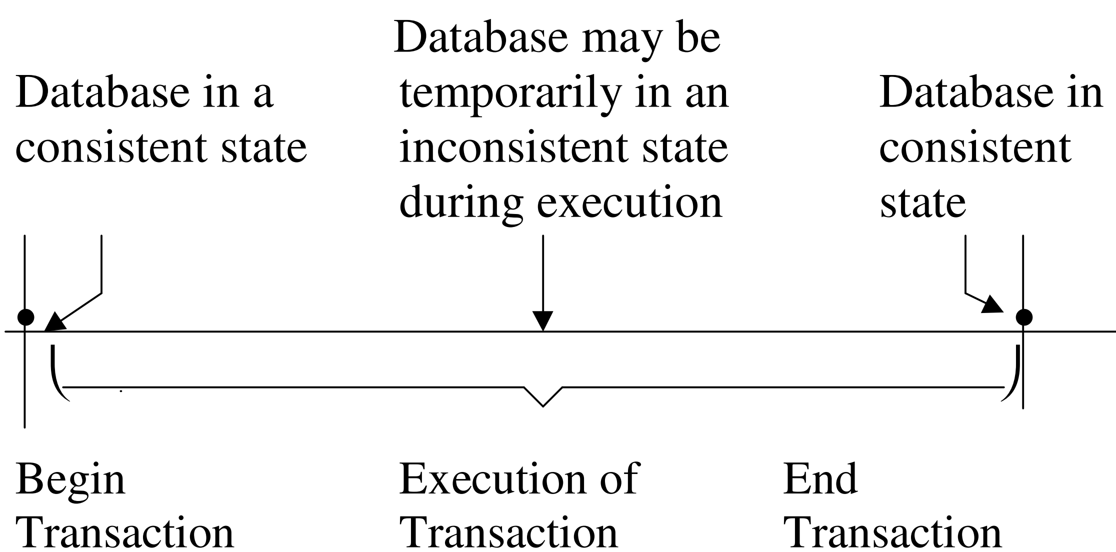
Transaction

A *transaction* is a collection of actions that make consistent transformations of system states while preserving system consistency.

- Concurrency transparency
- Failure transparency



Transaction Example – A Simple SQL Query

Transaction BUDGET_UPDATE

begin

```
EXEC SQL   UPDATE   J
           SET      BUDGET = BUDGET*1.1
           WHERE    JNAME = "CAD/CAM"
```

end.

Example Database

Consider an airline reservation example with the relations:

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

CUST(CNAME, ADDR, BAL)

FC(FNO, DATE, CNAME, SPECIAL)

STSOLD = no of tickets sold
CAP = capacity

Example Transaction – SQL Version

```
Begin_transaction Reservation
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 . {Reservation}
```

Termination of Transactions

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

end . {Reservation}
```

Characterization

- 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 (BS)
  - $RS \cup WS$

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

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

  $O_{ij} <_i O_{ik}$ *or* $O_{ik} <_i O_{ij}$

- $\forall\ O_{ij} \in OS_i$, $O_{ij} <_i N_i$

Example

Consider a transaction *T*:

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

Then

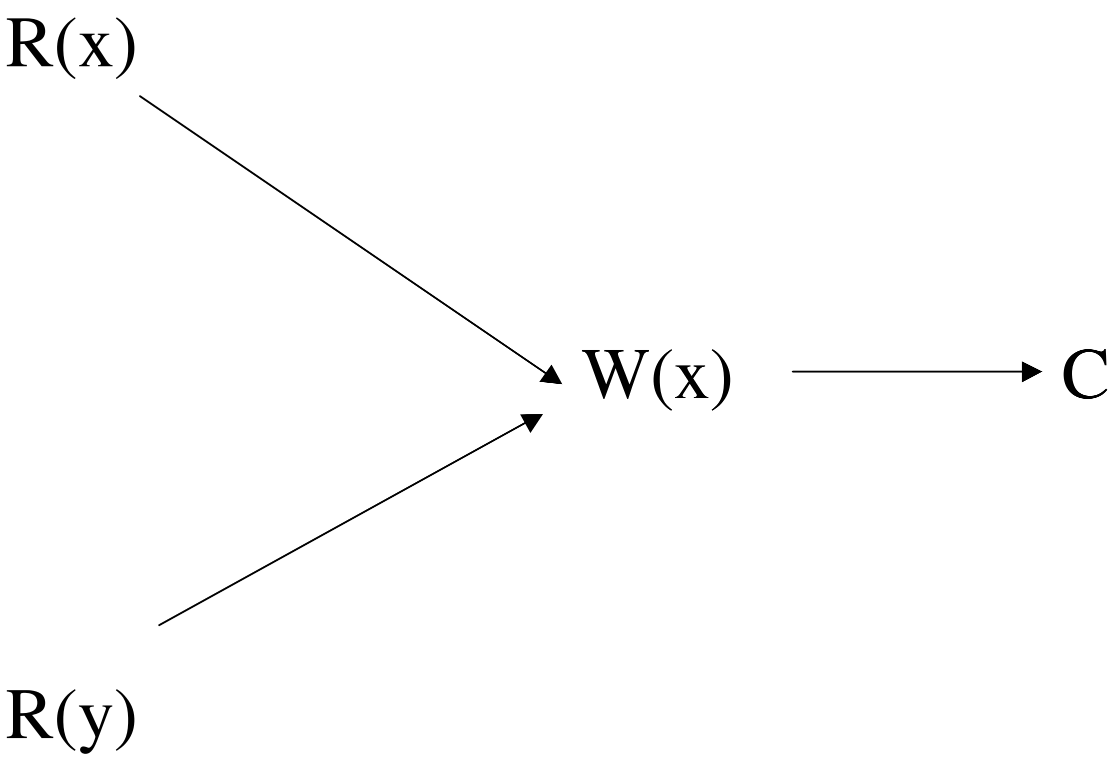
$$\Sigma = \{R(x), R(y), W(x), C\}$$

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

DAG Representation

Assume

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



Directed Acyclic graph (DAG)

Properties of Transactions

ATOMICITY

- all or nothing

CONSISTENCY

- no violation of integrity constraints

ISOLATION

- concurrent changes invisible
  i.e. serializable

DURABILITY

- committed updates persist

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.
- Transactions are correct programs

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 cascading aborts.

Durability

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

- Database recovery

Characterization of Transactions

Based on

- Application areas
  - non-distributed vs. distributed
  - compensating transactions
  - heterogeneous transactions
- Timing
  - on-line (short-life) vs
    batch (long-life)
- Structure
  - flat (or simple) transactions
  - nested transactions
- Organization of read and write actions
  - two-step
  - restricted
  - action model

Transaction Structure

- Flat transaction
  - ➢ Consists of a sequence of primitive operations embraced between a **begin** and **end** markers.

    Begin_transaction Reservation

    …

    end.

- Nested transaction
  - ➢ The operations of a transaction may themselves be transactions.

    Begin_transaction Reservation

    …

    Begin_transaction Airline

    _ …

    end. {Airline}

    Begin_transaction Hotel

    …

    end. {Hotel}

    end. {Reservation}

Nested Transactions

- Have the same properties as their parents $\Rightarrow$ may themselves have other nested transactions.
- Introduces concurrency control and recovery concepts to within the transaction.
- Types

  - Closed nesting
    - Subtransactions begin *after* their parents and finish *before* them.
    - Commitment of a subtransaction is conditional upon the commitment of the parent (commitment through the root).

  - Open nesting
    - Subtransactions can execute and commit independently.
    - Compensation may be necessary.

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)

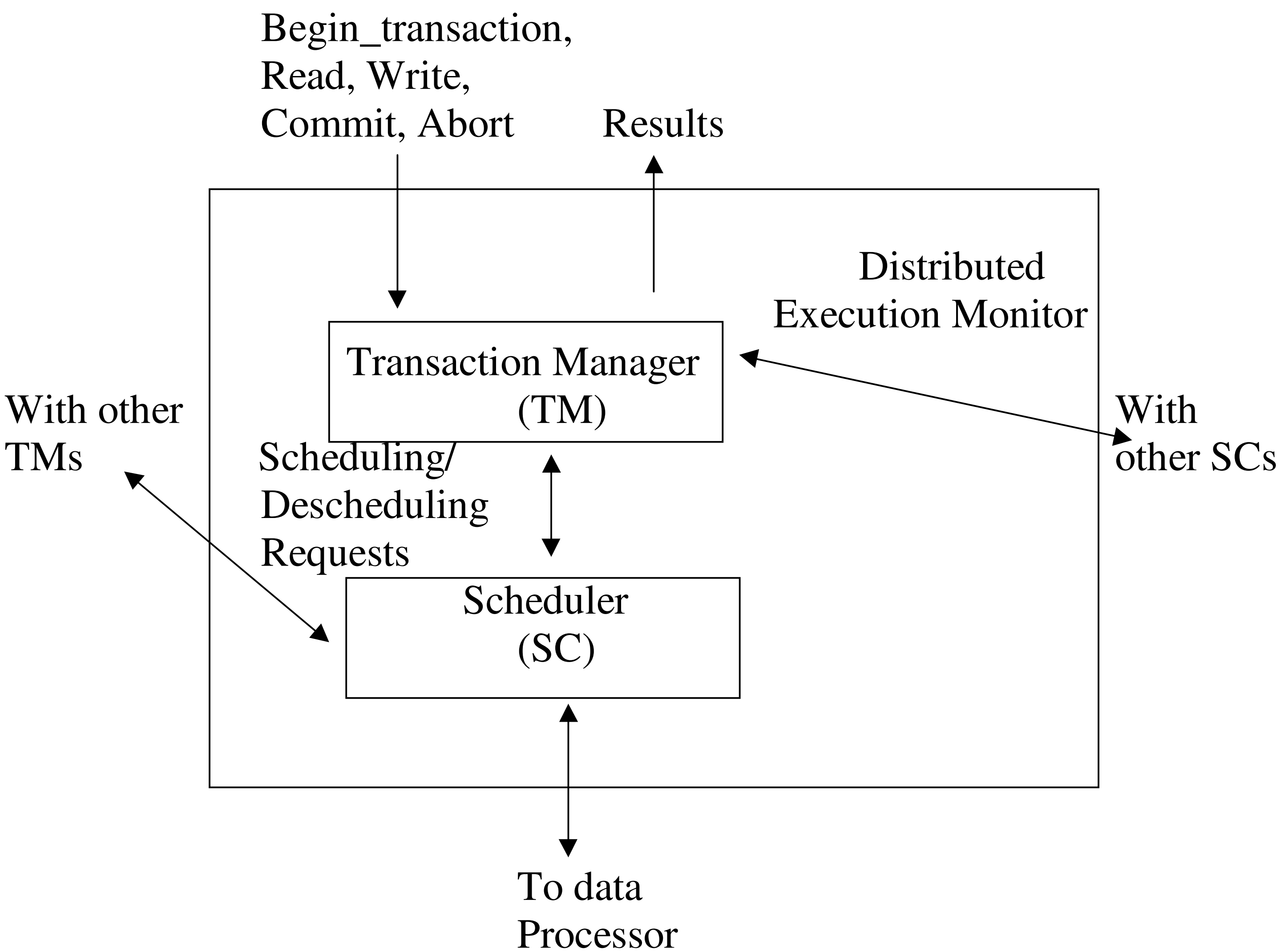
Transaction Processing Issues

- 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

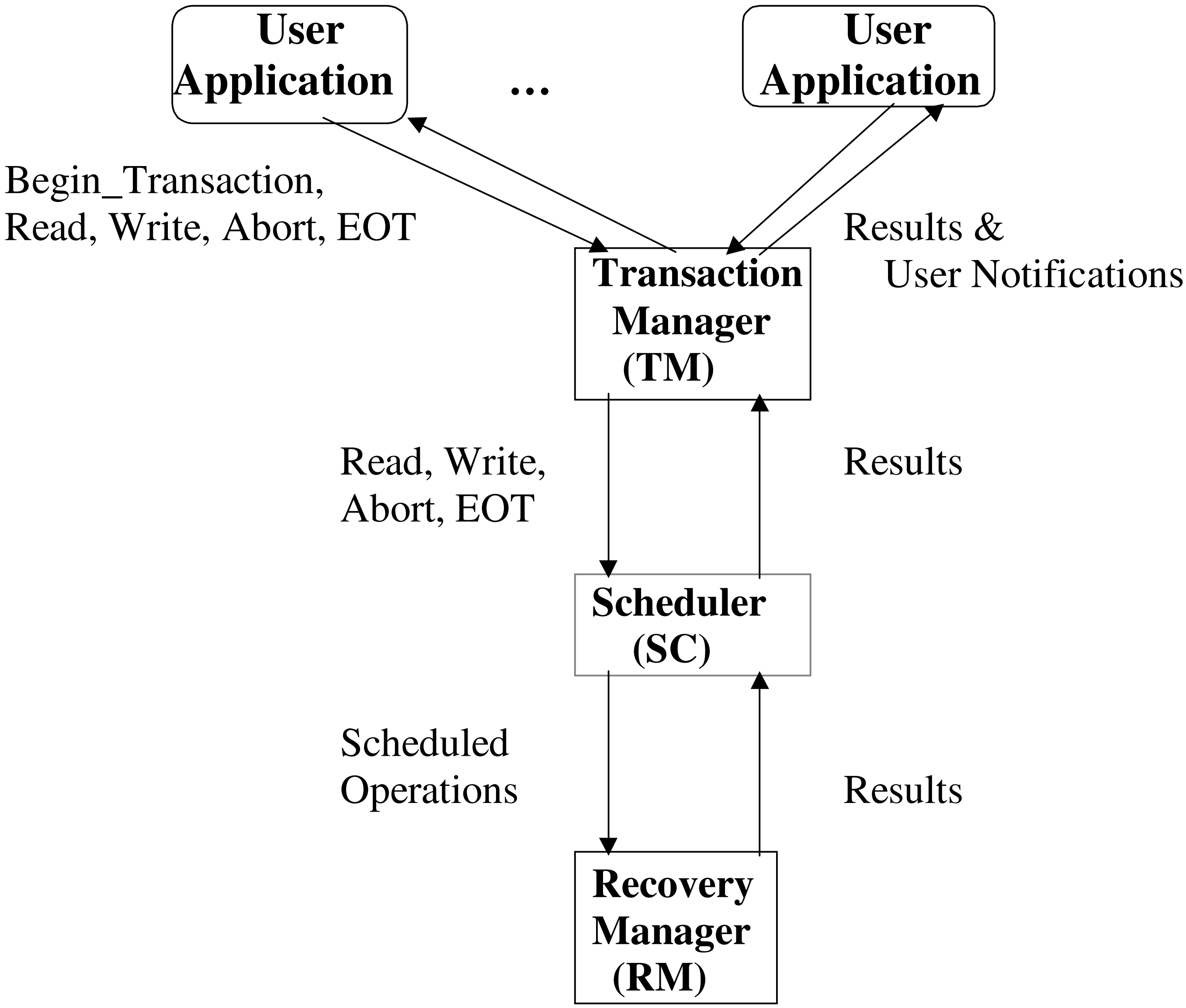
Transaction Processing Issues

- Concurrency control algorithms
  - How to synchronize concurrent transaction executions (correctness criterion)
  - Intra-transaction consistency, Isolation
- Replica control protocols
  - How to control the mutual consistency of replicated data
  - One copy equivalence and
    ROWA (read once write all)

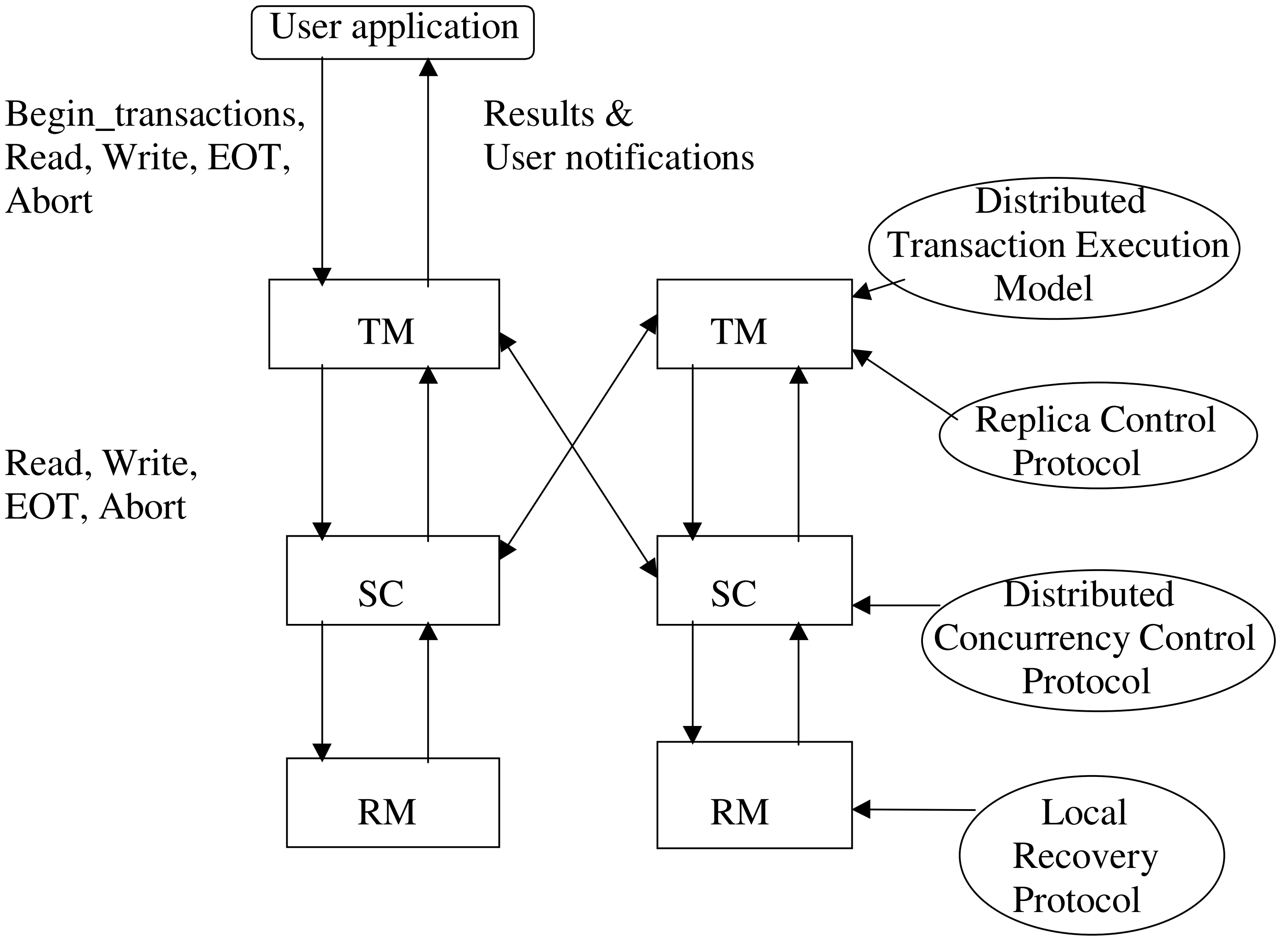
Architecture Revisited



Centralized Transaction Execution



Distributed 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 (or History)

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

A *complete schedule SC(T)* over a set of transactions $T=\{T_1, \ldots, T_n\}$ is a partial order $SC(T)=\{\Sigma_T, \prec_T\}$ where

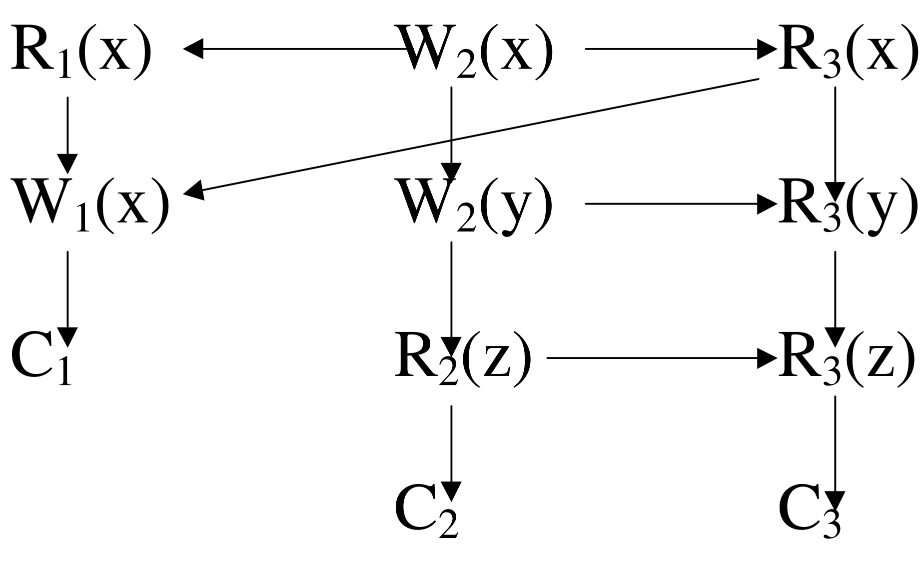
- $\Sigma_T = \cup_i \Sigma_i$, for $i = 1, 2, \ldots, n$
- $\prec_T \supseteq \cup_i \prec_i$, for $i = 1, 2, \ldots, n$
- For any two conflicting operations O_{ij}, $O_{kl} \in \Sigma_T$, either $O_{ij} \prec_T O_{kl}$ *or* $O_{kl} \prec_T O_{ij}$

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

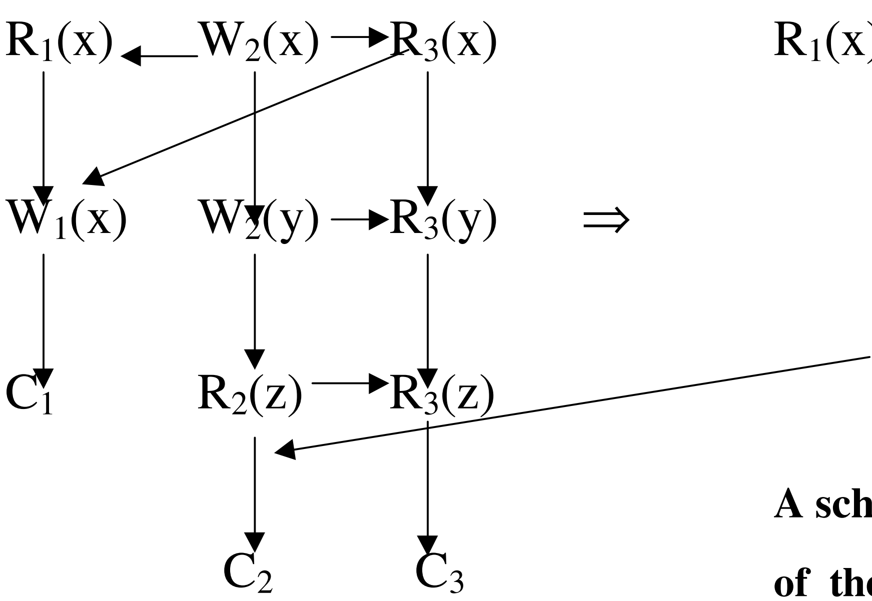
A possible complete schedule is given as the DAG

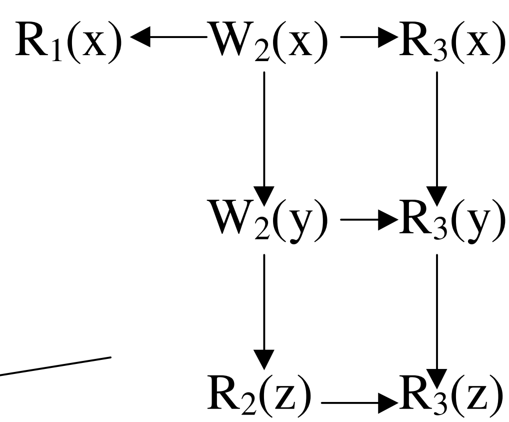


Schedule Definition

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





A schedule which is a prefix

of the complete schedule on the left

A complete schedule

(serializable?)

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

T2 < T1 < T3

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

Serializable History

- Transactions execute concurrently, but the net effect of the resulting history upon 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 (eg. 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: Read(x) Write(x) Commit	T_2: Write(x) Write(y) Read(z) Commit	T_3: Read(x) Read(y) Read(z) Commit

The following are not conflict equivalent

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

H_1= $\{W_2(x),R_1(x),R_3(x),W_1(x),C_1,W_2(y),R_3(y),R_2(z),C_2,R_3(z),C_3\}$

Why? $W_1(x) < R_3(x)$ in H_s but $R_3(x) < W_1(x)$ in H_1

The following are conflict equivalent; therefore H_2 is *serializable.*

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

H_2 = $\{W_2(x),R_1(x),W_1(x),C_1,R_3(x),W_2(y),R_3(y),R_2(z),C_2,R_3(z),C_3\}$

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

Consider 2 sites and one data item x that is duplicated in both sites

T_1: Read(x)
$x \leftarrow x+5$
Write(x)
Commit

T_2: Read(x)
$x \leftarrow x*10$
Write(x)
Commit

The following two local histories are individually serializable (in fact serial), but the two transactions are not globally serializable.

$$LH_1=\{R_1(x),W_1(x),C_1,R_2(x),W_2(x),C_2\}$$

$$LH_2=\{R_2(x),W_2(x),C_2,R_1(x),W_1(x),C_1\}$$

Assume x=1 initially. At site 1, x=60 after LH1.
At site 2, x=15 after LH2. Violates the mutual consistency.

Concurrency Control Algorithms

- Pessimistic
  - Two-Phase Locking-based (2PL)
    - Centralized (primary site) 2PL
    - Primary copy 2PL
    - Distributed 2PL
  - Timestamp Ordering (TO)
    - Basic TO
    - Multiversion TO
    - Conservative TO
  - Hybrid
- Optimistic
  - Locking-based
  - Timestamp ordering-based

Locking-Based Algorithms

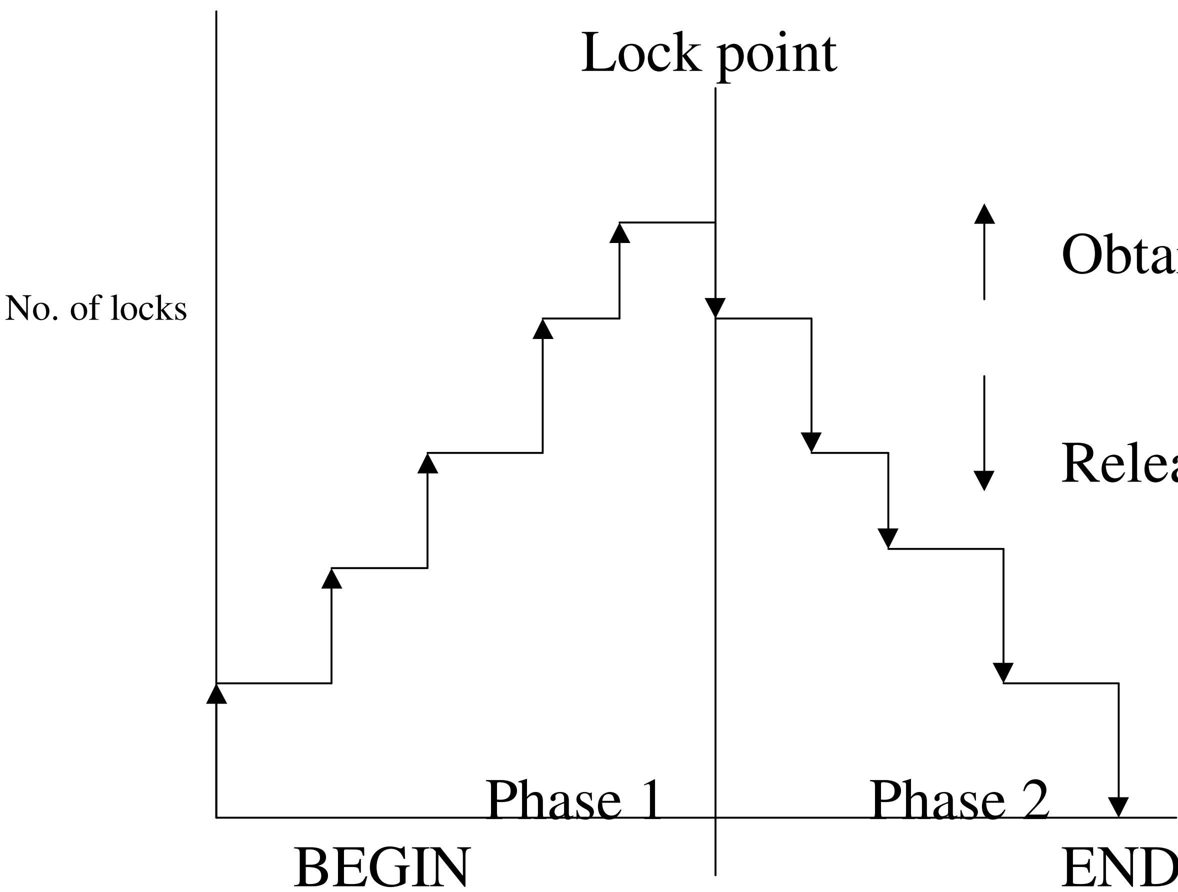
- Transactions indicate their intentions by requesting locks from the scheduler (called lock manager)
- Locks are either read lock (*rl*) [also called *shared lock*] or write lock (*wl*) [also called *exclusive lock*]
- Read locks and write locks conflict (because Read and Write operations are incompatible

	rl	*wl*
rl	yes	no
wl	no	no

- Locking works nicely to allow concurrent processing of transactions.

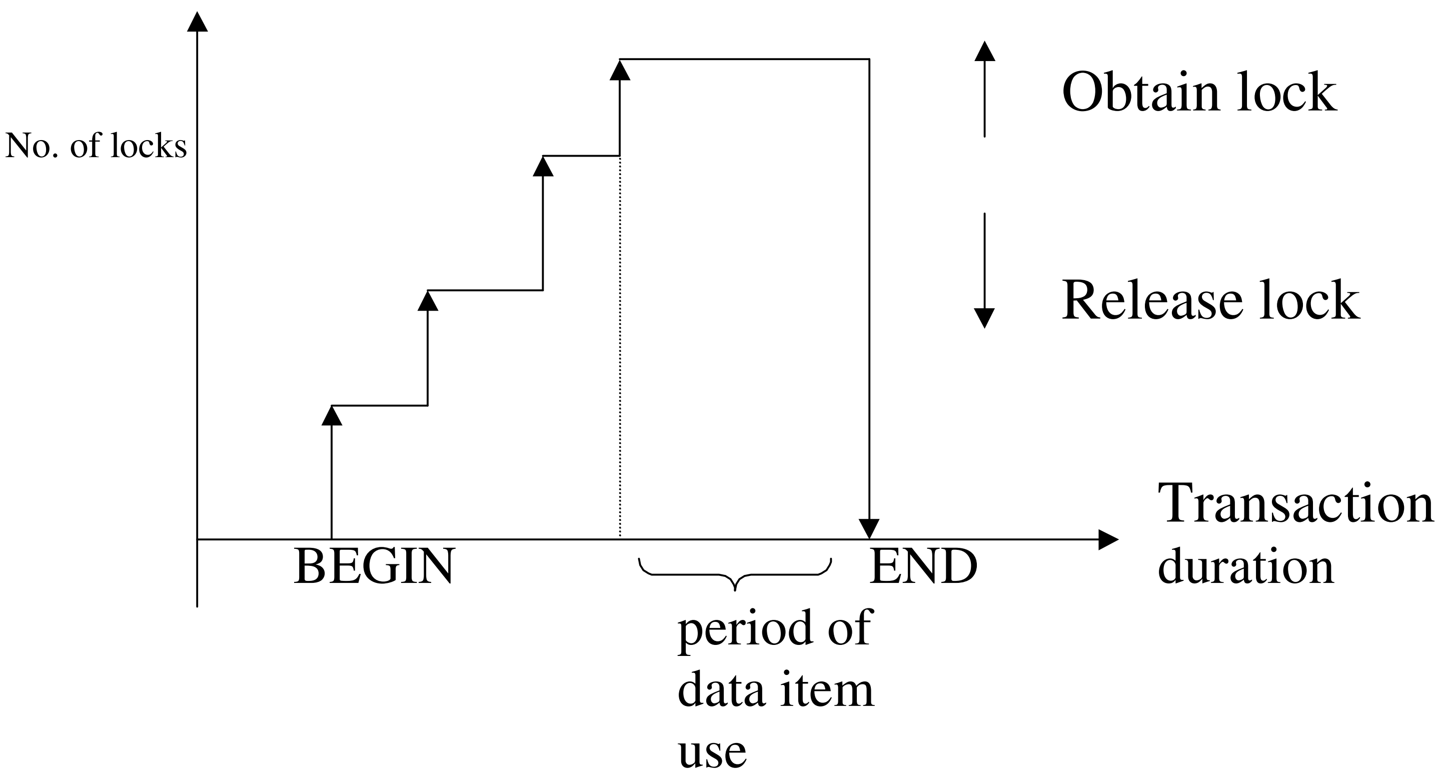
Two-Phase Locking (2PL)

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



Strict 2PL

Hold locks until the end.

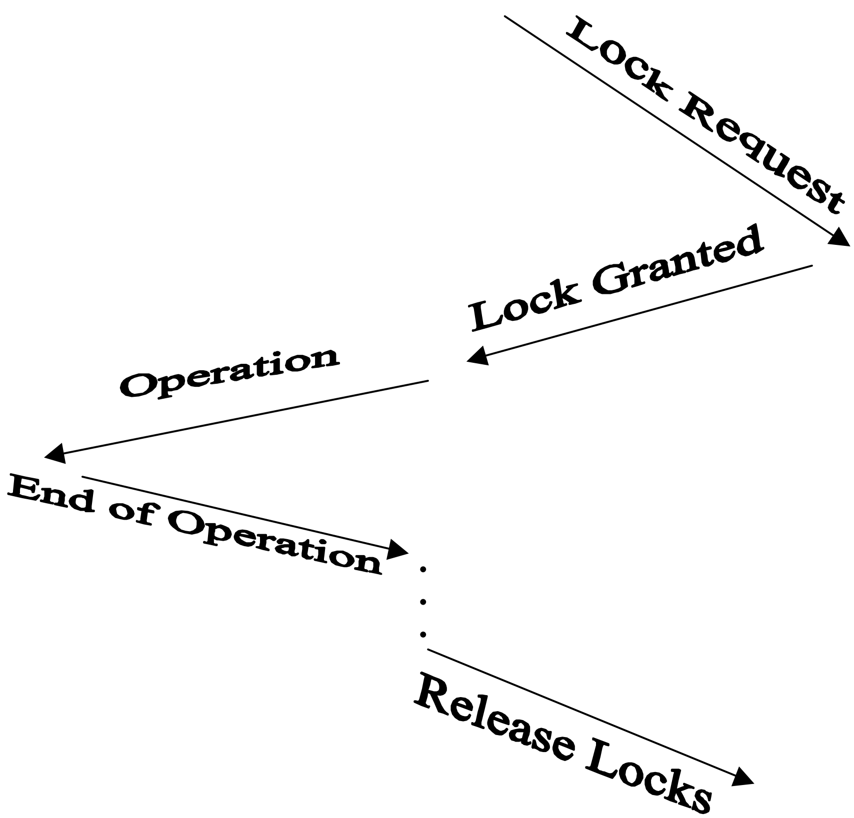


2PL Theorem: If all transactions in an execution are two-phase locked, then the execution is serializable.

Centralized 2PL

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

Data Processors at
Participating sites Coordinating TM Central Site LM

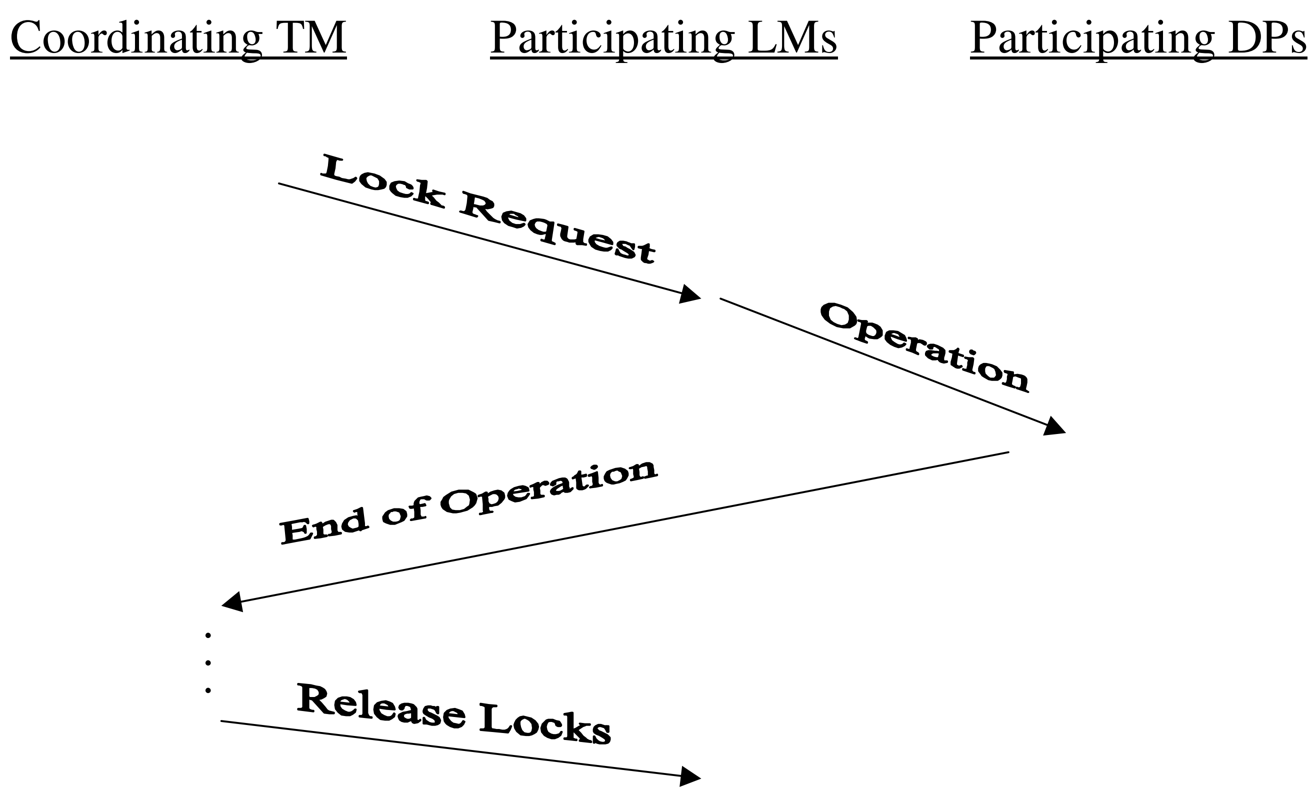


Distributed 2PL

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

- A transaction may read any of the replicated copies of item *x*, by obtaining a read lock on one of the copies of *x*. Writing into *x* requires obtaining write locks for all copies of *x*.

Distributed 2PL Execution



Timestamp Ordering

- Transaction (T_i) is assigned a globally unique timestamp $ts(T_i)$.

- Transaction manager attaches the timestamp to all operations issued by the transaction.

- Each data item is assigned a write timestamp (*wts*) and a read timestamp (*rts*):

  - rts(x) = largest timestamp of any read on *x*
  - wts(x) = largest timestamp of any write on *x*

- Conflcting operations are resolved by timestamp order.

Basic T/O:

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

for $W_i(x)$
if $ts(T_i) < rts(x)$ **or** $ts(T_i) < wts(x)$
then reject $W_i(x)$
else accept $W_i(x)$
$wts(x) \leftarrow ts(T_i)$
$rts(x) \longleftarrow ts(T_i)$

Conservative Timestamp Ordering

- Basic timestamp ordering tries to execute an operation as soon as it receives it

  - progressive
  - too many restarts since there is no delaying

- Conservative timestamping delays each operation until there is an assurance that it will not be restarted

- Assurance?

  - No other operation with a smaller timestamp can arrive at the scheduler
  - Note that the delay may result in the formation of deadlocks

Multiversion Timestamp Ordering

- Do not modify the values in the database, create new values.

- A $R_i(x)$ is translated into a read on one version of x.

  - Find a version of x (say x_v) such that $ts(x_v)$ is the largest timestamp less than $ts(T_i)$.

- A $W_i(x)$ is translated into $W_i(x_w)$ and accepted if the scheduler has not yet processed any $R_j(x_r)$ such that

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

Optimistic Concurency Control Algorithms

Pessimistic execution

Validate	Read	Compute	Write

Optimistic execution

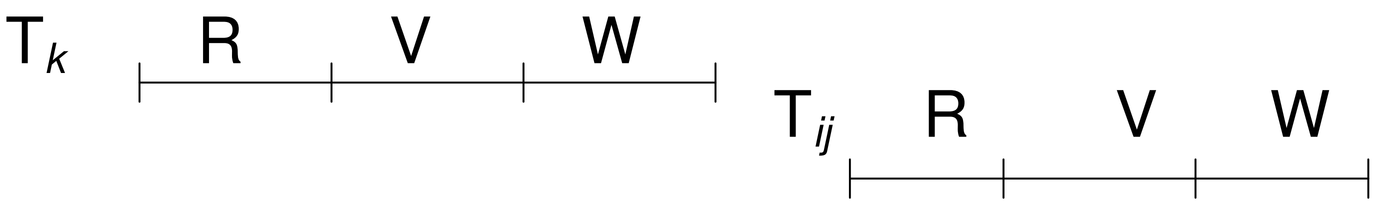
Read	Compute	Validate	Write

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.

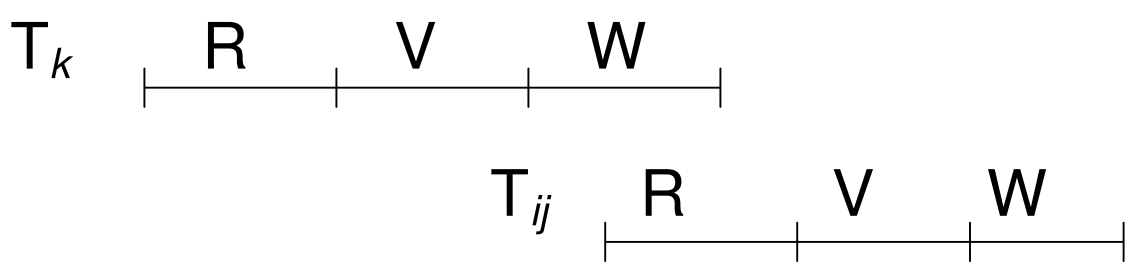
Optimistic CC Validation Test

- If all transaction 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



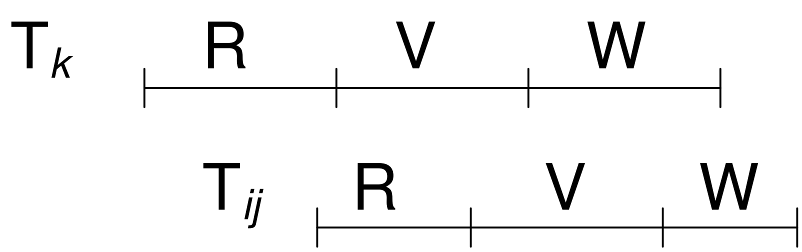
Optimistic CC Validation Test

- 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 $\mathrm{WS}(T_k) \cap \mathrm{RS}(T_{ij}) = \emptyset$
  - Read and write phases overlap, but T_{ij} does not read data items written by T_k



Optimistic CC Validation Test

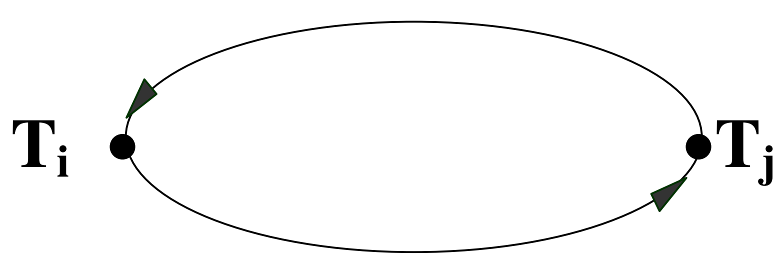
- If there is any transaction T_k such that $ts(T_k) < ts(T_{ij})$ and which completes its read phase before T_{ij} completes its read phase, then validation succeeds if $WS(T_k) \cap RS(T_{ij}) = \varnothing$ and $WS(T_k) \cap WS(T_{ij}) = \varnothing$
  - They overlap, but don't access any common data items.



Deadlock

- A transaction is deadlocked if it is blocked and will remain blocked until there is intervention.
- Locking-based CC algorithms may cause deadlocks.
- TO-based algorithms that involve waiting may cause deadlocks.
- Wait-for graph

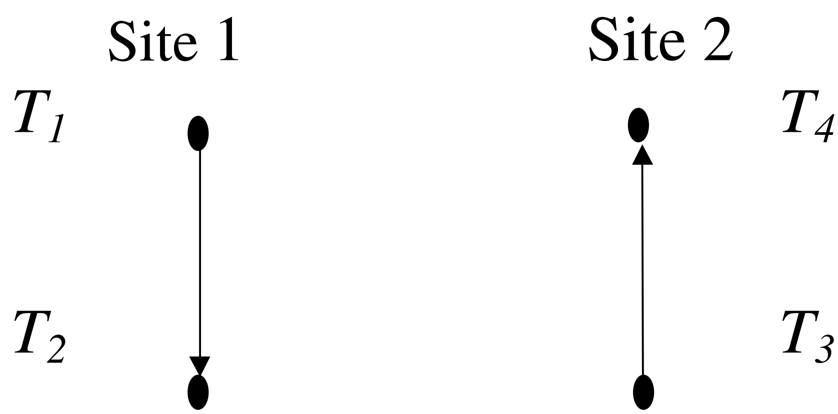
  - If transaction T_i waits for another transaction T_j to release a lock on an entity, then $T_i \rightarrow T_j$ in WFG.



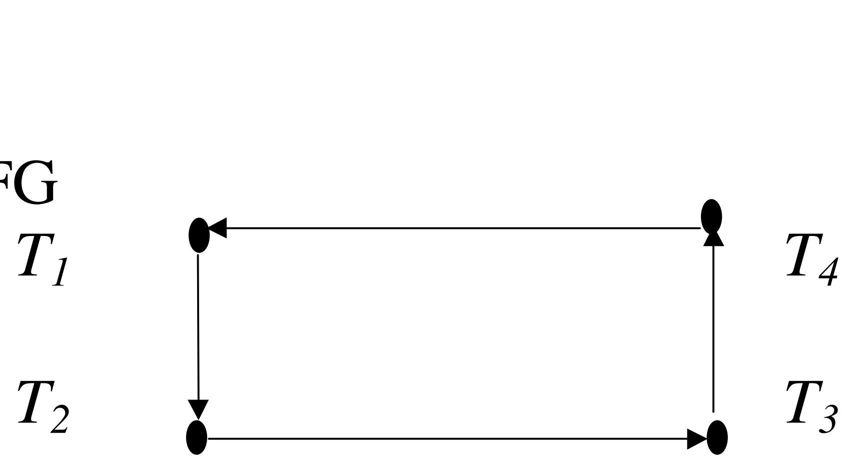
Local versus Global WFG

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 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 & Wound-Wait Algorithms

WAIT-DIE Rule: 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 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

WOUND-WAIT Rule: 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

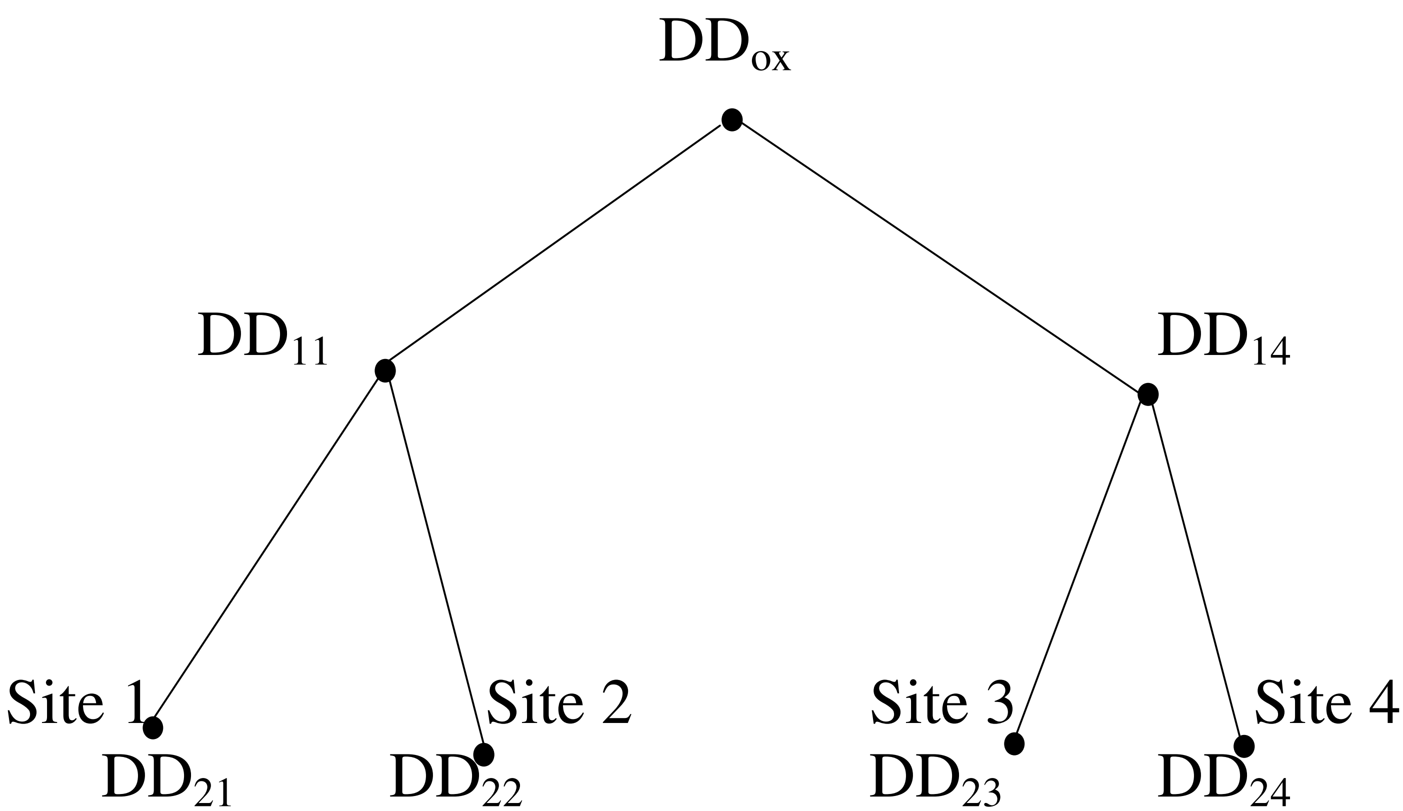
- One site is designated as the deadlock detector for the system. Each scheduler periodically sends its local WFG to the central site which merges them to a global WFG to determine cycles.

- 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



A deadlock at site 1 would be detected by the local deadlock detector DD_{21} at site 1.

If the deadlock involves sites 1 and 2, then DD_{11} detects it.

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:
    - Since each site receives the potential deadlock cycles from other sites, these edges are added to the local WFGs
    - The edges in the local WFG which show that local transactions are waiting for transactions at other sites are joined with edges in the local WFGs which show that remote transactions are waiting for local ones.

  - ➢ Each local deadlock detector:
    - ♦ looks for a cycle that does not involve the 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.