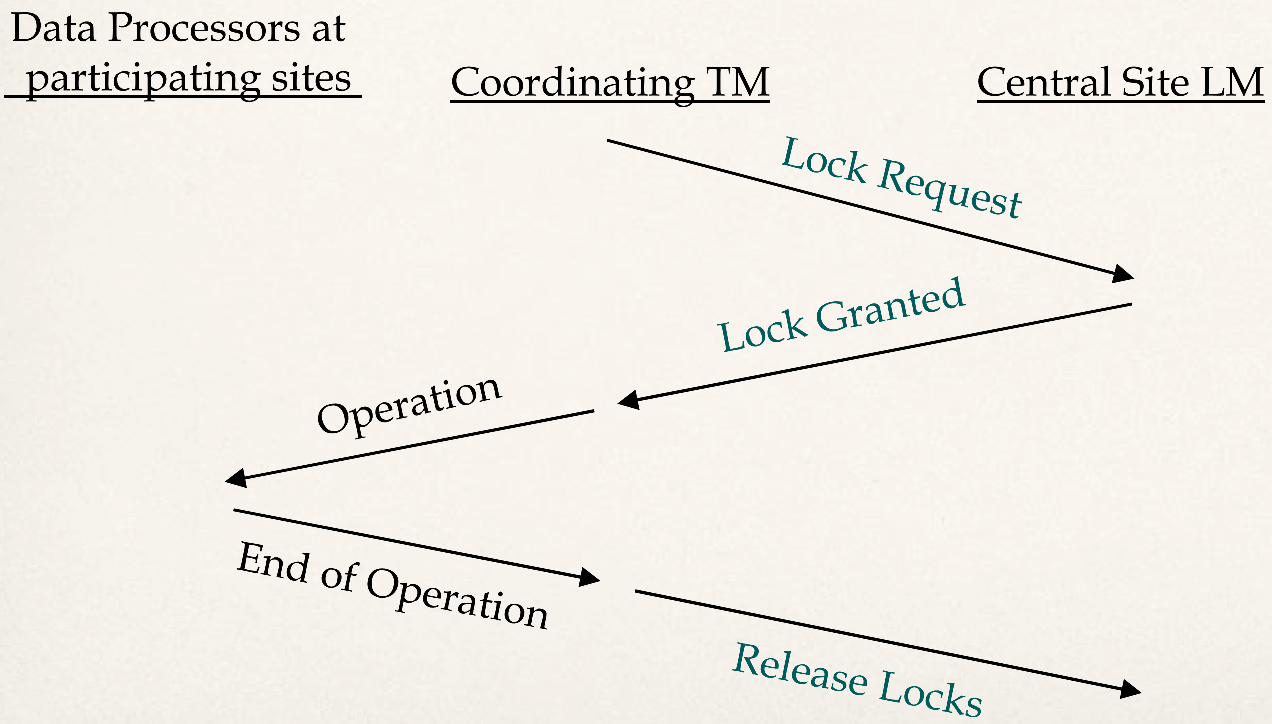
2PL: Implementation Alternatives

Two possible implementation for (strict) 2PL in the distributed context

- centralized: 1 SC
- distributed: many SC

Centralized 2PL

- There is only one 2PL scheduler in the distributed system
- Lock requests are issued to the central scheduler
- Coordinating TM is where the query is initiated



Data Processors at participating sites

Coordinating TM

Central Site LM

Lock Request

Lock Granted

Operation

End of Operation

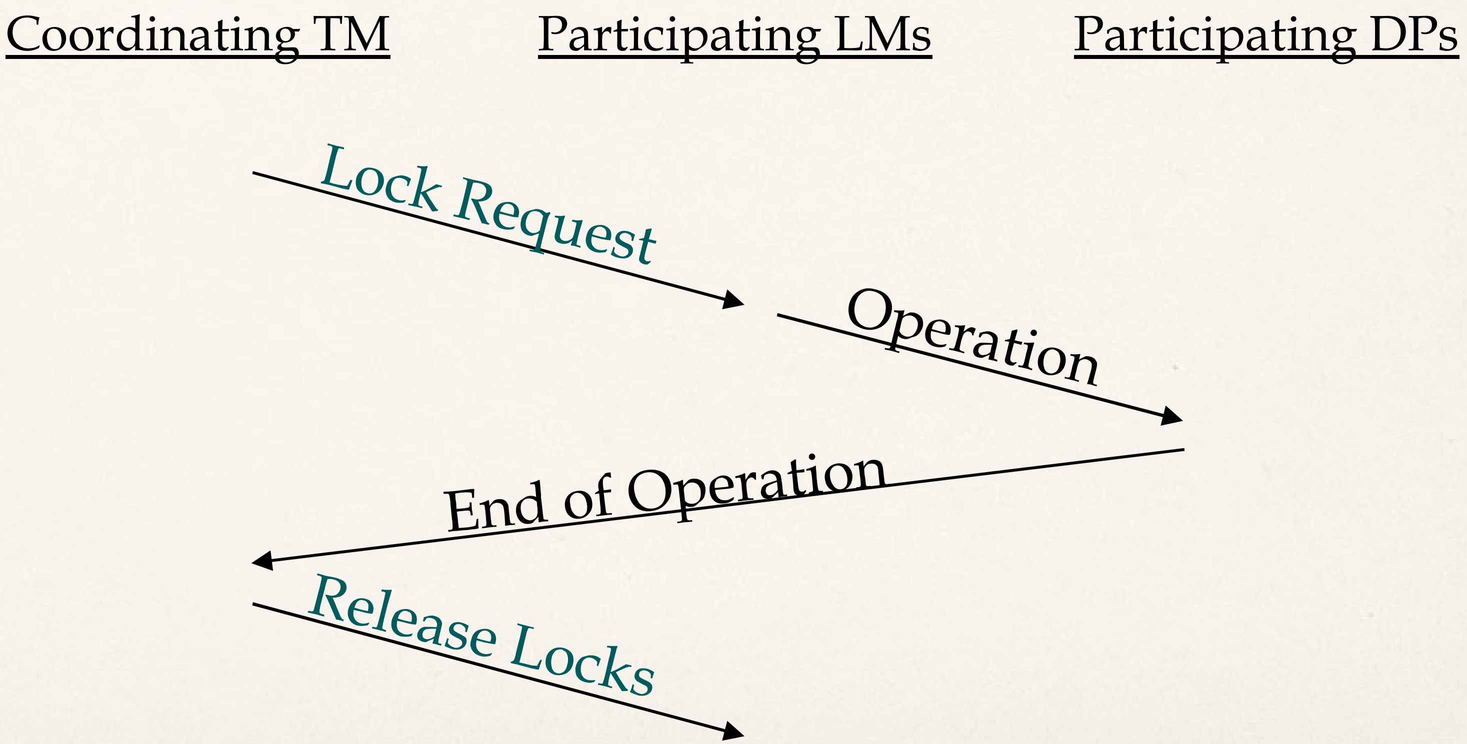
Release Locks

Issues with centralized 2PL

- Bottleneck at central LM for high workload at central LM
- Low reliability in case of failure of central LM

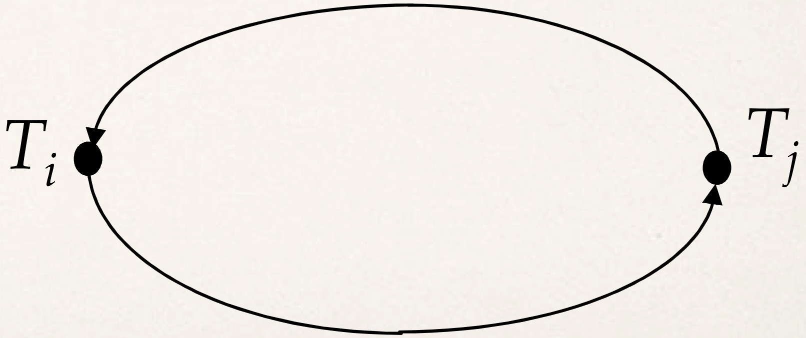
Distributed 2PL

- 2PL schedulers (LM's) are placed at each site
  - each scheduler handles lock requests for data at that site



Deadlock

- T_1 has write lock on x
  T_2 has write lock on y
  T_1 asks for write access to y
  T_2 asks for write access to x
- Deadlock!!!
- Deadlock are modeled through wait-for graph (WFG)
  - 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
  - In distributed context the WFG is distributed across nodes
    - LWFG: a local graph at each site
    - GWFG: union of all LWFG



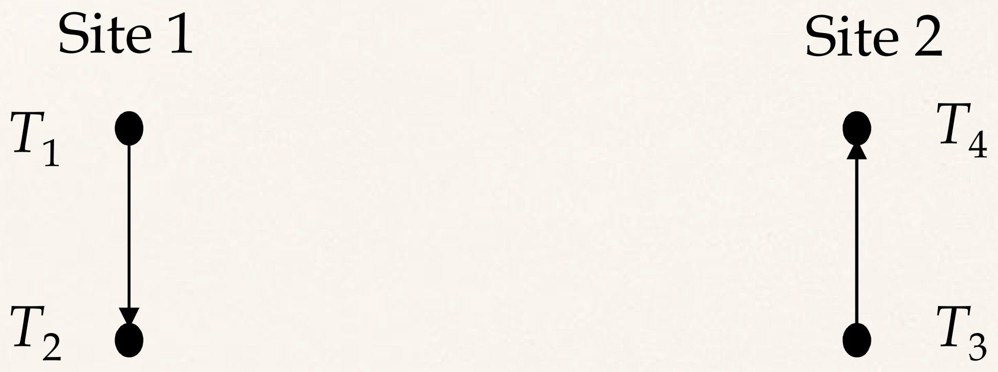
Deadlock Management

- Prevention (deadlock is avoided before transaction is started)
  - Guaranteeing that deadlocks can never occur in the first place
    - do not allow for risky transactions
    - e.g., a transaction starts if none of data is going to use is locked
- Avoidance (deadlock is avoided when a locked resource is requested)
  - Detecting potential deadlocks in advance and taking action to insure that deadlock will not occur. Requires run time support
    - Timestamps to prioritize transactions
    - Ordered resources
- Detection and Recovery
  - Allowing deadlocks to form and then finding and breaking them. As in the avoidance scheme, this requires run time support.

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

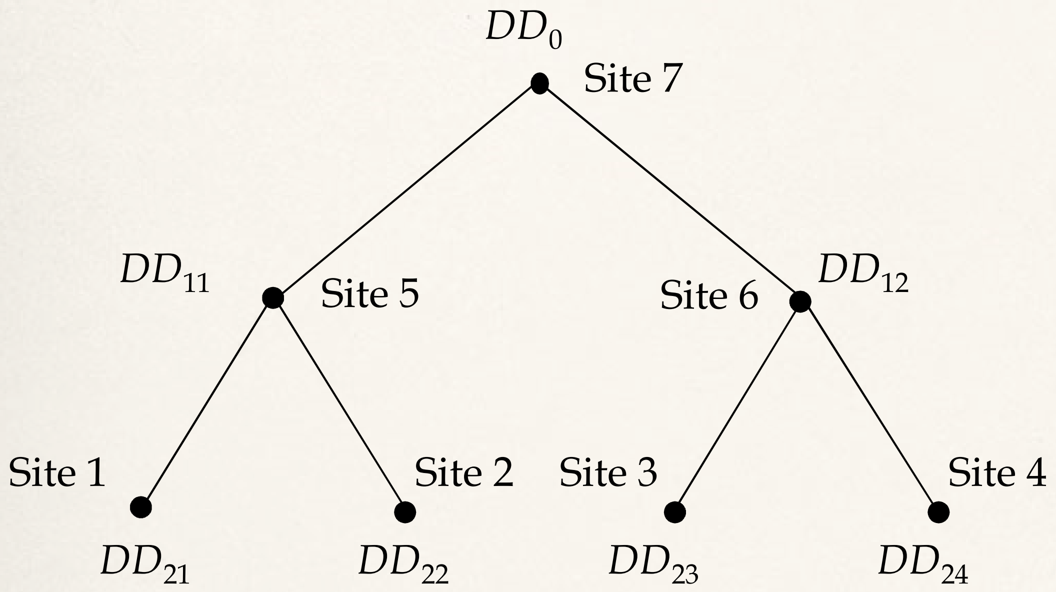
Deadlock Detection

- 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 (centralized 2PL)

Hierarchical Deadlock Detection



- Each site has a DD
- DD are arranged in a hierarchy (e.g., tree shaped)
- Each DD search for cycle in its and lower-level LWFG
- Each DD sends its LWFG to upper levels
- PRO: less dependence from central DD
  - → less communication costs
- CONTRO: implementation issues

Distributed Deadlock Detection

- Each site has a DD that maintain an LWFG
- Sites cooperate in detection of global 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