6. Distributed Transaction Management

Chapter 10

Introduction to Transaction Management

Definition of Transaction

- A transaction is a sequence of read and write operations on a database with some special properties (e.g., ACID, BASE, …).

- An SQL statement is a transaction.
- An embedded SQL statement is a transaction.
- A program enclosed by “Begin-transaction” and “end” is a transaction.

An Airline Database Example

FLIGHT(FNO, DATE, SRC, DEST, STSOLD, CAP)

CUST(CNAME, ADDR, BAL)

FC(FNO, DATE, CNAME, SPECIAL)

Example Transaction – SQL Version

The reservation program

```
Begin-transaction
begin
    input(flight_no, date, customer_name);
    EXEC SQL
        UPDATE FLIGHT
        SET STSOLD = STSOLD + 1
        WHERE FNO = FLIGHT_NO
        AND DATE = date;
    EXEC SQL
        INSERT  INTO FC(FNO, DATE, CNAME, SPECIAL)
        VALUES(flight_no, date, customer_name, null);
    output("Reservation completed");
end.
```

Termination Conditions of Transactions

- A transaction may be terminated by command of
  - Commit (successfully completed), or
  - Rollback (aborted)
- Commit makes DB operations effect permanent and the result is visible to other transactions.
- Rollback undoes all DB operations and restore the DB to the state before the execution of the transaction.

Termination of Transaction Example

```
Begin_transaction Reservation
 begin {Reservation}
   input(flight_no, date, customer_name);
   EXEC SQL SELECT  STSOLD,CAP
                    INTO      temp1,temp2
                    FROM     FLIGHT
                    WHERE  FNO = flight_no AND DATE = date;
   if (temp1 == temp2)   then   { output("no free seats");     Abort  }
   else {  EXEC SQL   UPDATE   FLIGHT
                      SET          STSOLD = STSOLD + 1
                      WHERE     FNO = flight_no AND DATE = date;
           EXEC SQL   INSERT
                      INTO  FC(FNO, DATE, CNAME, SPECIAL);
                      VALUES (flight_no, date, customer_name, null);
           Commit
           output("reservation completed")
        }
   endif
end {Reservation}
```

Example Transaction – Reads & Writes

```
Begin_transaction Reservation
 begin {Reservation}
   input(flight_no, date, customer_name);
   temp ← Read(flight(date).stsold);
   if (temp == flight(date).cap)  then
   begin
       output("no free seats");
       Abort;
   end
   else begin
          Write(flight(date).stsold, temp + 1);
          Write(flight(date).cname, customer_name);
          Write(flight(date).special, null);
          Commit;
          output("reservation completed")
        end
end. {Reservation}
```

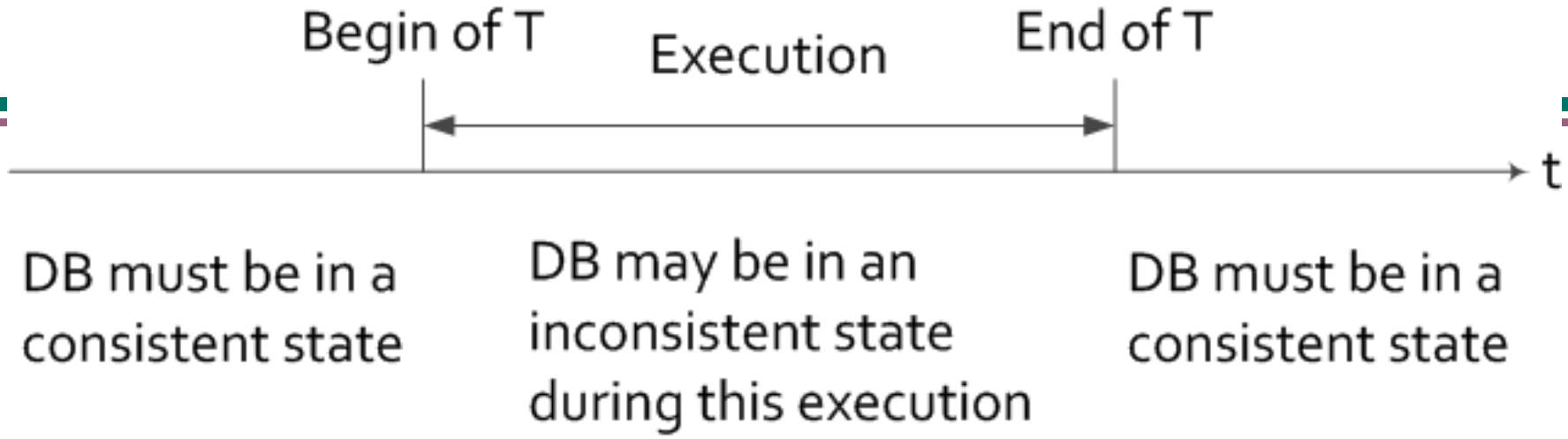
What is a Transaction?

A transaction is a sequence of database operations organized in a basic unit for keeping database consistent and reliable.

Consistency of Database

A database is in a consistent state if it follows:

- entity integrity
- referential integrity
- domain value constraints, etc.



- A temporary inconsistent state of a transaction should not be exposed to other transactions.
- A database should be in a consistent state even if there are a number of concurrent transactions accessing the database.

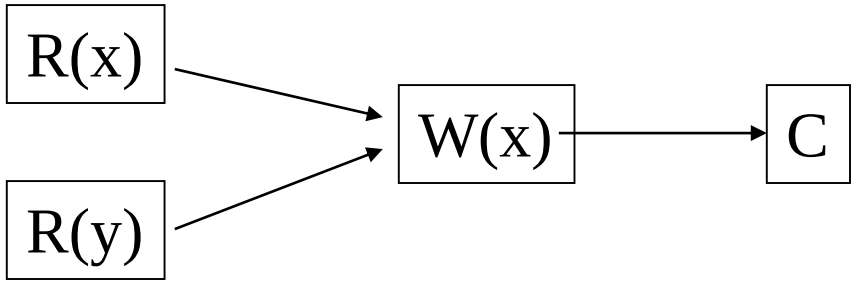
Reliability of Database

A database is reliable if it is resilient and capable of recovering.

Transaction Example

T:

Read(x)
Read(y)
x ← x + y
Write(x)
Commit



Σ = { R(x), R(y), W(x), C }

< = { (R(x), W(x)), (R(y), W(x)) , (W(x), C), (R(x), C), (R(y), C)}

Characterization of Transactions

- Read Set (RS)
  - the set of data items that are read by a transaction
- Write Set (WS)
  - the set of data items whose values are changed by this transaction
- Base Set (B)
  - RS $\cup$ WS

Formalization

Let

- $O_{ij}(x)$ be some operation O_j of transaction T_i operating on entity x, where $O_j \in$ {*read,write*} and O_j is atomic
- $OS_i = \cup_j O_{ij}$
- $N_i \in$ {abort, commit}

Transaction T_i is a partial order $T_i = \{ \Sigma_i , <_i \}$, where

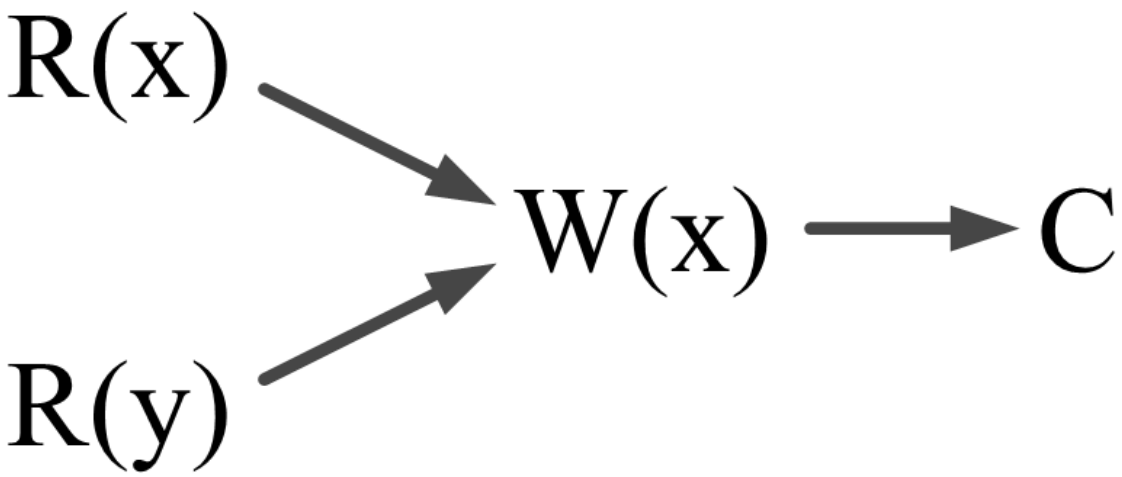
- $\Sigma_i = OS_i \cup \{N_i\}$
- $\forall O_{ij} \in OS_i , O_{ij} <_i N_i$
- For any two operations $O_{ij}, O_{ik} \in OS_i$, if ($O_{ij} = R(x)$) and ($O_{ik} = W(x)$) for any data item x, then either $O_{ij} <_i O_{ik}$ or $O_{ik} <_i O_{ij}$

Conflict Operations

$<_i$ is an ordering relation for database operations of T_i . Two operations are in conflict if one of them is a **write**.

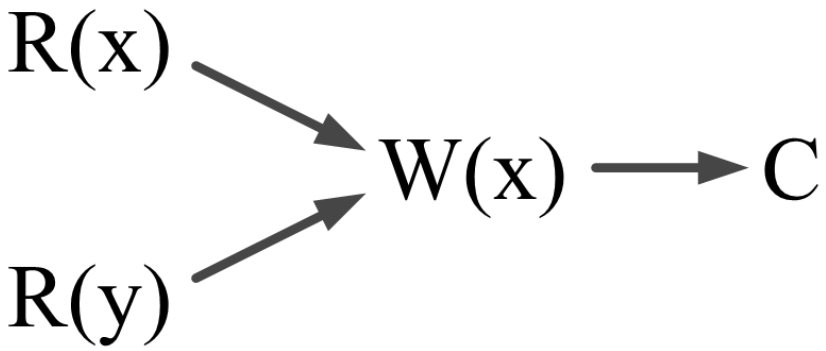
Directed Acyclic Graph

- A partial order can be represented by a DAG (Directed Acyclic Graph) whose vertices are the operations and edges are ordering.



Directed Acyclic Graph (*cont*.)

No order exists between R(x) and R(y)



A transaction can be simplified by using its relative order of operations, e.g., the above T can be written as:

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

Properties of Transactions -- ACID

ATOMICITY

all or nothing

CONSISTENCY

no violation of integrity constraints

ISOLATION

concurrent changes invisible and serializable

DURABILITY

committed updates persist

ACID (ATOMICITY, CONSISTENCY, ISOLATION, DURABILITY)

```
Begin_transaction Reservation
 begin {Reservation}
   input(flight_no, date, customer_name);
   EXEC SQL SELECT  STSOLD,CAP
                    INTO      temp1,temp2
                    FROM     FLIGHT
                    WHERE  FNO = flight_no AND DATE = date;
   if (temp1 == temp2)   then   { output("no free seats");     Abort  }
   else { EXEC SQL   UPDATE   FLIGHT
                     SET          STSOLD = STSOLD + 1
                     WHERE     FNO = flight_no AND DATE = date;
          EXEC SQL   INSERT
                     INTO  FC(FNO, DATE, CNAME, SPECIAL);
                     VALUES (flight_no, date, customer_name, null);
          Commit
          output("reservation completed")
        }
   endif
end {Reservation}
```

Atomicity

- Either all or none of the transaction's operations are performed.
- Atomicity requires that if a transaction is interrupted by a failure, its partial results must be undone.
- The activity of preserving the transaction's atomicity in presence of transaction aborts due to input errors, system overloads, or deadlocks is called transaction recovery.
- The activity of ensuring atomicity in the presence of system crashes is called crash recovery.

Consistency

- Internal consistency
  - A transaction which executes alone against a consistent database leaves it in a consistent state.
  - Transactions do not violate database integrity constraints.
- The consistency of a transaction is simply its correctness
- A transaction is a correct program that maps one consistent state database state to another .
- The property to be guaranteed by concurrency control.

Consistency (*cont*.)

- Four consistency degrees can be defined on the basis of dirty data concept.
- Dirty data - data values that have been updated by a transaction prior to its commitment.

Conditions \ Consistency degree	3	2	1	0
A transaction T does not overwrite dirty data of other transactions.				√
+ A transaction T does not commit any writes until it completes all writes, i.e. until the end of T.			√	
+ A transaction T does not read dirty data from other transactions.		√		
+ Other transactions do not dirty any data read by T before T completes.	√			

Isolation

- Serializability
  - If several transactions are executed concurrently, the results must be the same as if they were executed serially in some order.
- Incomplete results
  - An incomplete transaction cannot reveal its results to other transactions before its commitment.
  - Necessary to avoid lost update and cascading aborts.

Isolation Example

- Consider the following two transactions:

T_1: **Read(x)**
x ←x+1
Write(x)
Commit

T_2: **Read(x)**
x ←x+1
Write(x)
Commit

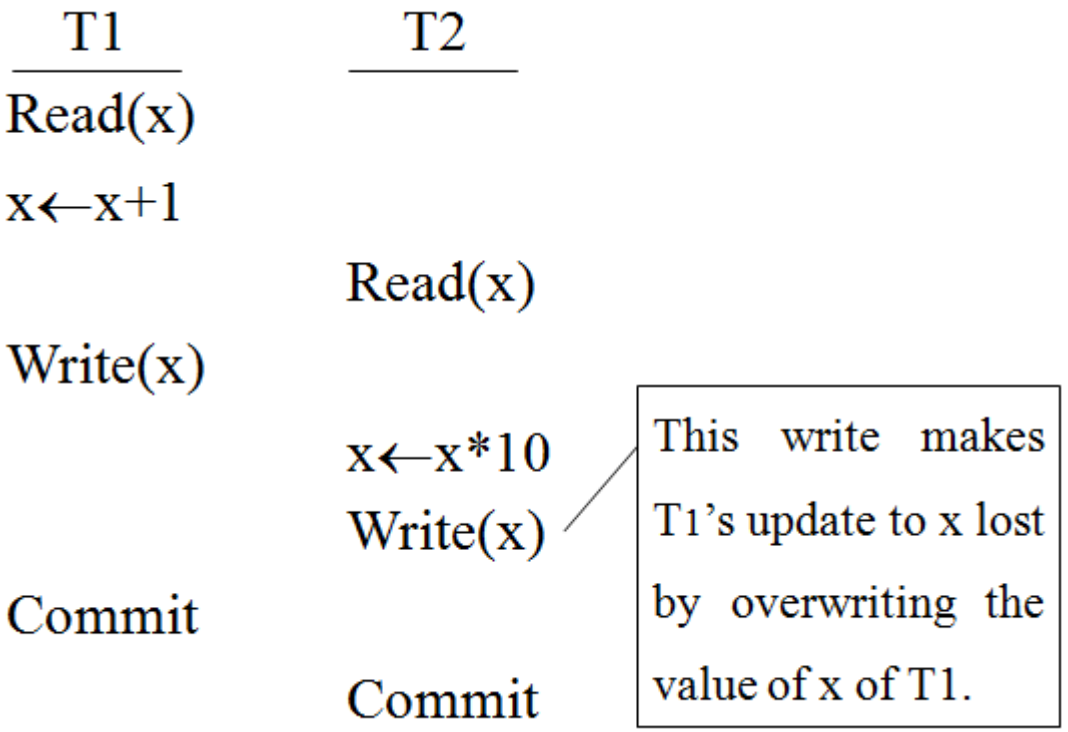
- Possible execution sequences:

T_1 : **Read(x)**
T_1 : **x ← x+1**
T_1 : **Write(x)**
T_1 : **Commit**
T_2 : **Read(x)**
T_2 : **x ← x+1**
T_2 : **Write(x)**
T_2 : **Commit**

T_1 : **Read(x)**
T_1 : **x ← x+1**
T_2 : **Read(x)**
T_1 : **Write(x)**
T_2 : **x ← x+1**
T_2 : **Write(x)**
T_1 : **Commit**
T_2 : **Commit**

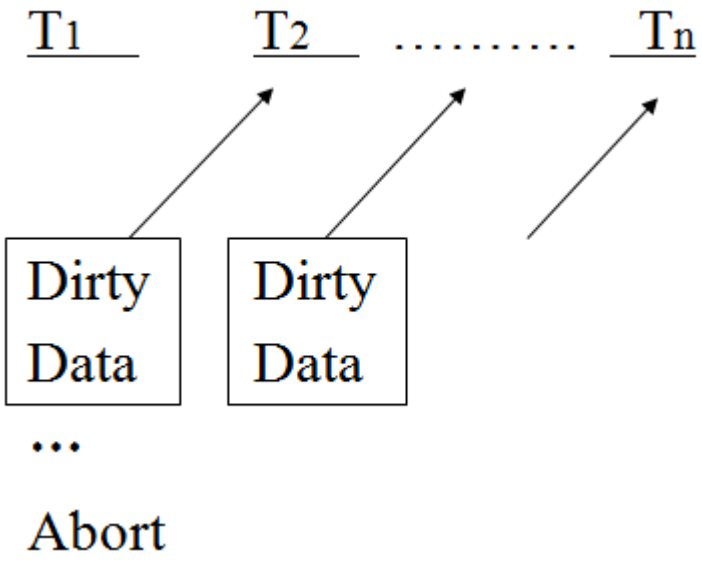
Reasons for Requiring Isolation

Lost update



Reasons for Requiring Isolation (*cont.*)

Cascading abort



T_i reveals its data to T_{i+1} before commit, when T_1 aborts, all other transactions must abort.

SQL92 Isolation Levels

Phenomena that can occur if proper isolation is not maintained

1) Dirty read

T_1 modifies x which is then read by T_2 before T_1 terminates; T_1 aborts $\Rightarrow$ T_2 has read value which never exists in the database.

2) Non repeatable (fuzzy) read

T_1 reads x; T_2 then modifies or deletes x and commits. T_1 tries to read x again but might read a different value or can't find it.

3) Phantom

T_1 searches the database according to a predicate while T_2 inserts new tuples that satisfy the predicate.

Consistency Degree vs. Isolation

Conditions \ Consistency degree	3	2	1	0
A transaction T does not overwrite dirty data of other transactions.				√
+ A transaction T does not commit any writes until it completes all writes, i.e. until the end of T.			√	
+ A transaction T does not read dirty data from other transactions.		√		
+ Other transactions do not dirty any data read by T before T completes.	√			

- Consistency degree level 2 avoids cascading aborts
- Consistency degree level 3 provides complete isolation

Durability

- Once a transaction commits, the system must guarantee that the results of its operations will never be lost in spite of subsequent failures.

- Durability demands recovery functions of a DBMS.

Types of Transactions

- Based on
  - Application areas
    - Non-distributed vs. Distributed
    - Compensating transactions, if the purpose is to undo the effect of previous transactions
    - Heterogeneous transactions, if running in a heterogeneous DBMS
  - Timing
    - On-line (short-life) vs. batch (long-life)

Types of Transactions (*cont.*)

- Organization of read and write actions
  - Two-step (e.g., all read actions before any write)
  - Restricted (e.g., for a data item, read-before-write)
  - Action model (e.g., restricted, each <read, write> pair executed atomically)
  - Write prior read

- Structure
  - Flat (or simple) transactions
  - Nested transactions
  - Workflows

Transaction Structure - Flat

- Flat transaction consists of a sequence of primitive operations embraced between a begin and end markers.

```
Begin_transaction  Reservation
  begin
  ...
  end.
```

Transaction Structure - Nested

- The operations of a nested transaction may themselves be transactions.

```
Begin_transaction Reservation
  begin {Reservation}
        Begin_transaction Airline
        {  begin
           ...
           end. {Airline}
        Begin_transaction Hotel
        {  begin
           ...
           end. {Hotel}
end. {Reservation}
```

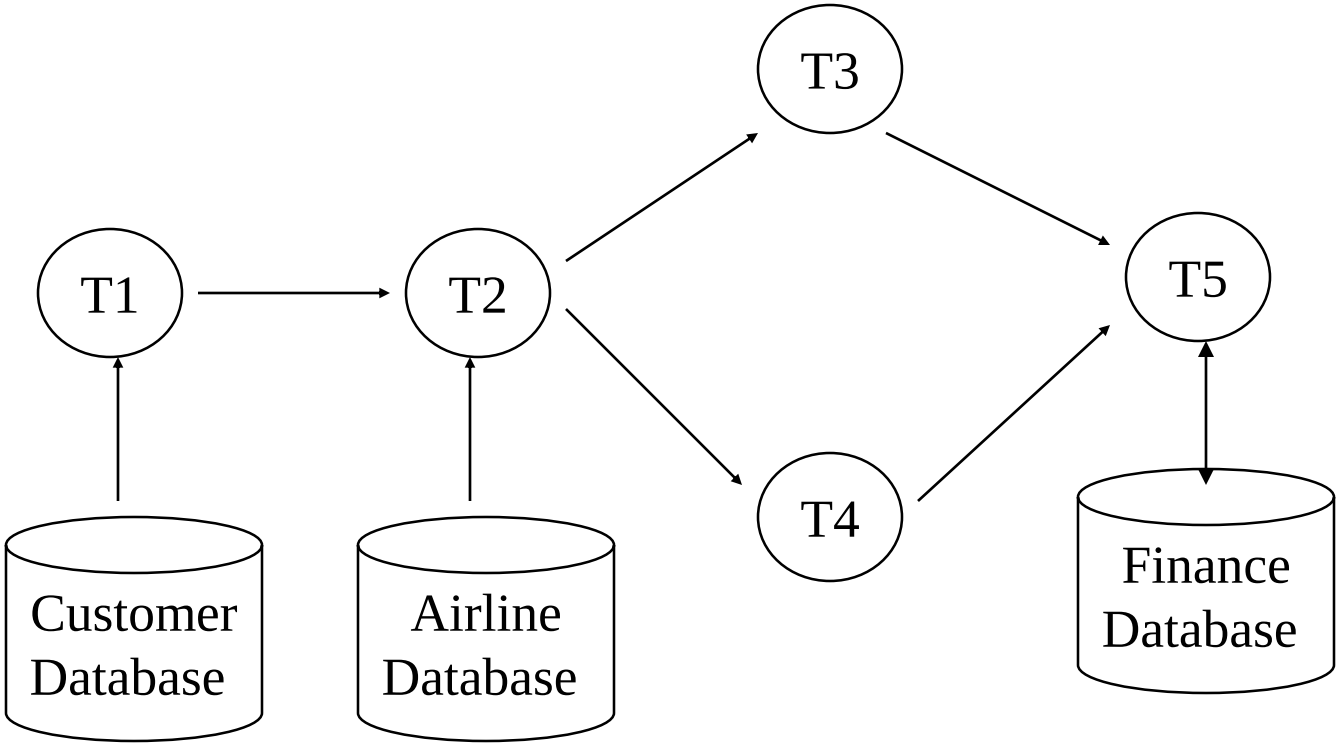
Nested Transactions

- Have the same properties as their parents → may themselves have other nested transactions.
- Introduce concurrency control and recovery concepts to within the transaction.
- Types
  - Closed nesting
    - Sub-transactions begin after their parents and finish before them.
    - Commitment of a sub-transaction is conditional upon the commitment of the parent (commitment through the root).
  - Open nesting
    - Sub-transactions can execute and commit independently.
    - Compensation may be necessary.

Workflows

- A collection of tasks organized to accomplish some business process.
- Three Types
  - **Human-oriented workflows**
    - Involve humans in performing the tasks
    - System support for collaboration and coordination; but no system-wide consistency definition
  - **Transactional workflows**
    - May involve humans, require access to heterogeneous, autonomous and/or distributed systems, and support selective use of ACID properties
  - **System-oriented workflows**
    - Computation-intensive & specialized tasks that can be executed by a computer
    - System support for concurrency control and recovery, automatic task execution, notification, etc.

Workflow Example



T1: Customer request obtained
T2: Airline reservation performed
T3: Hotel reservation performed
T4: Auto reservation performed
T5: Bill generated

Transactions Provide ...

- *Atomic* and *reliable* execution in the presence of failures
- *Correct* execution in the presence of multiple user accesses
- Correct management of *replicas* (if they support it)

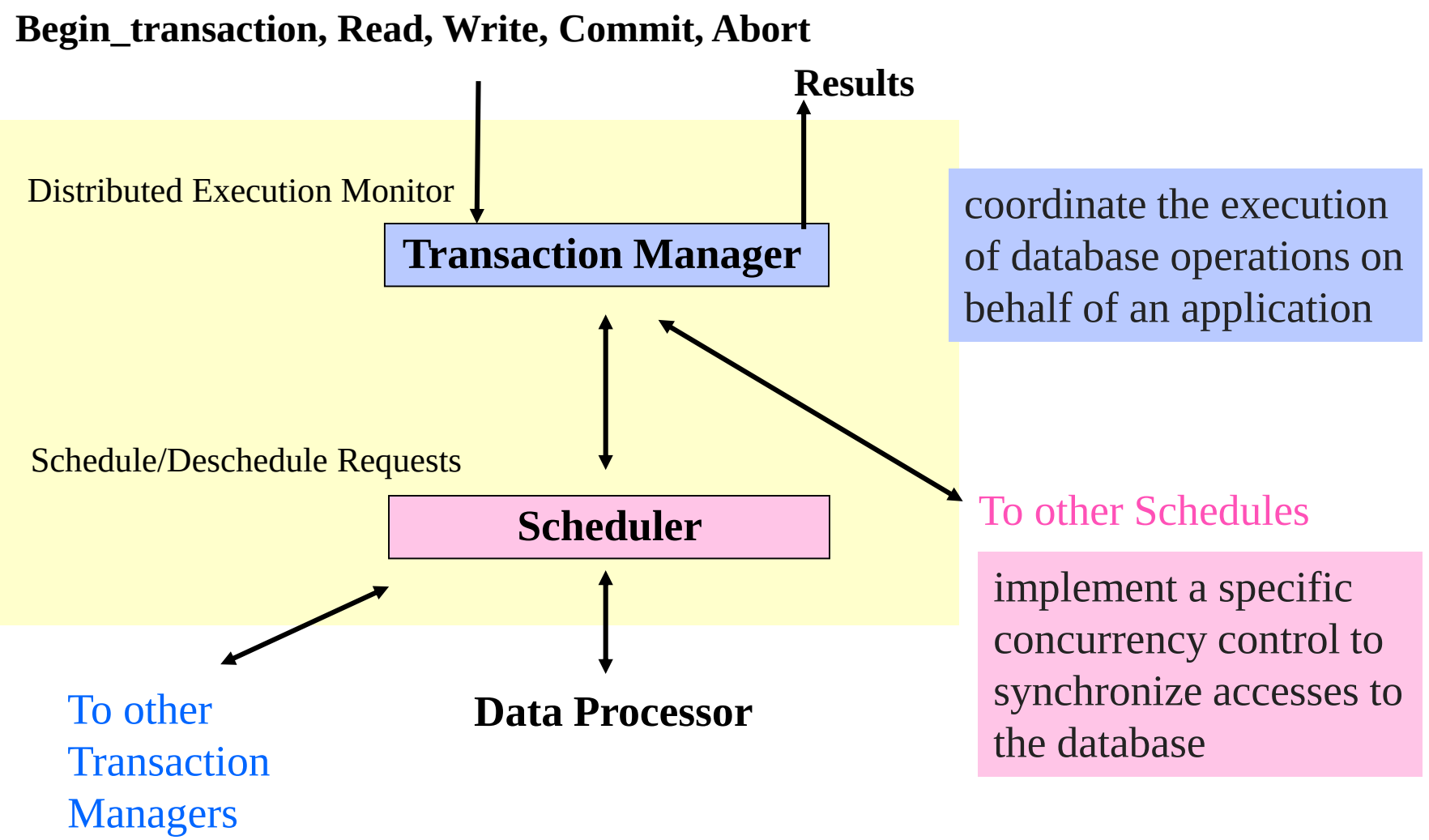
Transaction Processing Issues

- Transaction structure (usually called transaction model)
  - Flat (simple), nested
- Internal database consistency
  - Semantic data control (integrity enforcement) algorithms
- Reliability protocols
  - Atomicity & Durability
  - Local recovery protocols
  - Global commit protocols

Transaction Processing Issues (*cont*.)

- Concurrency control algorithms
  - How to synchronize concurrent transaction executions (correctness criterion)?
  - Consistency and Isolation
- Replica control protocols
  - How to control the mutual consistency of replicated data?
  - One copy equivalence and ROWA (Read One Write All)

Architecture Revisited

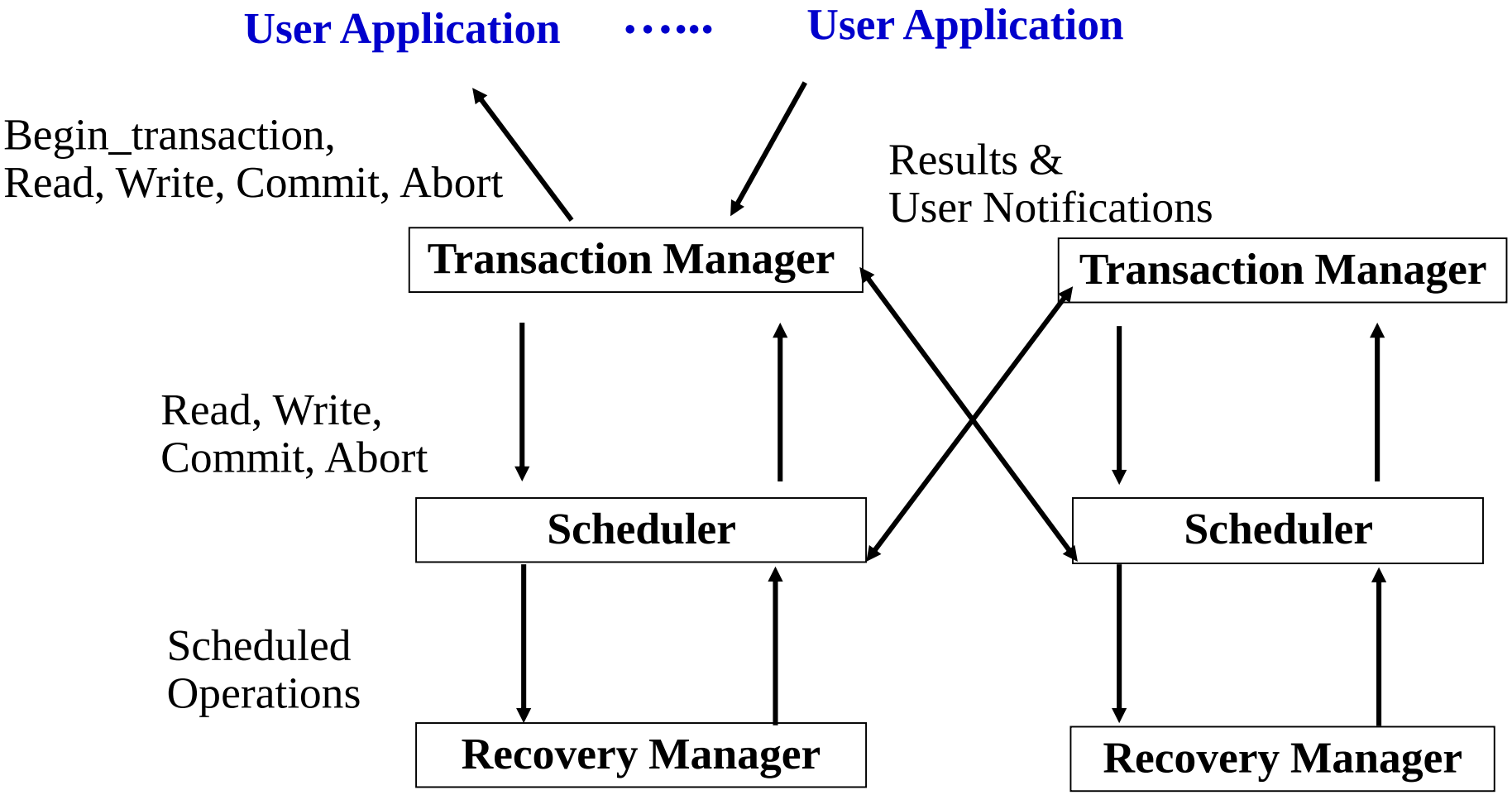


7. Distributed Concurrency Control

Chapter 11

Distributed Concurrency Control

Centralized Transaction Execution



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 Schedule

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

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

$S_1 = \{ W_2(x), R_1(x), R_3(x), W_1(x), C_1 , W_2(y), R_3(y), R_2(z), C_2 , R_3(z), C_3 \}$

Formalization of Schedule

- A complete schedule $SC(T)$ over a set of transactions $T = \{T_1, ..., T_n\}$ is a partial order $SC(T) = \{ \Sigma_T , <_T \}$, where

  1) $\Sigma_T = \cup_i \Sigma_i$, for i = 1, 2, ..., n

  2) $<_T \supseteq \cup_i <_i$, for i = 1, 2, ..., n

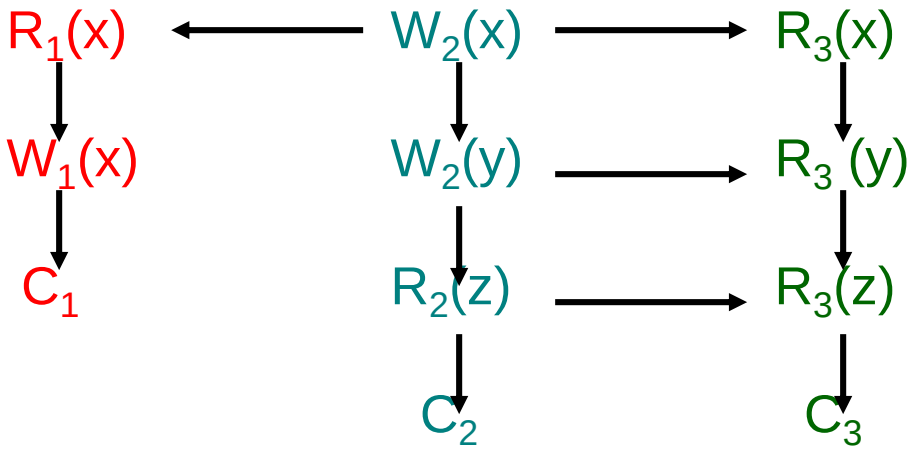
  3) For any two conflicting operations $O_{ij}, O_{kl} \in \Sigma_T$, either $O_{ij} <_T O_{kl}$ or $O_{kl} <_T O_{ij}$

Complete Schedule -- Example

- 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

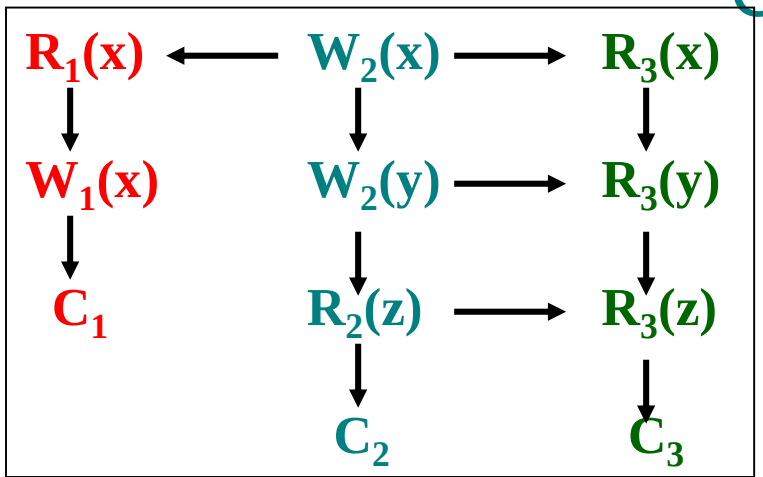
A possible complete schedule is given as the DAG

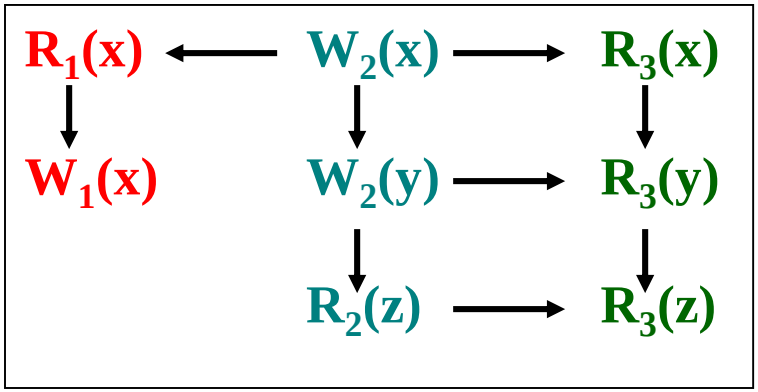


Schedule Definition

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





Serial Schedule

- 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.

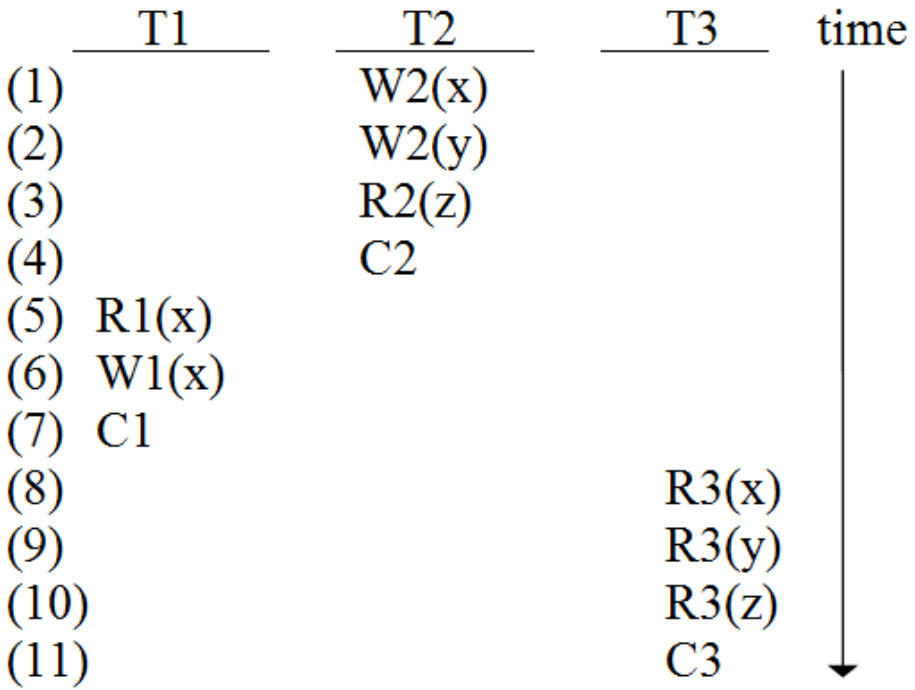
T_1 :	Read(x)	T_2 :	Write(x)	T_3 :	Read(x)
	Write(x)		Write(y)		Read(y)
	Commit		Read(z)		Read(z)
			Commit		Commit

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

Serializable Schedule Example

- A serial schedule for the three transactions

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



Serializable Schedule

- Transactions execute concurrently, but the net effect of the resulting schedule upon the database is equivalent to some serial schedule.
- Equivalent with respect to what?
  - 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.
  - Conflict equivalence: the relative order of execution of the conflicting operations belonging to unaborted transactions in two schedules are the same.

Serializable Schedule Example (*cont.*)

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

Is the following conflict equivalent?

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

$S_1 = \{W_2(x), R_1(x), R_3(x), W_1(x), C_1, \underline{R_3(y)}, \underline{W_2(y)}, R_2(z), C_2, R_3(z), C_3\}$

No!

Serializable Schedule Example (*cont*.)

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

Is the following conflict equivalent?

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

$S_2 = \{W_2(x), R_1(x), W_1(x), C_1, R_3(x), W_2(y), R_3(y), R_2(z), C_2, R_3(z), C_3\}$

Yes, so S2 is serializable.

Serializable Schedule Example (*cont.*)

	T1	T2	T3
(1)		W2(x)	
(2)	R1(x)		
(3)	W1(x)		
(4)			R3(x)
(5)		W2(y)	
(6)			R3(y)
(7)		R2(z)	
(8)		C2	
(9)	C1		
(10)			R3(z)
(11)			C3

time ↓

	T1	T2	T3
(1)		W2(x)	
(2)		W2(y)	
(3)		R2(z)	
(4)		C2	
(5)	R1(x)		
(6)	W1(x)		
(7)	C1		
(8)			R3(x)
(9)			R3(y)
(10)			R3(z)
(11)			C3

time ↓

Serializable Schedule

Concurrency Control Algorithms

- Pessimistic
  - Two Phase Locking based (2PL)
    - Centralized (primary site) 2PL
    - Primary copy 2PL
    - Distributed 2PL
  - Timestamp Ordering (TO)
    - Basic TO
    - Conservative TO
    - Multiversion 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/shared lock (*rl*) or write lock/exclusive lock (*wl*)
- Read locks and write locks conflict (because Read and Write operations are incompatible)

	rl	***wl***
rl	yes	no
wl	no	no

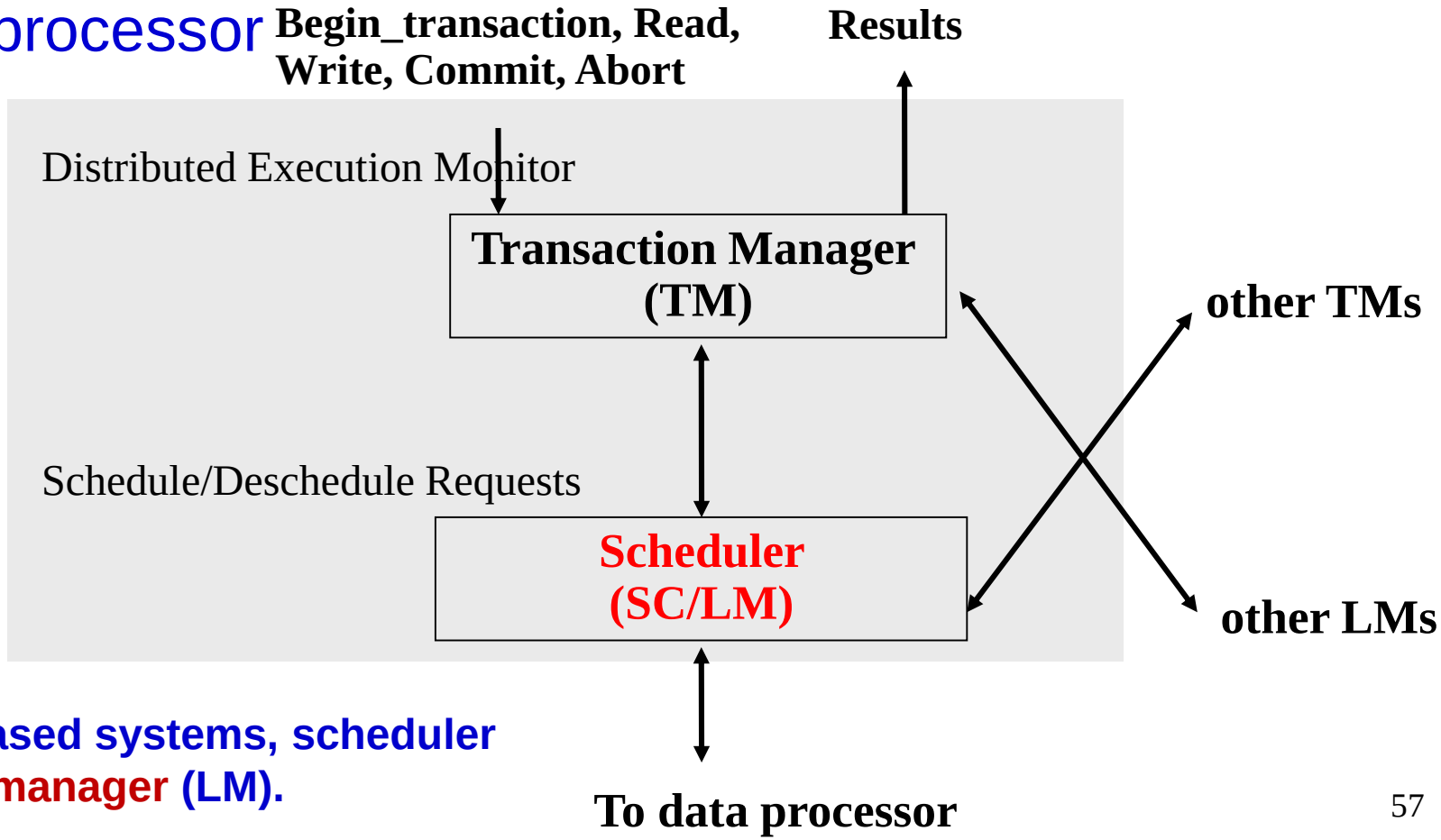
Locking-Based Strategy

- A transaction locks an object before using it.
- When an object is locked by another transaction, the requesting transaction (with an incompatible lock mode) must wait.

☞ Locking works nicely to allow concurrent processing of transactions.

Basic Lock Manager Algorithm

1. LM receives DB operations and associated information (transaction ID etc.) from the TM, or a message from the data processor



In locking-based systems, scheduler (SC) is lock manager (LM).

Basic LM Algorithm (*cont*.)

2. For a message from the TM, LM checks if the *lu* which contains the required data item is locked:
   - If locked, and the locking mode is incompatible, then put the current operation on queue;
   - Otherwise
     I. the lock is set in an appropriate mode, and
     II. database operation is passed onto the data processor, and then
     III. wait for messages from the data processor.

Locking granularity, i.e. the size of the portion of database that a lock is applied. This portion is called locking unit (lu).

Basic LM Algorithm (*cont*.)

3. Upon the receipt of a message from the data processor:

   I. Release the lock on *lu* held by the transaction;

   II. Check, if there are no more locks on *lu,* and there are operations waiting in queue to lock *lu*, then

   SOP ← first operation from the queue;

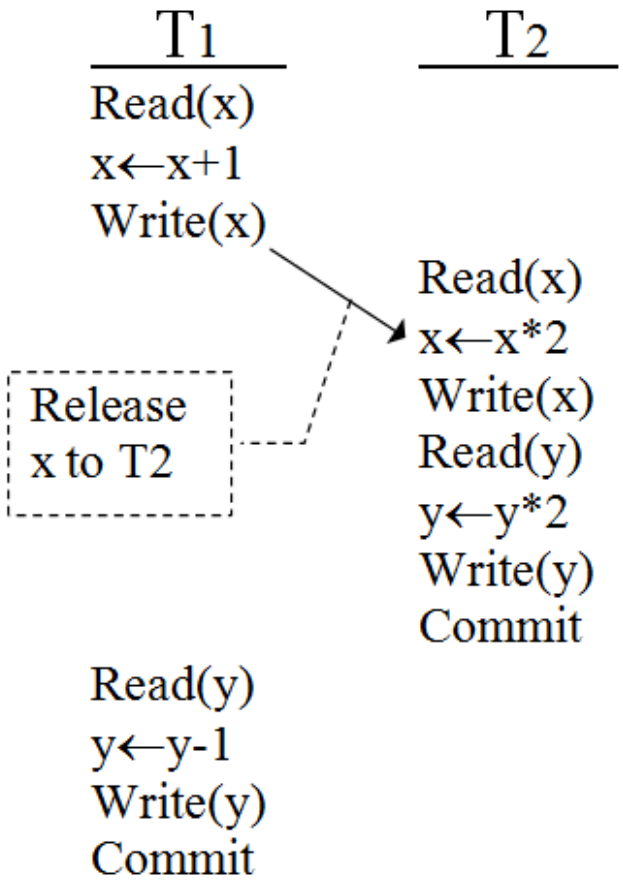
   SOP ← *SOP* ∪ {*O* | *O* is an operation that can lock *lu* in a compatible mode with *the current operations in SOP*};

   set the locks on *lu* for these operations in *SOP*;

   send the operations to corresponding data processors.

Problem with Basic LM Algorithm ?

Problem with Basic LM - Example



Initial: x = 50 y = 20

Result of above scheduling:

x = 102 y = 39

Result of schedule {T1,T2}:

x = 102 y = 38

Result of schedule {T2,T1}:

x = 101 y = 39

Basic LM algorithm may generate non-serializable schedule!

Problem with Basic LM Algorithm

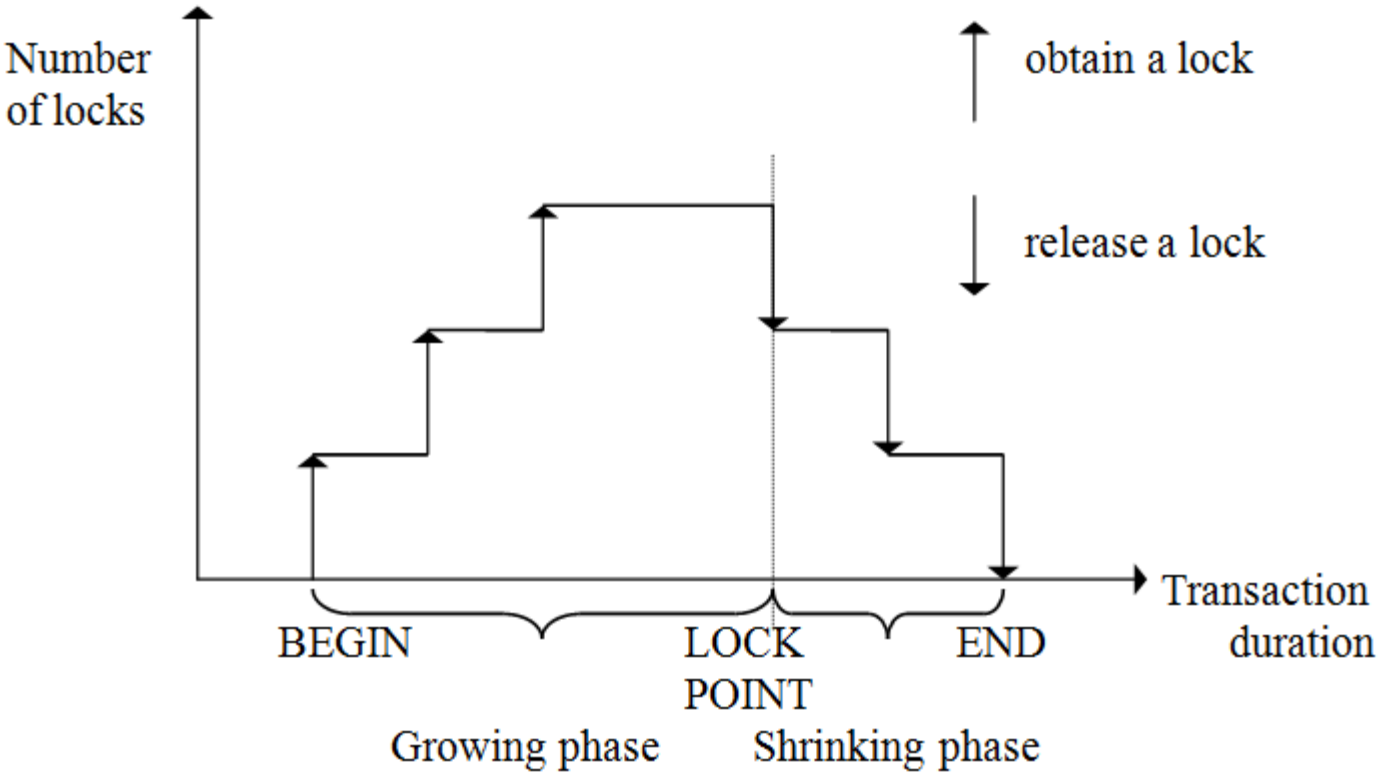
Early release of data

Solution ?

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.

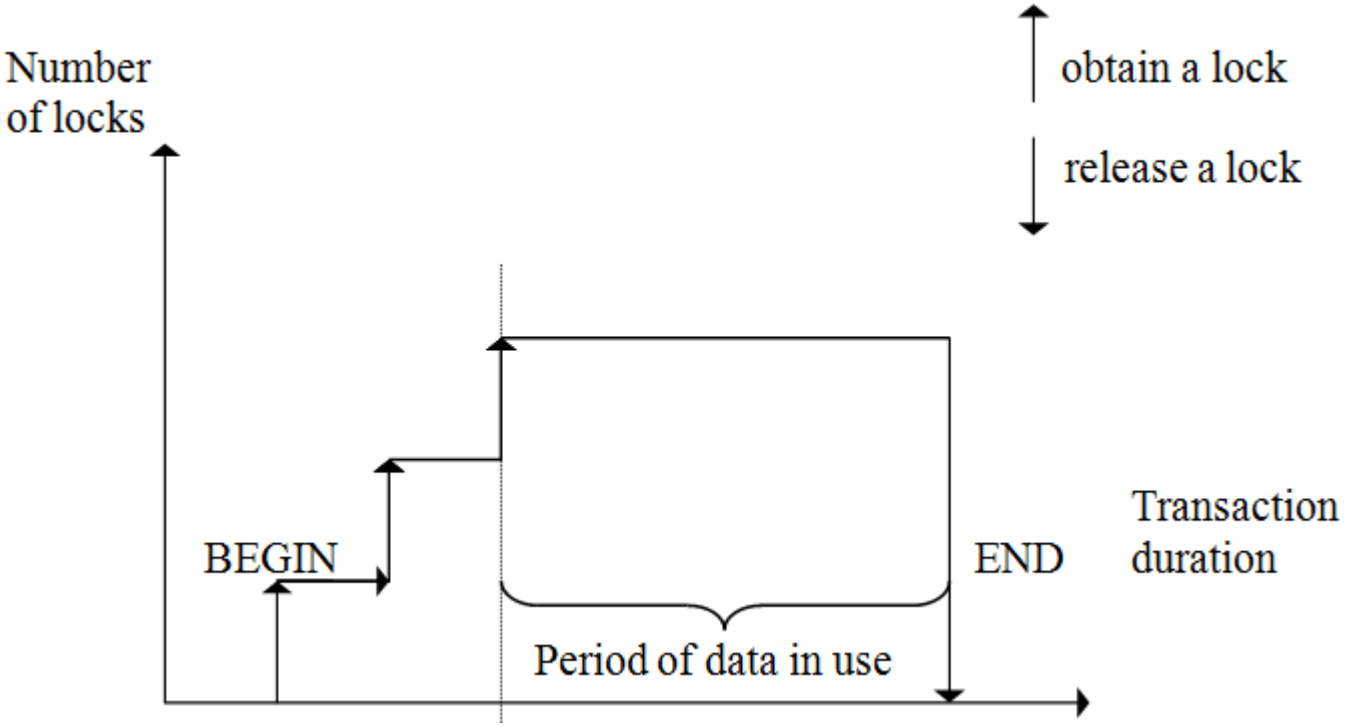
Two-Phase Locking (2PL)



Note: (1) 2PL is serializable; (2) Lock point is hard to determine;
(3) 2PL can cause cascading aborts!!!

Strict 2PL (S2PL)

Hold locks until the end of transaction.



- S2PL is serializable.
- S2PL requires minimal modification to 2PL algorithm.

Centralized S2PL

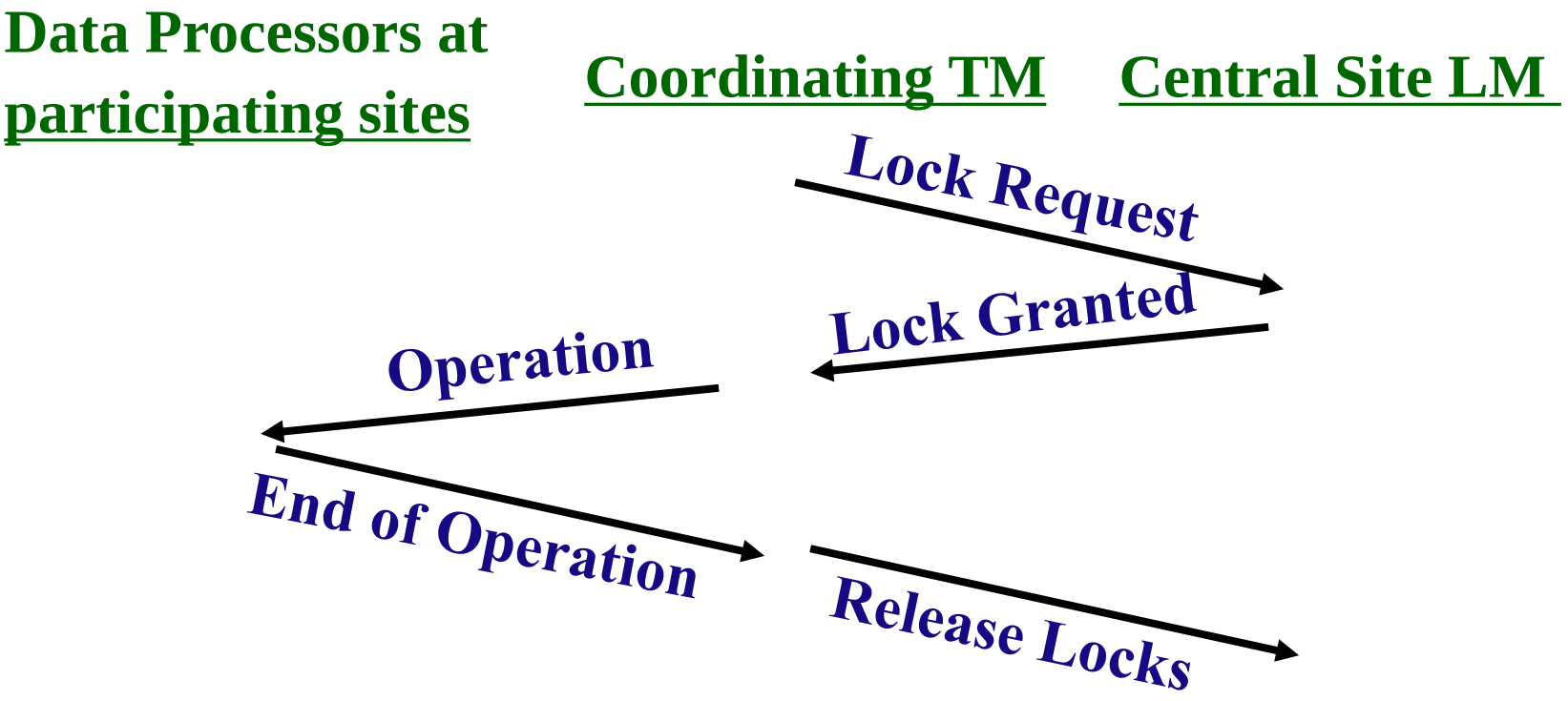
- A method to delegate lock management responsibility to a single site only (known as primary S2PL algorithm)
  - Coordinating TM

    The TM at the site where the transaction is initiated.
  - Participating sites

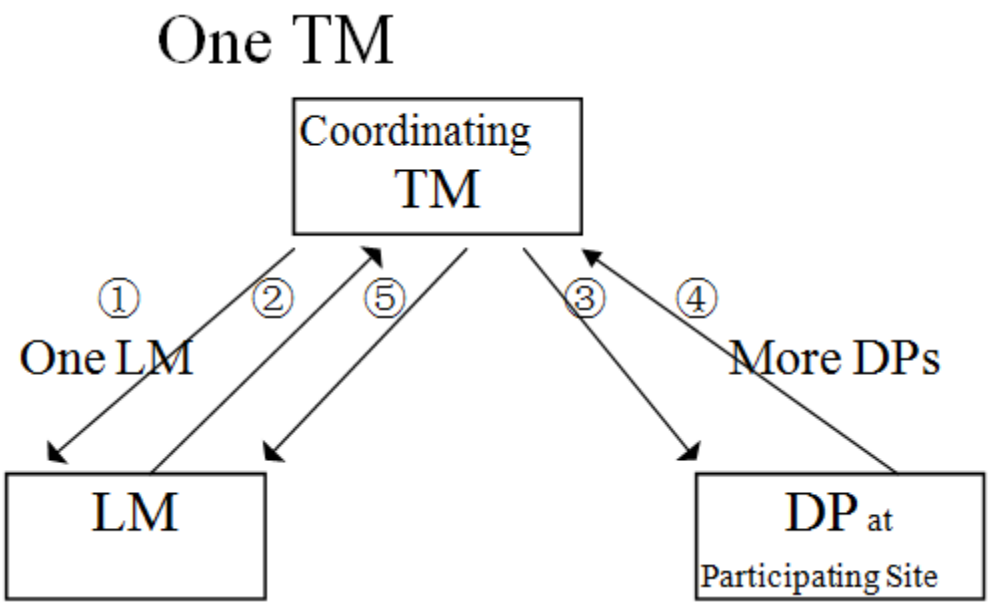
    Those at which database operations are to be carried out.

One Central Lock Manager

- There is only one S2PL scheduler in the distributed system.
- Lock requests are issued to the central scheduler.



Communication Structure of Centralized S2PL



Messages:

① **Lock request**

② **Lock granted**

③ **Database operation**

④ **End of database operation**

⑤ **Release locks**

Problems - poor performance and reliability

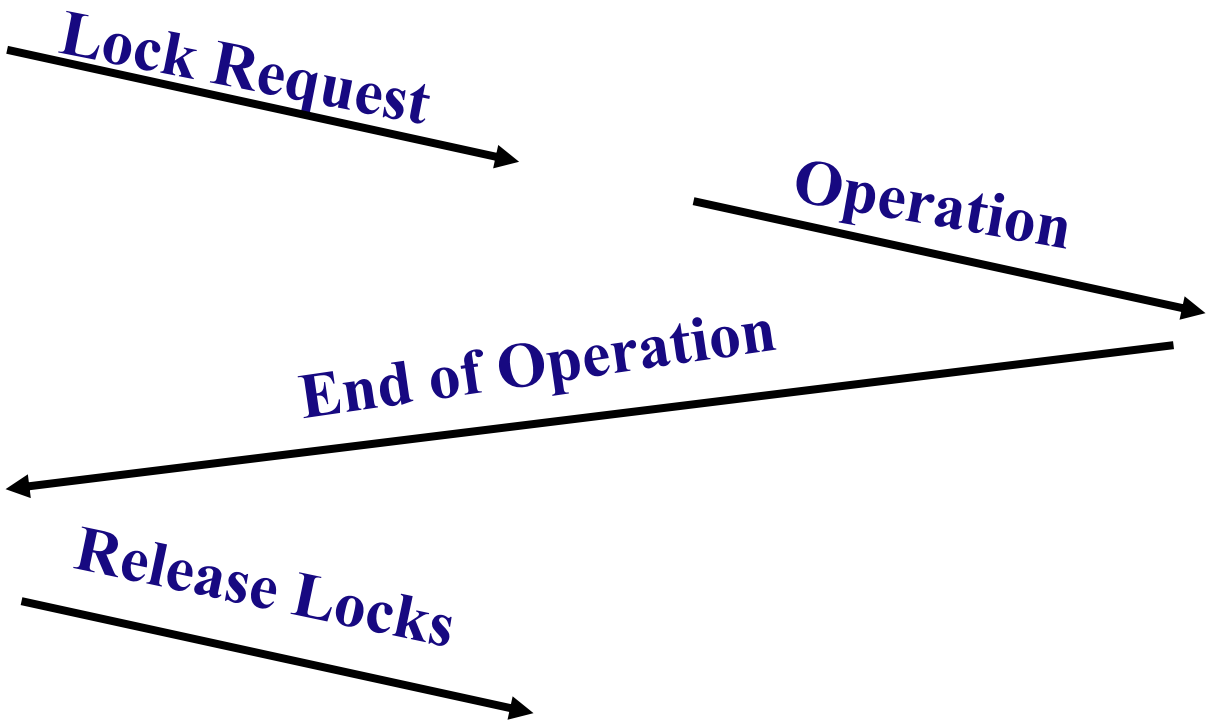
Primary Copy S2PL

- Implement lock management at a number of sites and make each lock manager responsible for a set of lock units.
- A dictionary is needed to keep the allocation of lock managers information.
- Changes from centralized S2PL to primary copy S2PL are minimal.

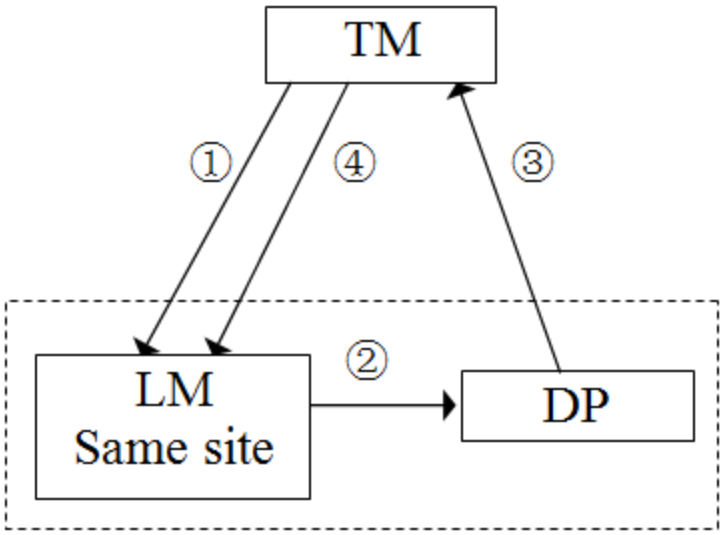
Distributed S2PL

- There is a lock manager at every site.
- Each lock manager handles lock requests for data at that site.
- Concurrency control is accomplished by the cooperation of lock managers at the sites where data are involved in the set of transactions.

Distributed S2PL Execution



Communication Structure of Distributed S2PL



Messages:

① **Lock request**

② **Lock granted**

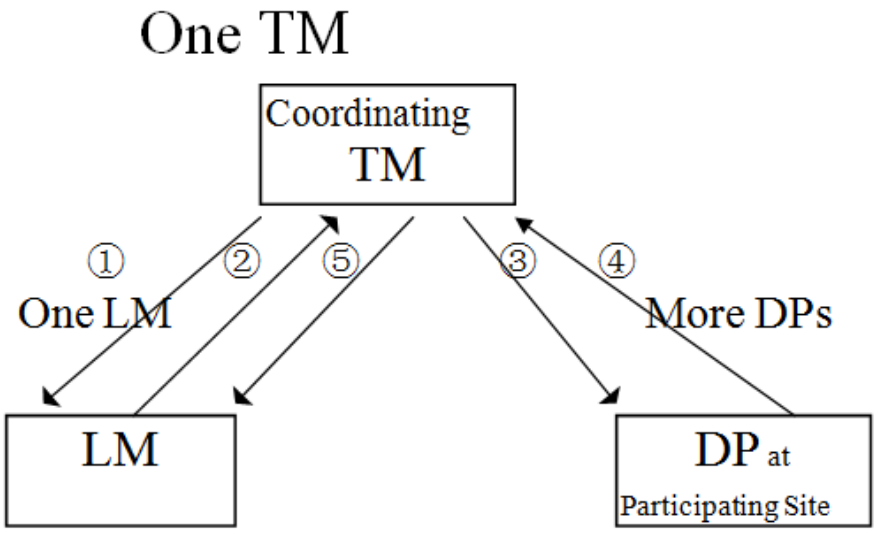
③ **End of database operation**

④ **Release Lock**

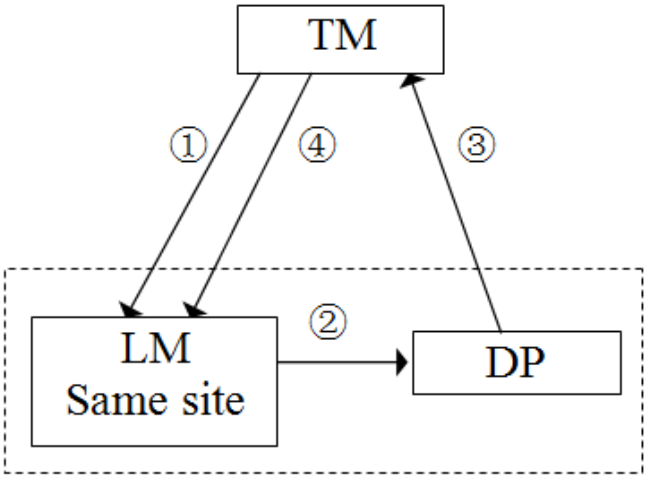
- Differences between centralized 2PL and distributed 2PL can be observed by looking at their communication structures.

Differences between Centralized and Distributed S2PL

Centralized S2PL



Distributed S2PL



① **Lock request**
② **Lock granted**
③ **Database operation**
④ **End of database operation**
⑤ **Release locks**

① **Lock request**
② **Lock granted**
③ **End of database operation**
④ **Release Lock**

Distributed S2PL Execution

- 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

  -- Read One Write All protocol

Summary of S2PL Locking

- **Centralized:** One site does all locking
  - Vulnerable to single site failure
- **Primary Copy:** All locking for an object done at the primary copy site for this object
  - Reading requires access to locking site as well as site where the object is stored
- **Distributed:** Locking for a copy done at site where the copy is stored
  - Locks at all sites while writing an object (Read One Write All)

Timestamp-based Concurrency Control Algorithms

- Lock method maintains serializability by mutual exclusion.

- Timestamp method maintains serializability by assigning a unique timestamp to every transaction and executing transactions in order.

Timestamp of Transaction

- Transaction manager assigns a globally unique timestamp $ts(T_i)$ to transaction T_i
- Transaction manager attaches the timestamp to all operations issued by the transaction.
- Timestamp can be used to permit ordering
- Monotonicity -- timestamps generated by the same TM monotonically increase in value.

How to Assign a Timestamp Value?

- Use a monotonically increased counter, but this is difficult in a distributed environment.
- Use a two-tuple form:

  `<local-counter-value, site-identifier>`

- Use another two-tuple form:

  `<local-system-clock, site-identifier>`

site ID is put in the least significant position

to avoid the situation that timestamps from a site will always be larger/smaller than timestamps from another site.

Timestamp Ordering (TO) Rule

- Given two conflicting operations $O_{ij}(x)$ and $O_{kl}(x)$ belonging to transaction T_i and T_k respectively,

  $O_{ij}(x)$ is executed before $O_{kl}(x)$
  if and only if $ts(T_i) < ts(T_k)$

i.e., the older transaction gets executed first.

Schedule by the TO Rule

Basic procedure:

1. Check each new operation against conflicting operations that have been scheduled;
2. IF the new operation belongs to a younger (later) transaction than all conflicting ones THEN

   accept it;

   ELSE

   reject it;

   restart the entire transaction with a new timestamp.

This means the system maintains the execution order according to the timestamp order.

Basic TO Algorithm

- Conflicting operations are resolved by timestamp order.
- To facilitate easy checking whether some transaction with a larger timestamp has already accessed a data item, each data item is assigned a write timestamp (*wts*) and a read timestamp (*rts*):
  - *rts*(*x*) = largest (youngest) timestamp of any read on *x*
  - *wts*(*x*) = largest (youngest) timestamp of any write on *x*

Basic TO Algorithm (*cont*.)

for *Ri*(*x*):

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

then reject *Ri*(*x*)

else accept *Ri*(*x*)

rts(*x*) ←*ts*(*Ti*)

for *Wi*(*x*):

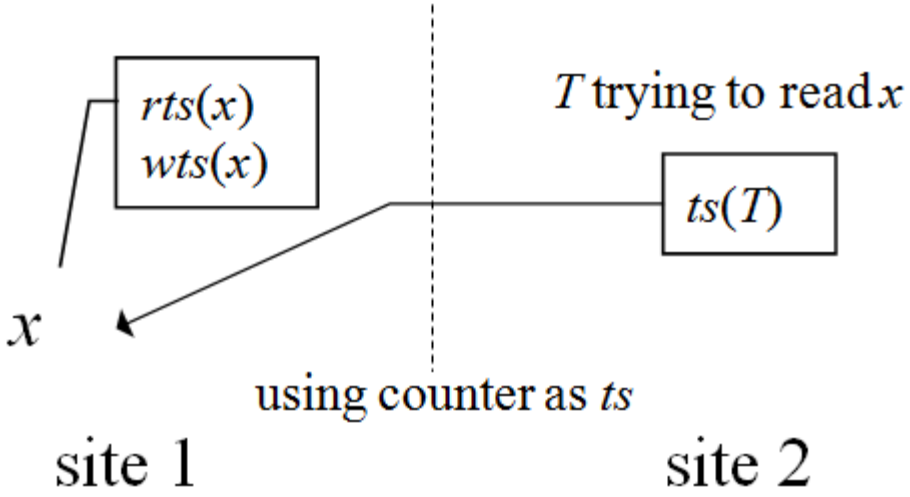
if *ts*(*Ti*) < *rts*(*x*) **or**
ts(*Ti*) < *wts*(*x*)

then reject *Wi*(*x*)

else accept *Wi*(*x*)

wts(*x*) ←*ts*(*Ti*)

Maintain Timestamps Among Sites



- When $ts(T) < wts(x)$, the read is rejected.
- Site 2 adjusts its timestamp by making it larger than $wts(x)$ and restart.
- Avoid $ts(\text{site 1}) >> ts(\text{site 2})$ which may make operations from site 2 never get executed.

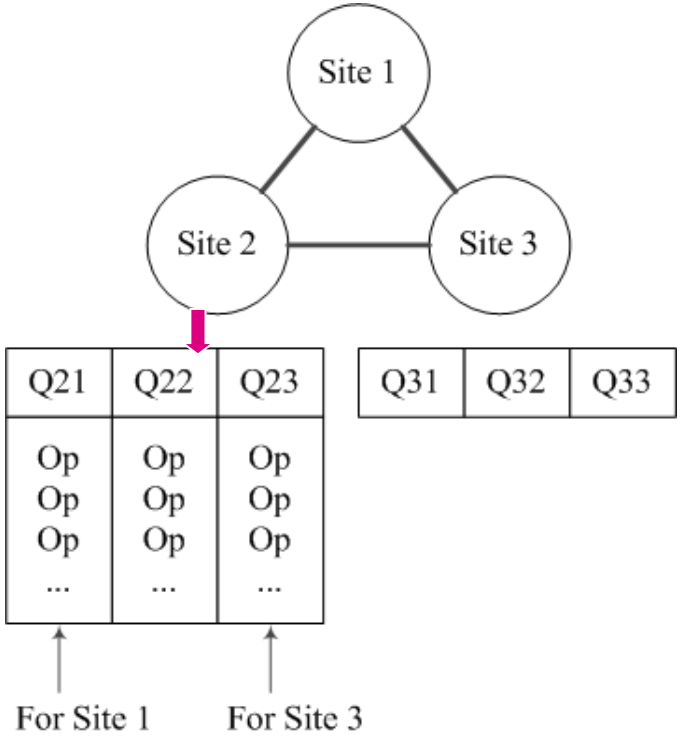
Conservative TO Algorithm

- Strength and weakness of Basic TO algorithm
  - deadlock free but too many restarts.

- Conservative TO algorithm
  - Reduce the number of restarts;
  - delay each operation until there is an assurance that no operation with smaller timestamps can arrive at that scheduler.

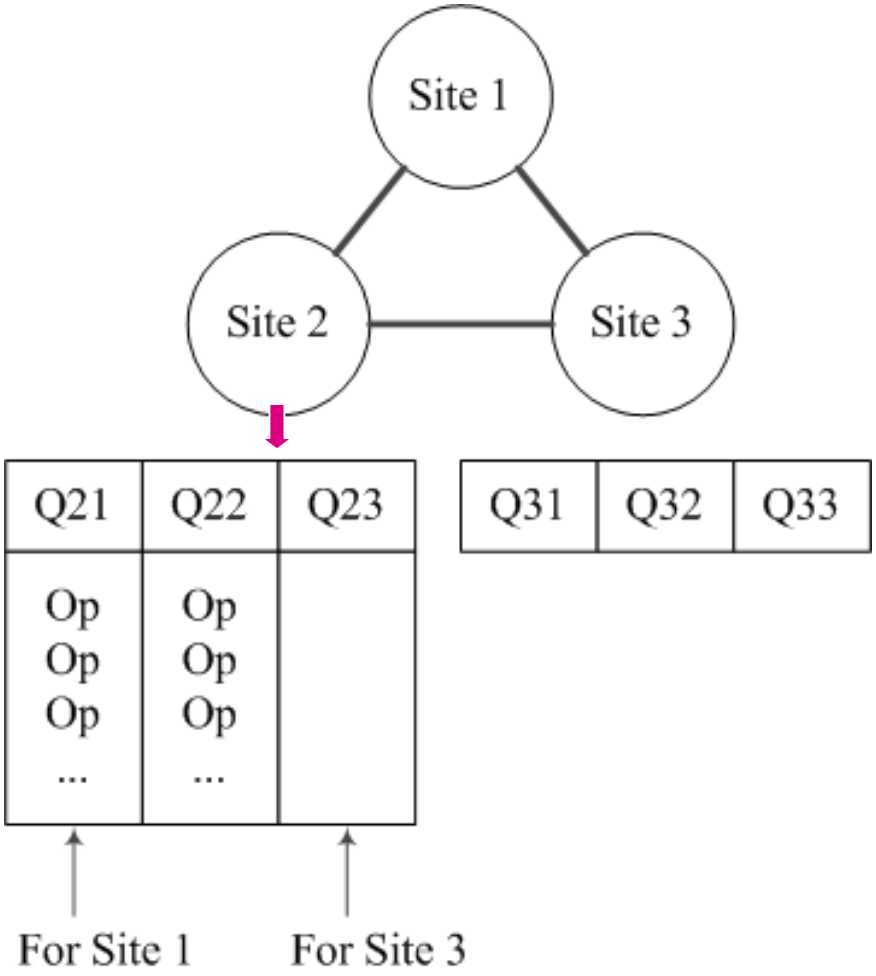
Conservative TO Technique

- At site i, each scheduler i has one queue Q_{ij} for each TM at site j in the system.
- An operation from TM_j is placed in Q_{ij} in increasing timestamp order.
- Scheduler i executes operations from all queues in this order also.



This reduces the number of restarts. But restart may occur when a queue is empty.

Conservative TO Technique (*cont.*)



- Suppose Q23 is empty and scheduler 2 chooses an operation op from Q21 and Q22.
- But later a conflicting operation with smaller timestamp than ts(op) from site 3 arrives, then this operation must be rejected and restarted!

Improvement ?

Improvement 1

- Use **extremely conservative** algorithm, complemented by
  - in each queue, there is at least one operation, or
  - if a TM does not have a transaction to process, it has to send a message periodically to inform that in the future it will send timestamp larger than that of the message.

Problems:

Extremely conservative algorithm actually executes transaction serially at each site – too conservative !

An operation may have to wait for the coming of a younger operation of an empty queue – a delay problem !

Improvement 2

- Define transaction classes and set a queue for each class instead of each TM
  - Transaction classes are defined by their read set and write set:

    If a transaction's read set and write set are subset of a class, then the transaction belongs to that class.

Difficulty:

It is difficult to define transaction classes!

Multiversion TO Algorithm

- To eliminate restart overhead costs
- Updates do not modify the database; each write creates a new version of the data item to be updated.
- Versions are transparent to users.
- Transactions are processed on a state of database that it would have seen if they were executed serially.

Work of the Scheduler

For a read $R_i(x)$ from transaction T_i

1. Find a version of x (say x_v) such that $ts(x_v)$ is the largest timestamp less than $ts(T_i)$;
2. Send $R_i(x_v)$ to the data processor.

Work of the Scheduler (*cont*.)

For a write $W_i(x)$ from transaction T_i

A $W_i(x)$ is translated into $W_i(x_w)$ with $ts(x_w)=ts(T_i)$ and (accepted) sent to the data processor,

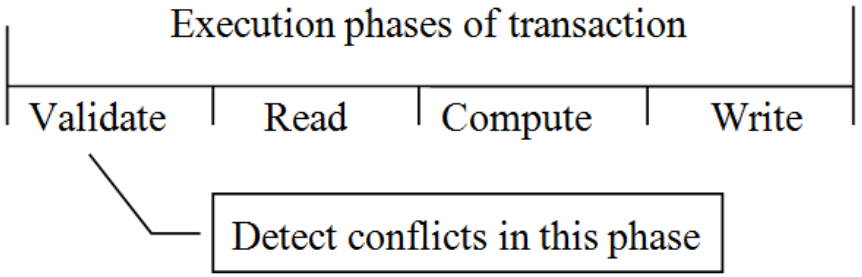
if and only if no other transaction with a timestamp greater than $ts(T_i)$ has read the value of a version of x (say x_r), where $ts(x_r)>ts(x_w)$.

Work of the Scheduler (*cont*.)

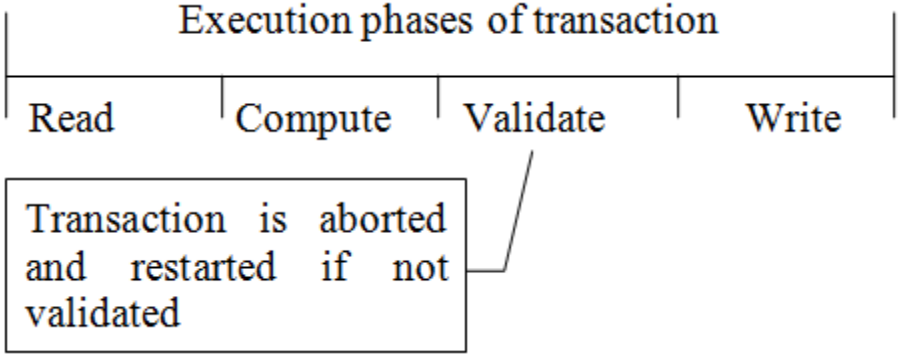
- This scheduler generates serializable schedule.
- Versions may be purged when they are not accessed by any transactions.

Optimistic Concurrency Control Algorithms

- Pessimistic algorithms assume conflicts happen quite often.



- Optimistic algorithms delay the validation phase until write phase.



Concepts

- Transaction execution model
  - Divide a transaction into sub-transactions, each of which executes at a site
  - T_{ij}: transaction T_i that executes at site j
- Transactions run independently at each site until reaching the end of their read phases
- All sub-transactions are assigned a timestamp at the end of their read phase

Optimistic Strategy

- Delay the validation phase until just before the write phase
  - Thus an operation submitted to an optimistic scheduler is never delayed.

- The read, compute, and write operations of each transaction are processed freely without updating the actual database.
  - Each transaction initially makes its updates on local copies of data items.

Optimistic Strategy (*cont*.)

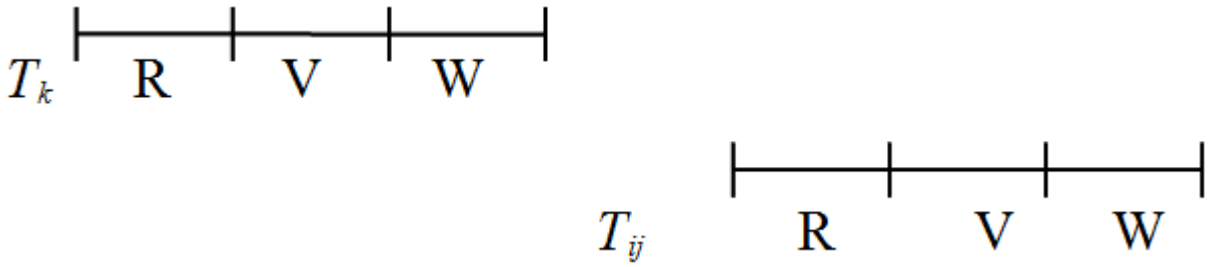
- The validation phase consists of checking if these updates would maintain the consistency of the database.

- If the answer is affirmative, the changes are made global (i.e., written into the actual database); Otherwise, the transaction is aborted and has to restart.

Local Validation of T_{ij} – Rule 1

- If all T_k with $ts(T_k) < ts(T_{ij})$ have completed before T_{ij} has started, then the validation succeeds.

- This is a serial execution order.

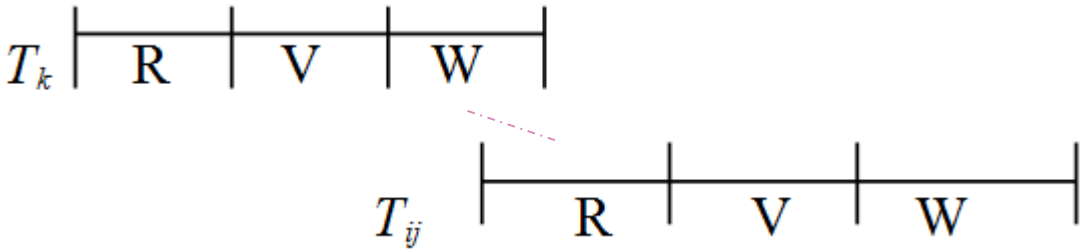


Local Validation of T_{ij} – Rule 2

- If there is any transaction T_k, such that
  - $ts(T_k) < ts(T_{ij})$
  - T_k is in write phase
  - T_{ij} is in read phase, and
  - $\mathrm{WS}(T_k) \cap \mathrm{RS}(T_{ij}) = \varnothing$

  then the validation succeeds.

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



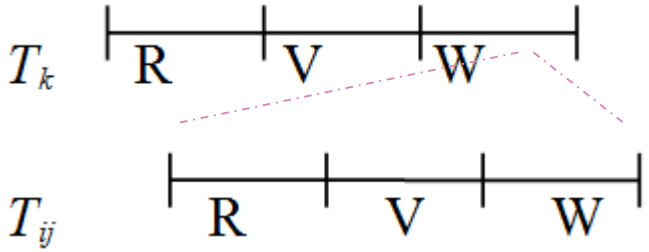
- None of the data items updated by T_k are read by T_{ij};
- Updates of T_{ij} will not be overwritten by the updates of T_k

Local Validation of T_{ij} – Rule 3

- If there is any transaction T_k, such that
  - $ts(T_k) < ts(T_{ij})$
  - T_k completes the read phase before T_{ij} completes the read phase
  - $\mathrm{WS}(T_k) \cap \mathrm{RS}(T_{ij}) = \varnothing$, and
  - $\mathrm{WS}(T_k) \cap \mathrm{WS}(T_{ij}) = \varnothing$

then the validation succeeds.

They overlap, but does not access any common data items.



- Update of T_k will not affect the read phase or write phase of T_{ij}.

Global Validation

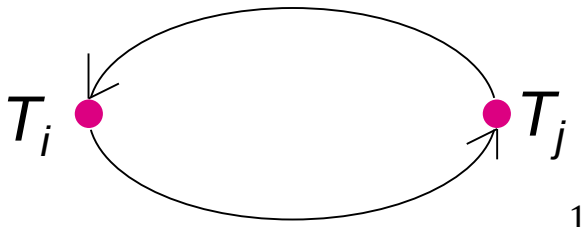
- Local validation ensures local database consistency.
- However, there is no known optimistic method for doing global validation!
- One possible way of doing it
  - A transaction is globally validated if all the transactions preceding it terminate (either by committing or aborting) in the serialization order (at that site).
  - It guarantees the transactions execute in the same order at each site.
  - This is a pessimistic strategy, as it performs global validation early and delays a transaction

Advantages and Disadvantages

- Advantage of optimistic algorithm
  - higher concurrency
- Disadvantage
  - higher storage cost for ordering of transactions

Deadlock

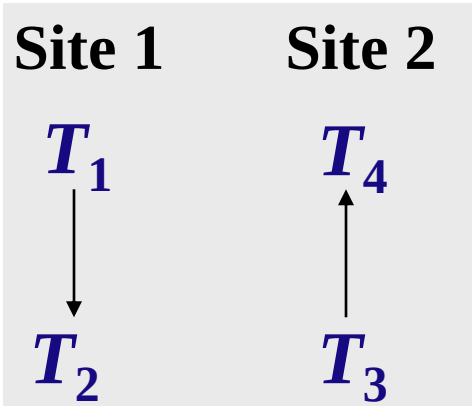
- A transaction is deadlocked if it is blocked and will remain blocked until there is intervention.
- Locking based CC algorithms may cause deadlocks.
- TO based algorithms that involve waiting may cause deadlocks.
- WaitFor Graph (WFG)
  - If transaction T_i waits for another transaction T_j to release a lock on an entity, then $T_i \rightarrow T_j$ in WFG.



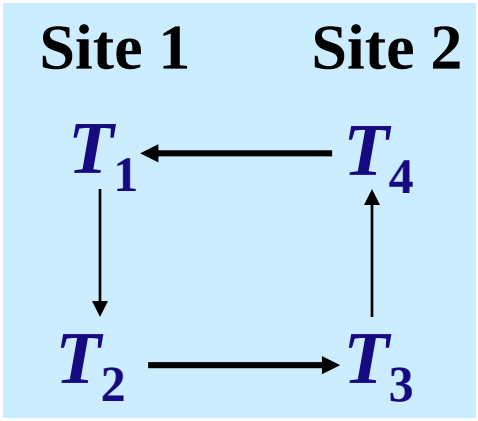
Local versus Global WFG

- Assume T_1 and T_2 run at site 1, T_3 and T_4 run at site 2.
- 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**
  - Guarantee that deadlocks can never occur in the first place. Check transaction when it is initiated. Require no run time support.
- **Avoidance**
  - Detect potential deadlocks in advance and take action to ensure that deadlock will not occur. Require run time support.
- **Detection and Recovery**
  - Allow deadlocks to form and then find and break them. Require run time support.

Deadlock Prevention

- All resources which may be needed by a transaction must be pre-declared.
  - 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 in database environments
  - 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.
  - Difficult to determine
  + 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 or may not 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 - WaitDie

- 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
  - younger transaction is aborted

Deadlock Avoidance - WoundWait

- 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
  - younger transaction is aborted

Deadlock Detection

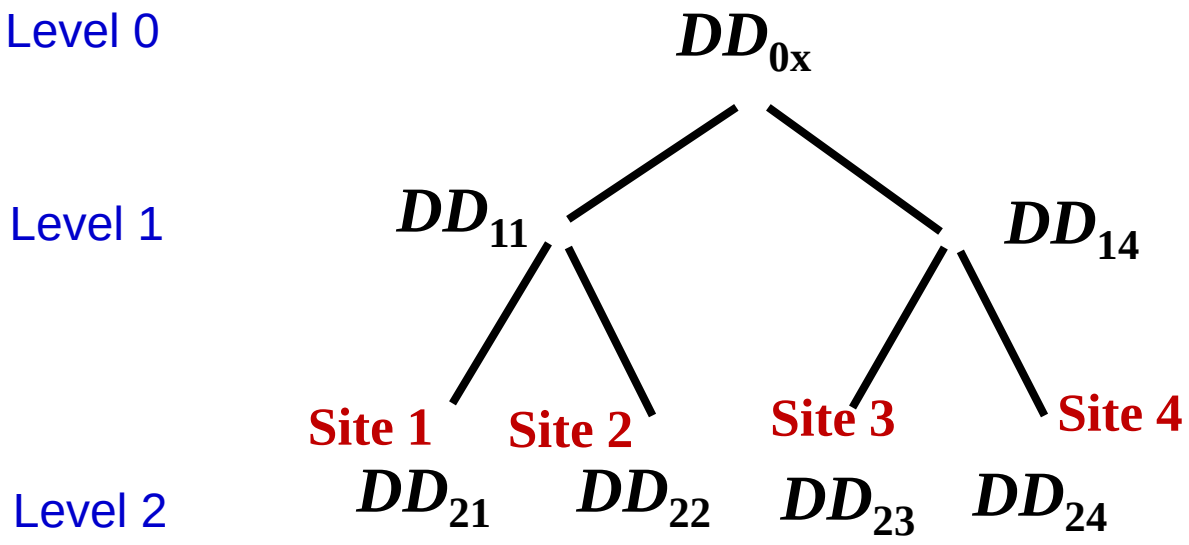
- Transactions are allowed to wait freely.
- Waitfor graphs and cycles.
- Topologies for deadlock detection algorithms
  - Centralized
  - Hierarchical
  - Distributed

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
- A reasonable choice if the concurrency control algorithm is also centralized.

Hierarchical Deadlock Detection

- Organize sites into a hierarchy and deadlock detectors (DD) send local graphs to parent in the hierarchy



Building a hierarchy of deadlock detectors

Distributed Deadlock Detection

Sites cooperate in detection of deadlocks

- 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.

Distributed Deadlock Detection (*cont*.)

- 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.

Question & Answer