CSE 512: Distributed and Parallel Data Systems

Lecture 5

Instructor: Mohamed Sarwat

Transaction Management Overview

Transactions

Concurrent execution of user programs is essential for good DBMS performance.

Why ?

Transactions

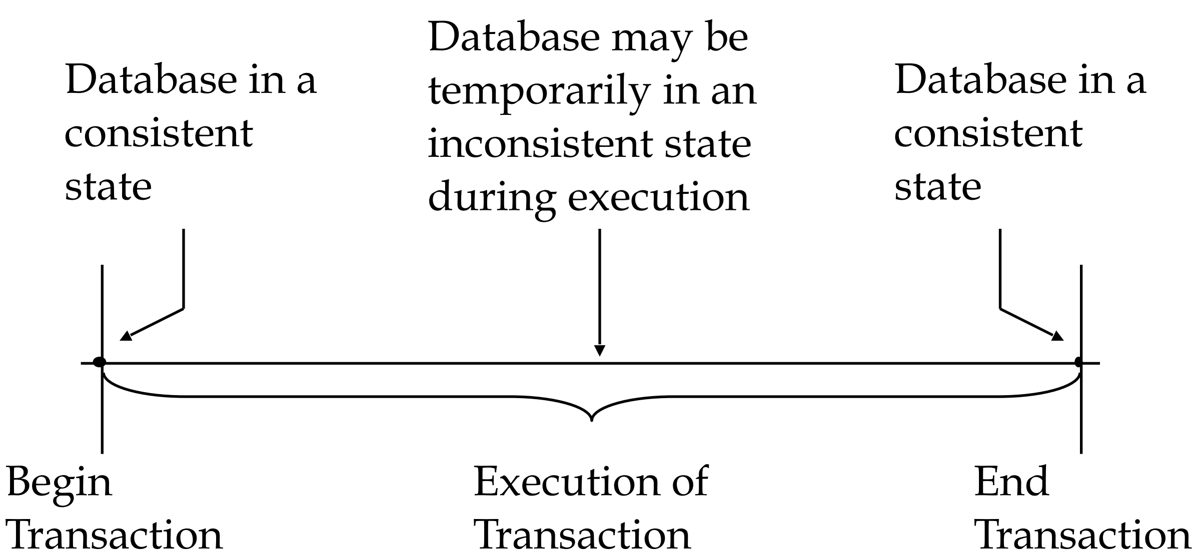
- Concurrent execution of user programs is essential for good DBMS performance.

- A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.

- A *transaction* is the DBMS's abstract view of a user program: a sequence of reads and writes.

Transaction

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



Concurrency in a DBMS

- Users submit transactions, and can think of each transaction as executing by itself.
  - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
  - Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
    - DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements.
    - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
- *Issues:* Effect of *interleaving* transactions, and *crashes*.

Principles of Transactions

ATOMICITY
- all or nothing

CONSISTENCY
- no violation of integrity constraints

ISOLATION
- concurrent changes invisible $\Rightarrow$ serializable

DURABILITY
- committed updates persist

Atomicity of Transactions

- A transaction might *commit* after completing all its actions, or it could *abort* (or be aborted by the DBMS) after executing some actions.
- A very important property guaranteed by the DBMS for all transactions is that they are *atomic.* That is, a user can think of a Xact as always executing all its actions in one step, or not executing any actions at all.
  - DBMS *logs* all actions so that it can *undo* the actions of aborted transactions.

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

Example

- Consider two transactions (*Xacts*):

```
T1:  BEGIN   A=A+100,   B=B-100   END
T2:  BEGIN   A=1.06*A,   B=1.06*B   END
```

- Intuitively, the first transaction is transferring $100 from B's account to A's account. The second is crediting both accounts with a 6% interest payment.
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect *must* be equivalent to these two transactions running serially in some order.

ARIZONA STATE UNIVERSITY

Example (Contd.)

- Consider a possible interleaving (*schedule*):

```
T1:   A=A+100,                  B=B-100
T2:              A=1.06*A,               B=1.06*B
```

- ❖ This is OK. But what about:

```
T1:   A=A+100,                        B=B-100
T2:              A=1.06*A, B=1.06*B
```

- ❖ The DBMS's view of the second schedule:

```
T1:   R(A), W(A),                               R(B), W(B)
T2:                R(A), W(A), R(B), W(B)
```

Scheduling Transactions

- *Serial schedule:* Schedule that does not interleave the actions of different transactions.
- *Equivalent schedules*: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions.

Interleaved Execution

S1

T1: R(A), W(A), R(B), W(B), Abort
T2: R(A), W(A), C

S2

T1: R(A), R(A), W(A), C
T2: R(A), W(A), C

S3

T1: W(A), W(B), C
T2: W(A), W(B), C

Identify Anomalies ?

Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

```
T1:  W(A),                W(B), C
T2:        W(A), W(B), C
```

Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, "dirty reads"):

```
T1:  R(A), W(A),                        R(B), W(B), Abort
T2:                R(A), W(A), C
```

- Unrepeatable Reads (RW Conflicts):

```
T1:  R(A),                      R(A), W(A), C
T2:          R(A), W(A), Abort
```

Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

```
T1:  W(A),              W(B), Abort
T2:        W(A), W(B), C
```

Conflict Serializable Schedules

- Two schedules are conflict equivalent if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way
- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	

Conflict Serializable?

T1:	R(A), W(A),		R(B), W(B)	
T2:		R(A), W(A),		R(B), W(B)

Conflict Serializable?

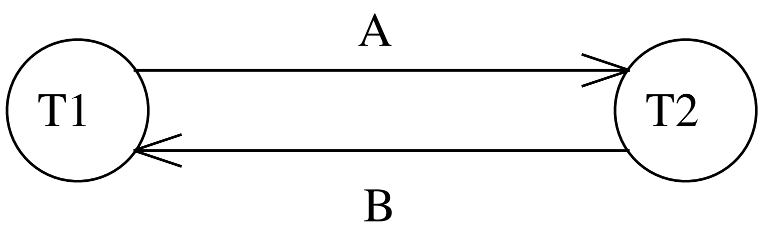
T1:	R(A), W(A),	R(B), W(B)		
T2:			R(A), W(A),	R(B), W(B)

Conflict Serializable?

Example

- A schedule that is not conflict serializable:

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	



Dependency graph

- The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

Dependency Graph

- *Dependency graph*: One node per Xact; edge from *Ti* to *Tj* if *Tj* reads/writes an object last written by *Ti*.
- Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic

Lock-Based Concurrency Control

- *Strict Two-phase Locking (Strict 2PL) Protocol*:
  - Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes
    - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- Strict 2PL allows only serializable schedules.
  - Additionally, it simplifies transaction aborts

Lock Management

- Lock and unlock requests are handled by the lock manager
- Lock table entry:
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations
- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock

Locking-Based Algorithms

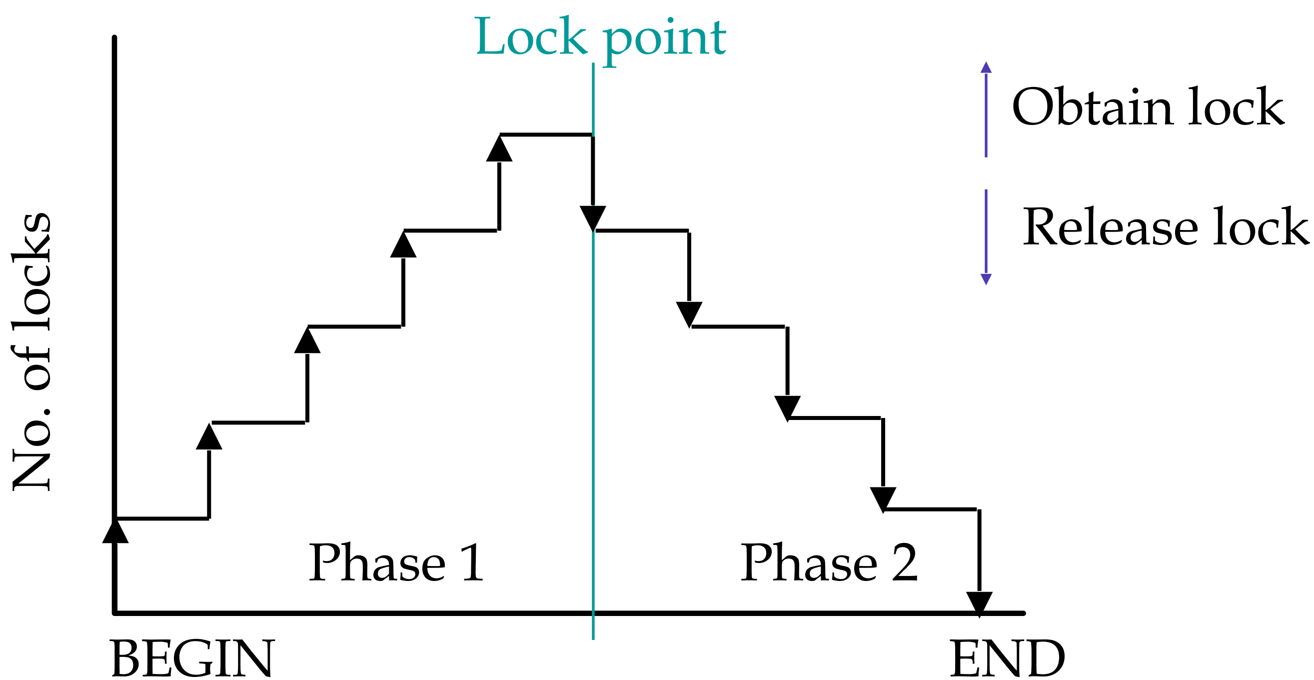
- Transactions indicate their intentions by requesting locks from the scheduler (called lock manager).
- Locks are either read lock (*rl*) [also called shared lock] or write lock (*wl*) [also called exclusive lock]
- Read locks and write locks conflict (because Read and Write operations are incompatible

	rl	*wl*
rl	yes	no
wl	no	no

- Locking works nicely to allow concurrent processing of transactions.

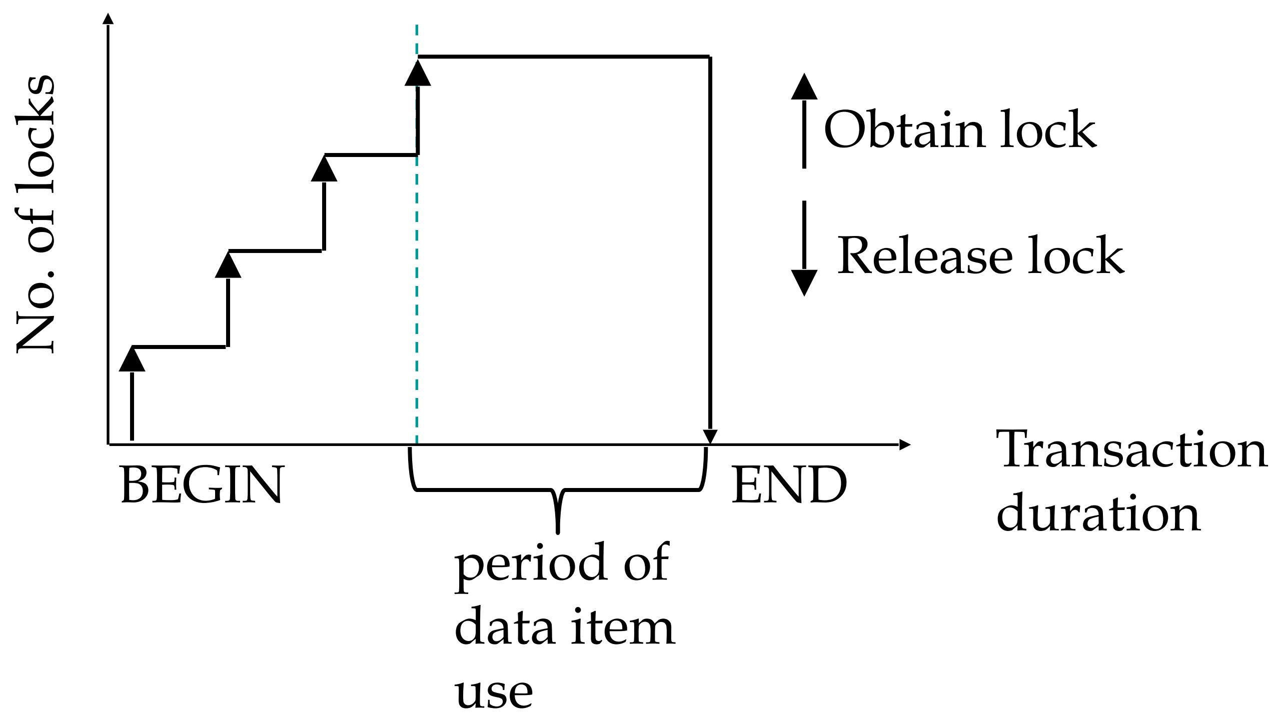
Two-Phase Locking (2PL)

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.



Strict 2PL

Hold locks until the end.



Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.

Deadlock Detection

- Create a waits-for graph:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Periodically check for cycles in the waits-for graph

Arizona State University

Deadlock Detection (Continued)

Example:

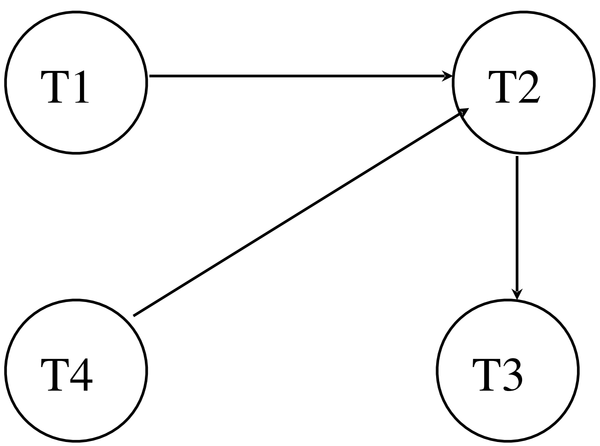
T1:	S(A), R(A),		S(B)			
T2:		X(B),W(B)			X(C)	
T3:				S(C), R(C)		
T4:						X(B)

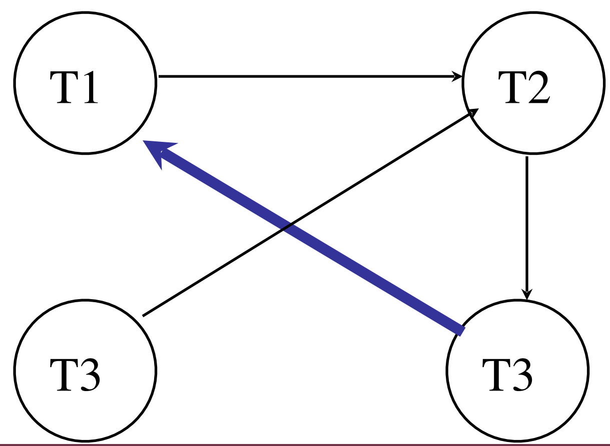
Waits-for-Graph ?

Deadlock Detection (Continued)

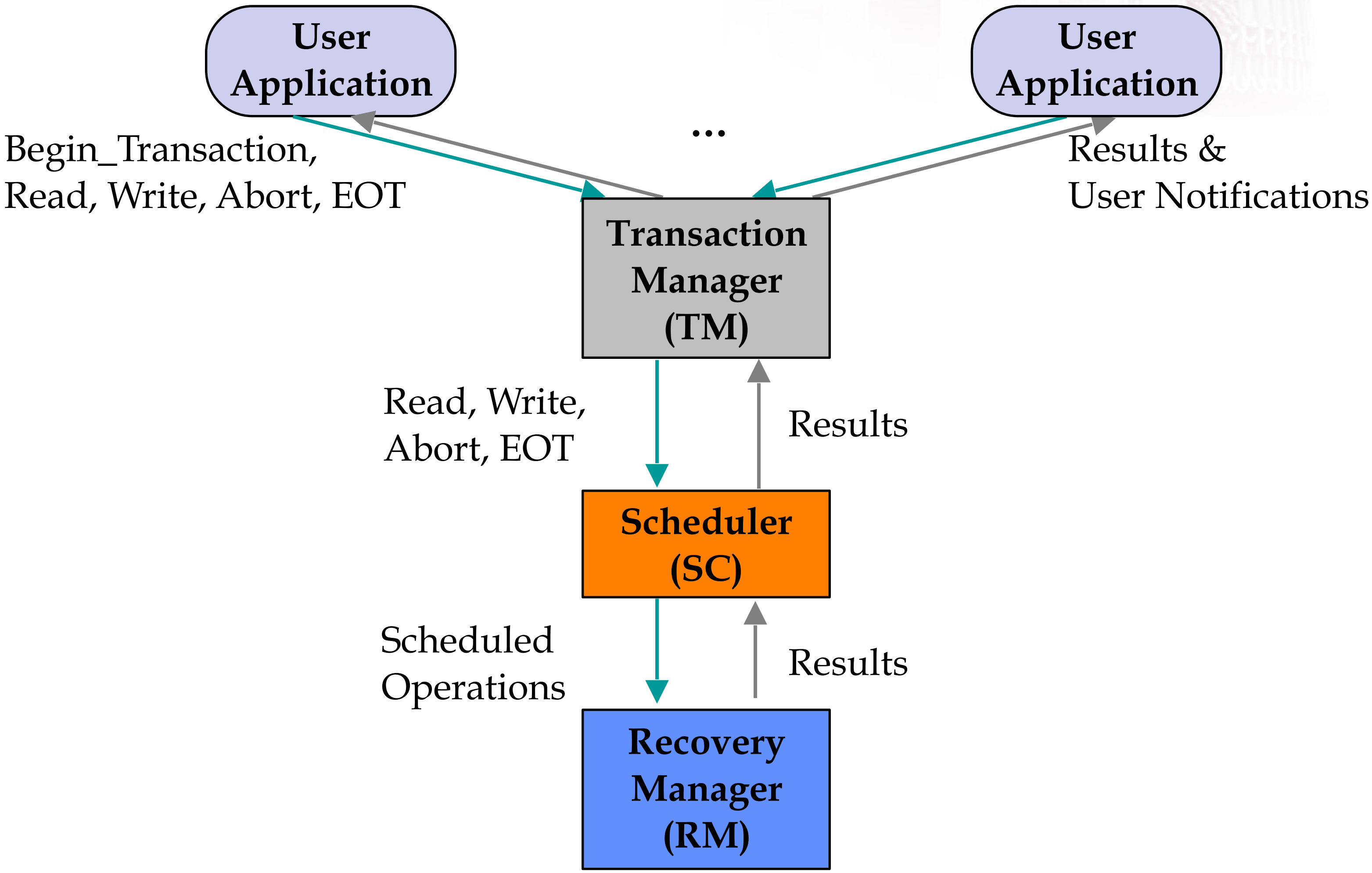
Example:

T1: S(A), R(A), S(B)
T2: X(B),W(B) X(C)
T3: S(C), R(C) X(A)
T4: X(B)

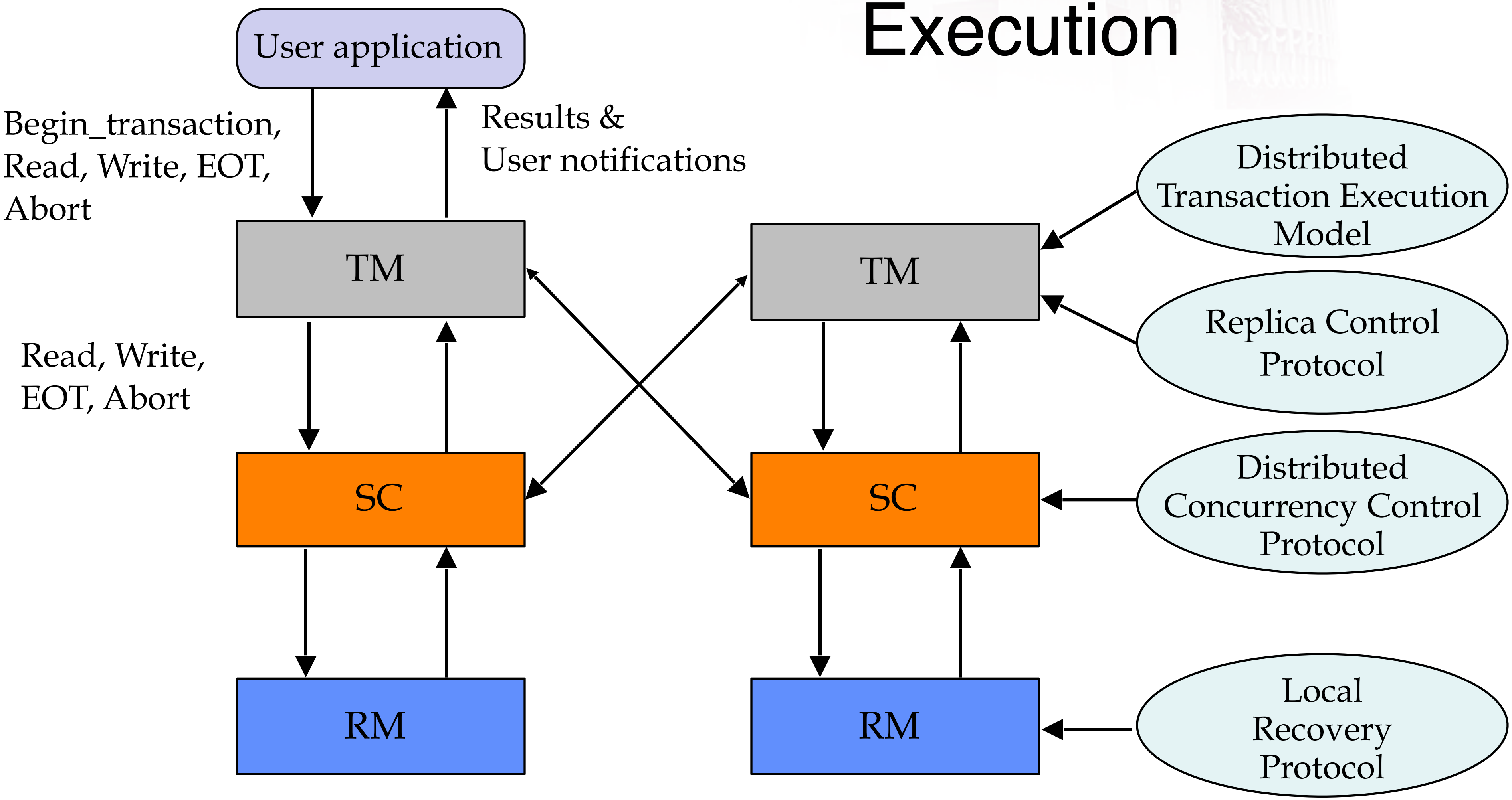




Centralized Transaction Execution



Distributed Transaction Execution



Concurrency Control

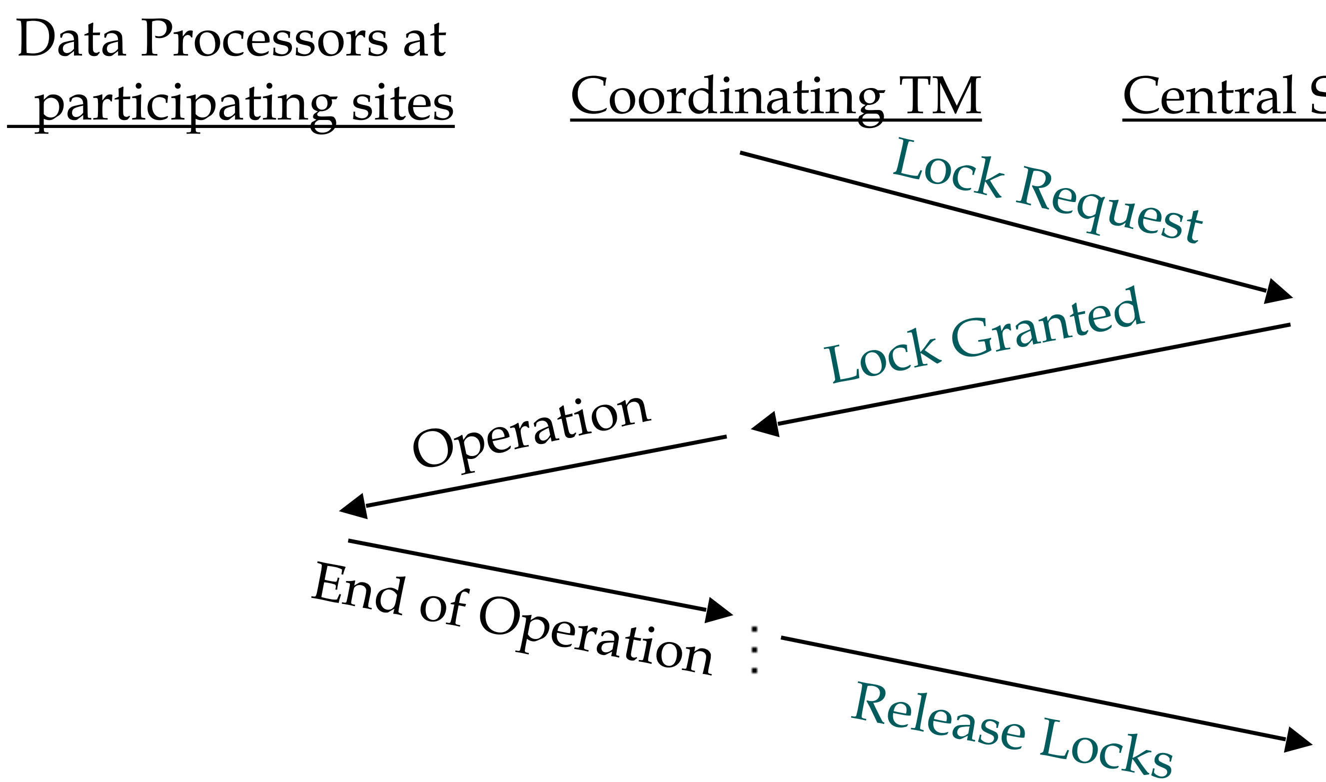
Centralized 2PL

Data Processors at participating sites | Coordinating TM | Central Site LM

- How does the protocol work?

Centralized 2PL

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



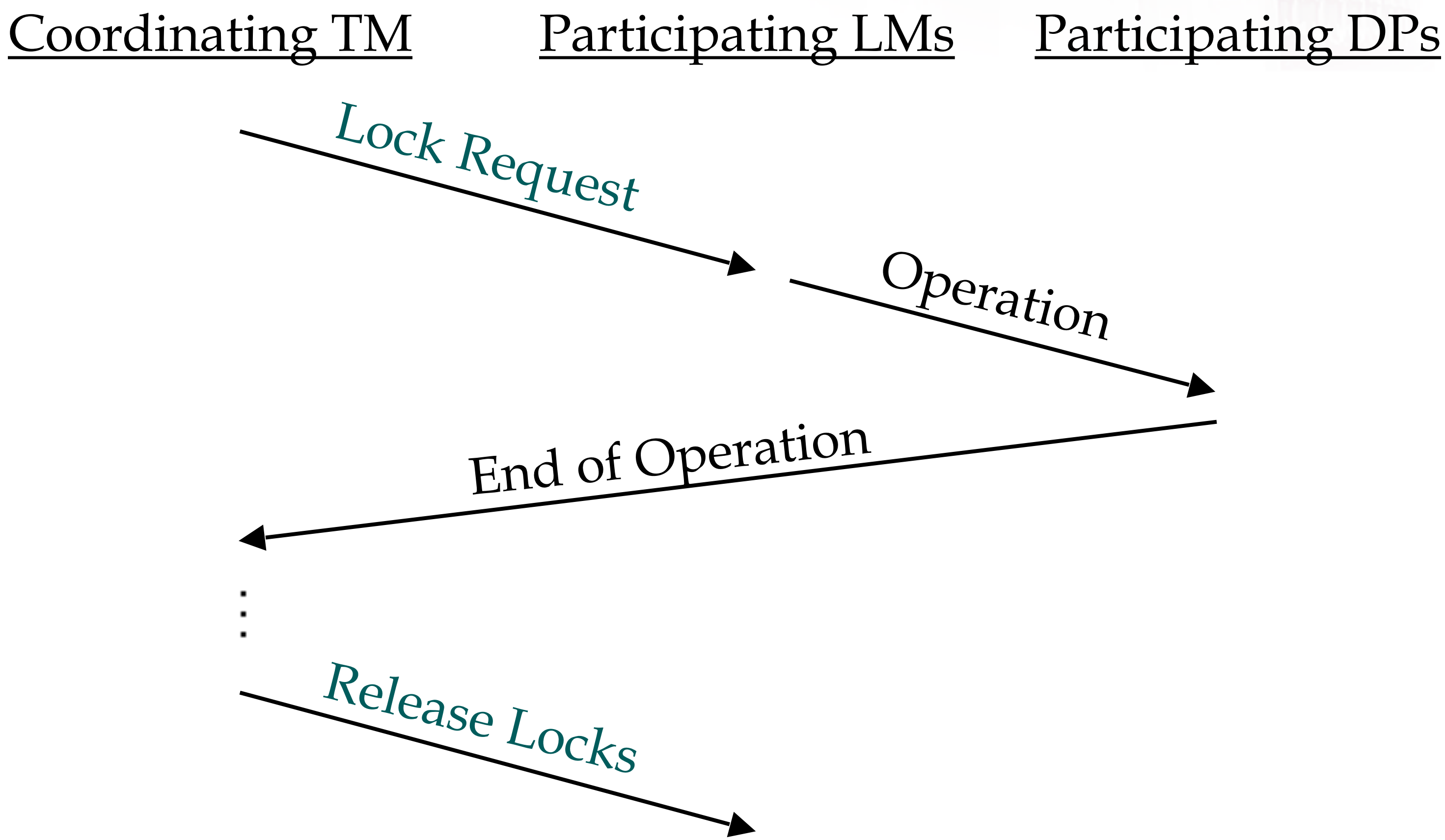
Issues?

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 (TO)

Timestamp Ordering (TO)

Each transaction is assigned a Timestamp that is Unique and Monotonic

How to achieve Serializable Schedule?

Timestamp Ordering (TO) Rule

- Given two conflicting operations:
  - O_{ij} and O_{kl}
- Belonging to Transactions:
  - T_i and T_k
    - O_{ij} is executed before O_{kl} if and only if:
    - $ts(T_i) < ts(T_k)$
    - T_i is the older transaction

Timestamp Ordering (TO)

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

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

❸ Each data item is assigned a write timestamp (*wts*) and a read timestamp (*rts*):
- $rts(x)$ = largest timestamp (of a transaction) of any read on x
- $wts(x)$ = largest timestamp (of a transaction) of any write on x

❹ Conflicting operations are resolved by timestamp order.

<u>for $R_i(x)$</u>

<u>for $W_i(x)$</u>

How to achieve Serializable Schedules ?

Timestamp Ordering (TO)

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

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

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

- $rts(x)$ = largest timestamp (of a transaction) of any read on x
- $wts(x)$ = largest timestamp (of a transaction) of any write on x

❹ Conflicting operations are resolved by timestamp order.

<u>for $R_i(x)$</u>

if $ts(T_i) < wts(x)$
then reject $R_i(x)$
else accept $R_i(x)$

$rts(x) \leftarrow ts(T_i)$

<u>for $W_i(x)$</u>

if $ts(T_i) < rts(x)$ **and** $ts(T_i) < wts(x)$
then reject $W_i(x)$
else accept $W_i(x)$

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

Timestamp Ordering (TO)

Disadvantage ?

Timestamp Ordering (TO)

Disadvantage ?

Lots of Restarts may affect the system performance

Conservative Timestamp Ordering

- Basic TO tries to execute an operation as soon as it receives it
  - progressive
  - too many restarts since there is no delaying
- Conservative time stamping 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

Conservative Timestamp Ordering

What Kind of Schedules Extreme Conservative TO lead to ?

Multiversion Timestamp Ordering

- Do not modify the values in the database, create new values.
- A $R_i(x)$ is translated into a read on one version of x.
  - Find a version of x (say x_v) such that $ts(x_v)$ is the largest timestamp less than $ts(T_i)$.
- A $W_i(x)$ is translated into $W_i(x_w)$ and accepted if the scheduler has not yet processed any $R_j(x_r)$ such that

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

Optimistic Concurrency Control Algorithms

Pessimistic execution

Validate	Read	Compute	Write

Optimistic execution

Read	Compute	Validate	Write

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

Textbook

- **Principles of Distributed Database Systems--by M. Tamer Ozsu and Patrik Valduriez**
  - Chapter 3: Fragmentation
  - Chapters 6, 7, 8: Query Processing
  - Chapters 10, 11: Concurrency Control

Concurrency Control

Deadlock Detection

- Transactions are allowed to wait freely.
- Wait-for graphs and cycles.
- Topologies for deadlock detection algorithms
  - Centralized
  - 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
- Would be a reasonable choice if the concurrency control algorithm is also centralized.

Distributed Deadlock Detection

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:
    - ❶ Since each site receives the potential deadlock cycles from other sites, these edges are added to the local WFGs
    - ❷ 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

- 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.

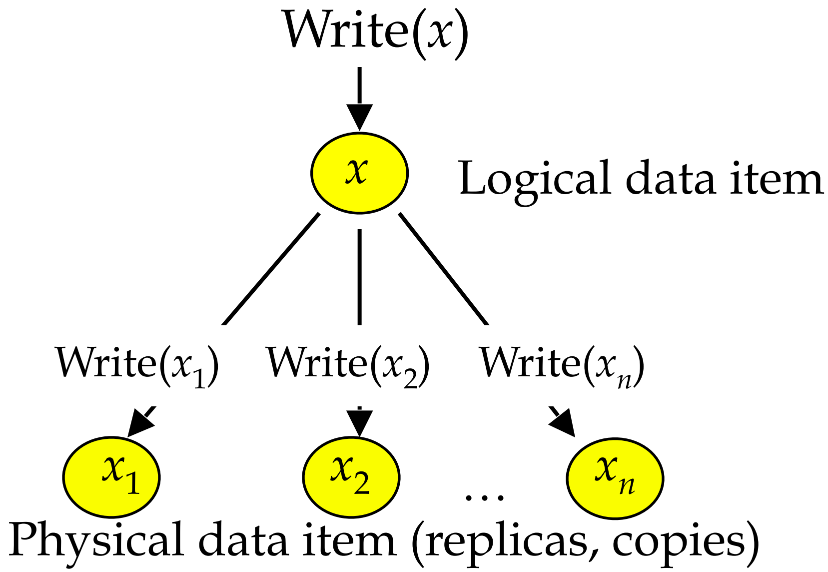
Replicated Data Management

Replication

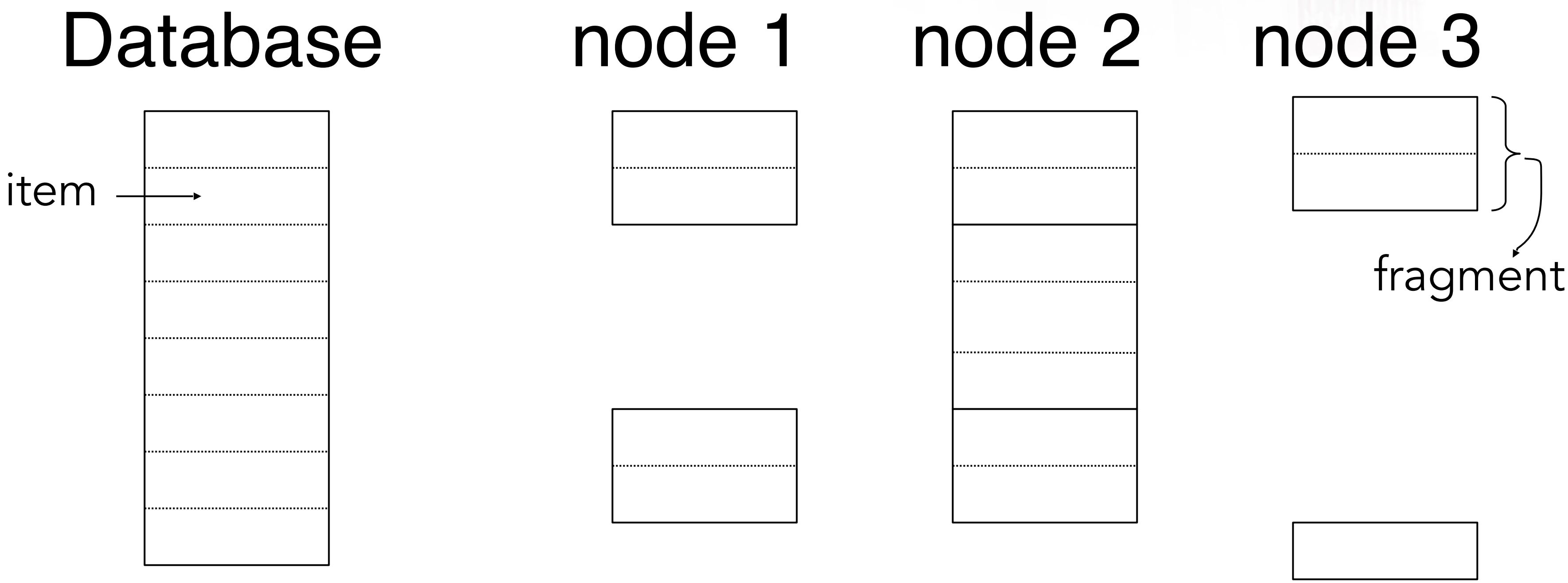
- Why replicate?
  - System availability : Avoid single points of failure
  - Performance: Localization
  - Scalability: Scalability in numbers and geographic area
  - Application requirements
- Why not replicate?
  - Consistency issues
    - Updates are costly
    - Availability may suffer if not careful

Execution Model

- There are physical copies of logical objects in the system.
- Operations are specified on logical objects, but translated to operate on physical objects.
- One-copy equivalence
  - The effect of transactions performed by clients on replicated objects should be the same as if they had been performed on a single set of objects.

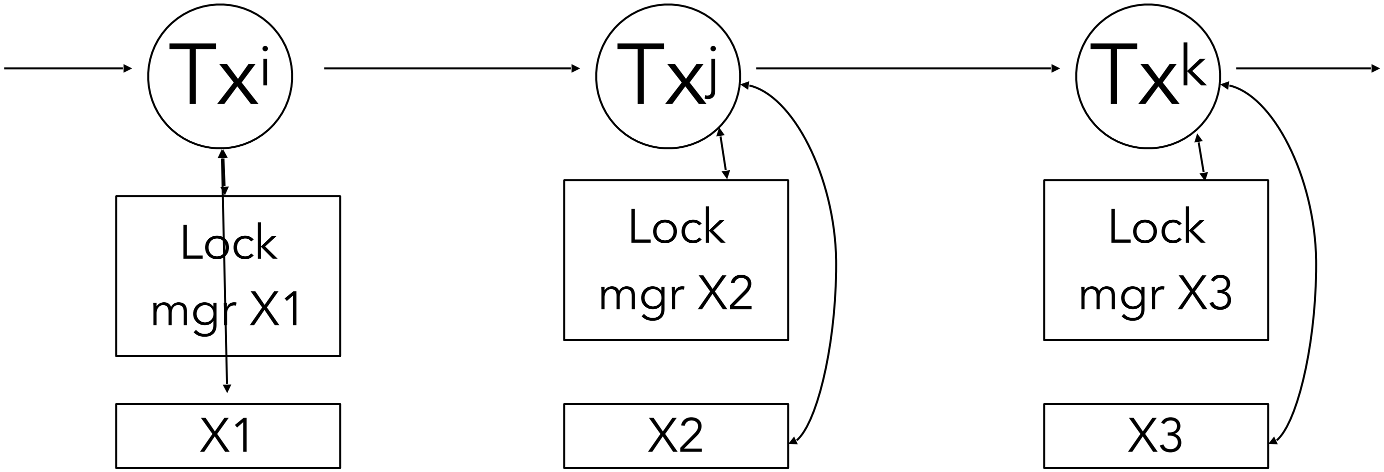


Data Replication



- Study one fragment, for time being
- Data replication ⇒ higher availability

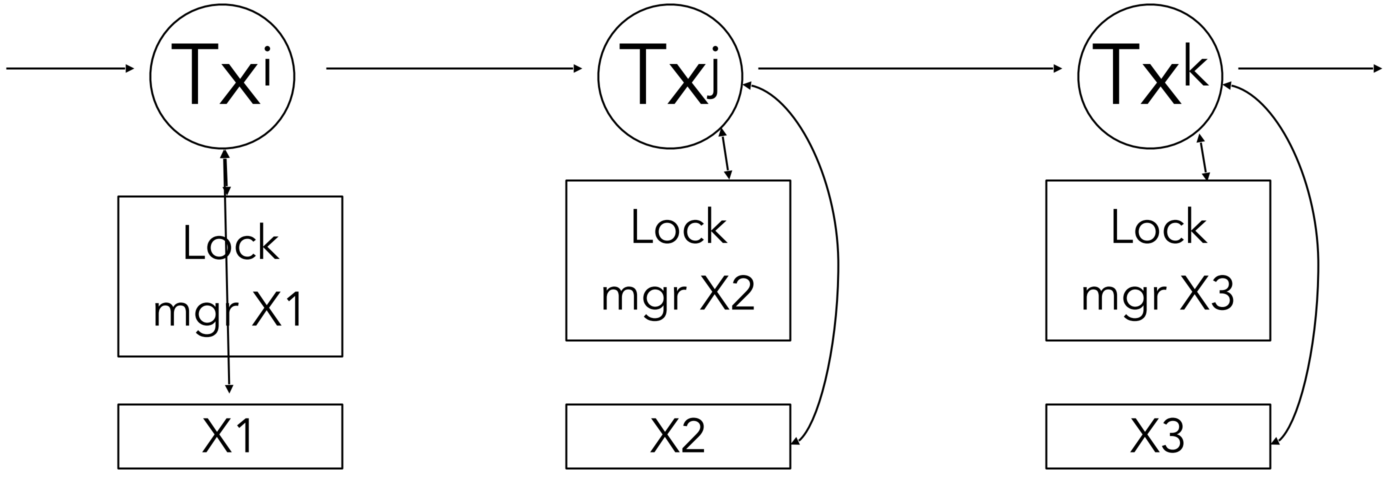
Basic Solution ?



Object X has copies X1, X2, X3

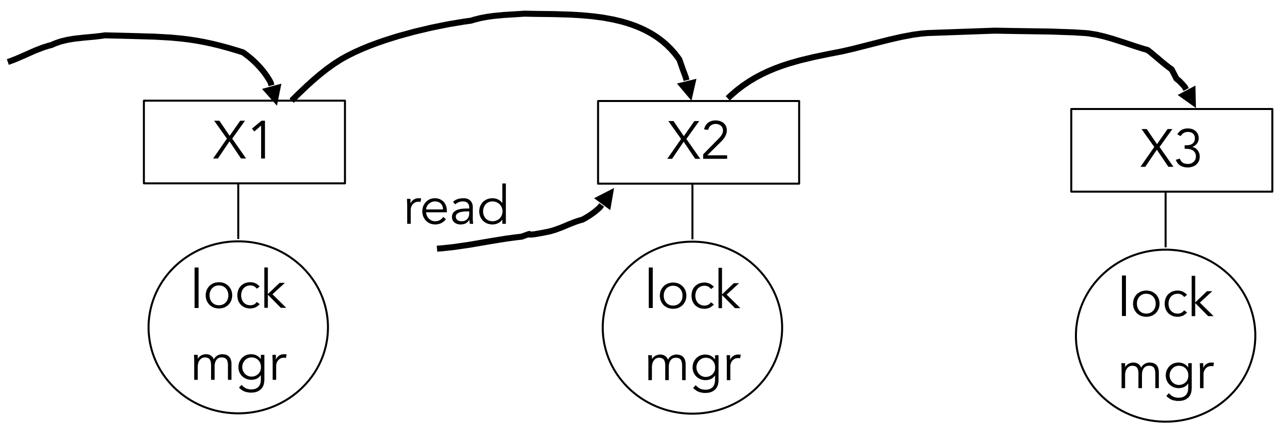
Basic Solution

- Treat each copy as an independent data item

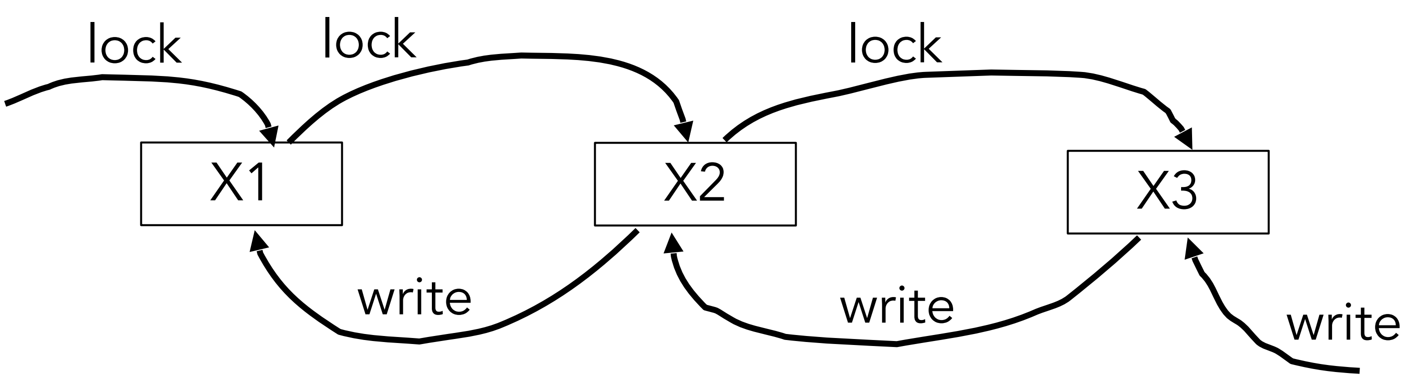


Object X has copies X1, X2, X3

- Read(X):
  - get shared X1 lock
  - get shared X2 lock
  - get shared X3 lock
  - read one of X1, X2, X3
  - at end of transaction, release X1, X2, X3 locks



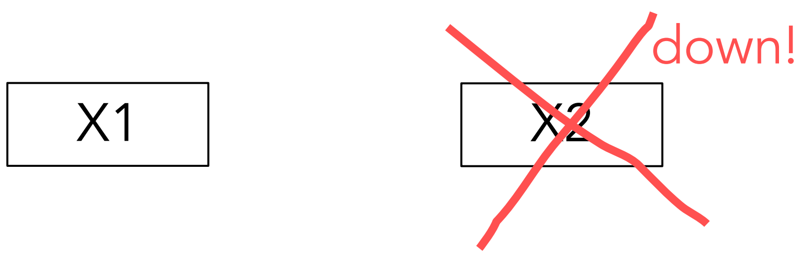
- Write(X):
  - get exclusive X1 lock
  - get exclusive X2 lock
  - get exclusive X3 lock
  - write new value into X1, X2, X3
  - at end of transaction, release X1, X2, X3 locks



Critique

- Is this Correct ?
- What are the Cons ?

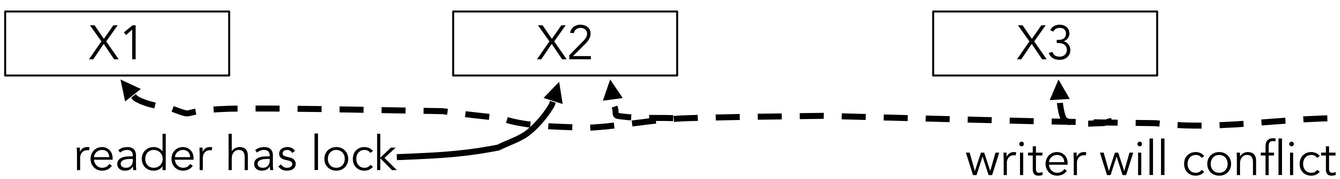
- Correctness OK
- Problem: Low availability



cannot access X!

Basic Solution — Improvement

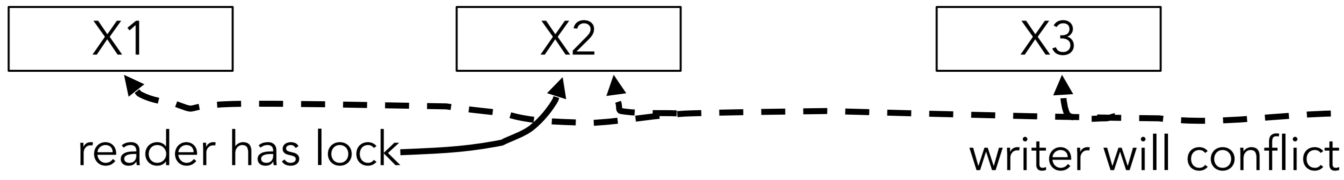
- Readers lock and access a single copy
- Writers lock all copies
  and update all copies



- Is this good for Read Availability?
- Is this good for Write Availability?

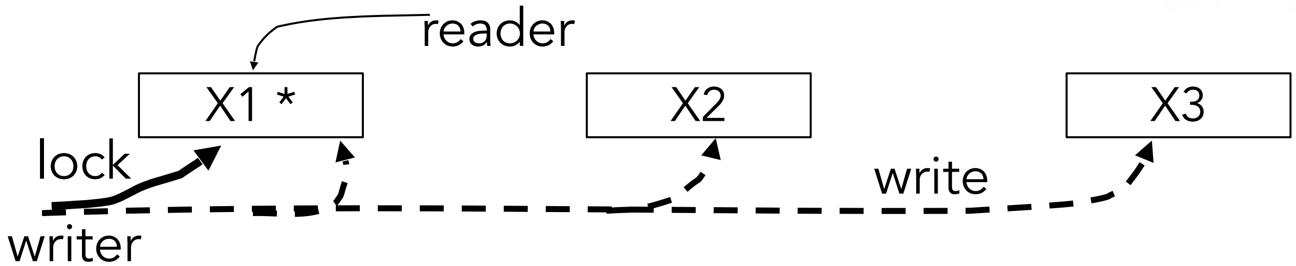
Basic Solution — Improvement

- Readers lock and access a single copy
- Writers lock all copies
  and update all copies



- Good availability for reads
- Poor availability for writes

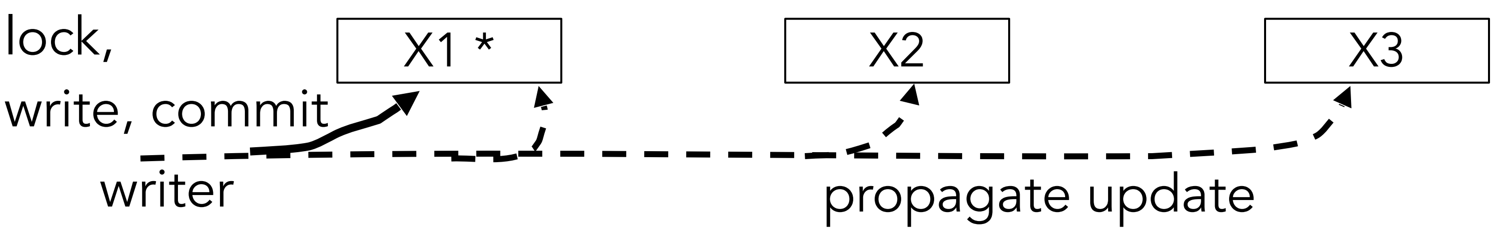
Variation on Basic: Primary copy



- Select primary site (static for now)
- Readers lock and access primary copy
- Writers lock primary copy
  and update all copies

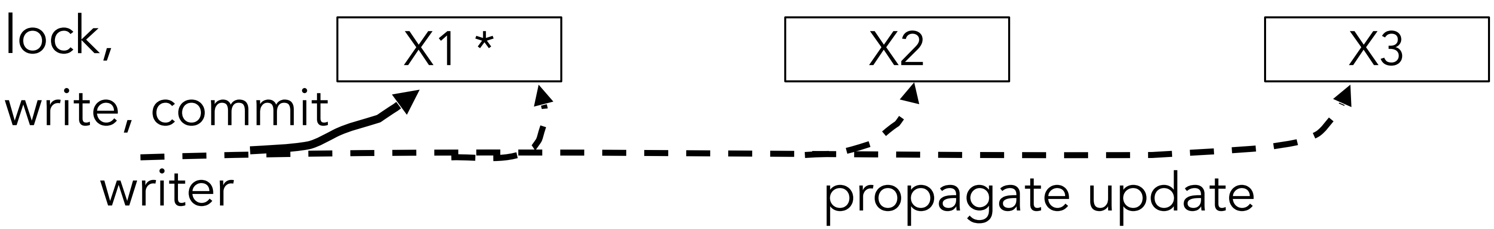
Commit Options for Primary Site Scheme

- Local Commit



Commit Options for Primary Site Scheme

- Local Commit



Write(X):

- Get exclusive X1* lock
- Write new value into X1*

⋮

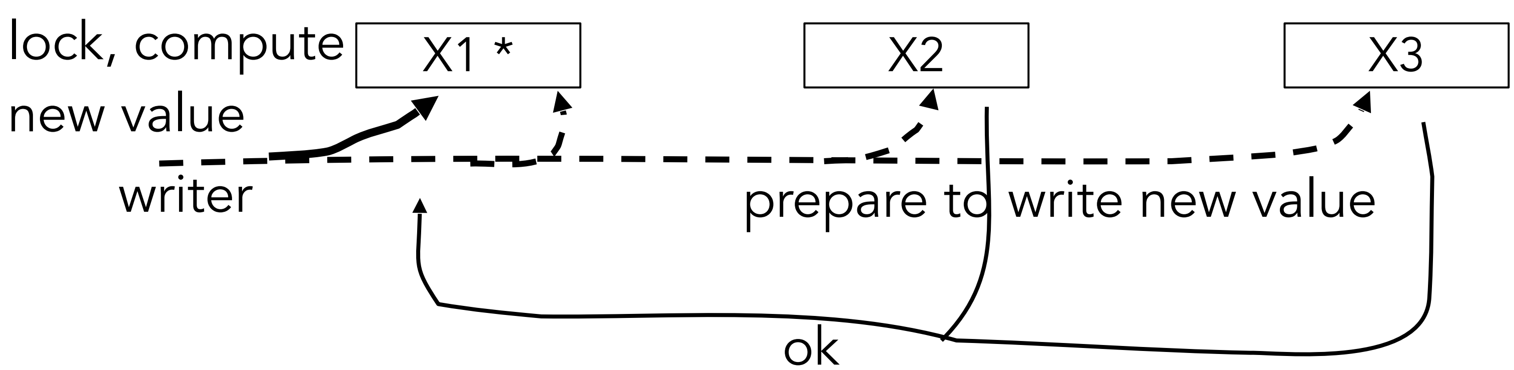
- Commit at primary; get sequence number

⋮

- Perform X2, X3 updates in sequence number order

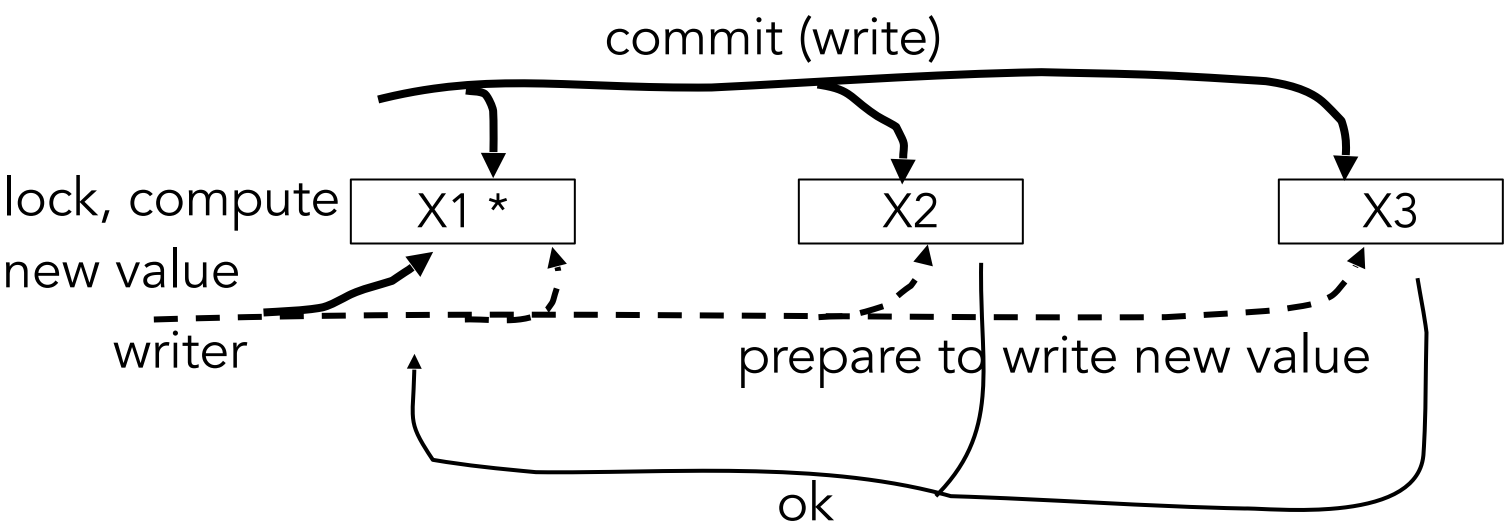
Commit Options for Primary Site Scheme

- Distributed Commit



Commit Options for Primary Site Scheme

- Distributed Commit



Replicated Data Management

Replication Issues

- Consistency models - how do we reason about the consistency of the “global execution state”?
  - Mutual consistency
  - Transactional consistency
- Where are updates allowed?
  - Centralized
  - Distributed
- Update propagation techniques – how do we propagate updates from one copy to the other copies?
  - Eager
  - Lazy

Consistency

- Mutual Consistency
  – How do we keep the values of physical copies of a logical data item synchronized?
  – Strong consistency
    - All copies are updated within the context of the update transaction
    - When the update TX completes, all copies have the same value
    - Typically achieved through 2PC
  – Weak consistency
    - Eventual consistency: the copies are not identical when update transaction completes, but they eventually converge to the same value

Transactional Consistency

- How can we guarantee that the global execution schedule over replicated data is serializable?
- One-copy serializability (1SR)
  - The effect of transactions performed by clients on replicated objects should be the same as if they had been performed *one at-a-time* on a single set of objects.
- Weaker forms are possible

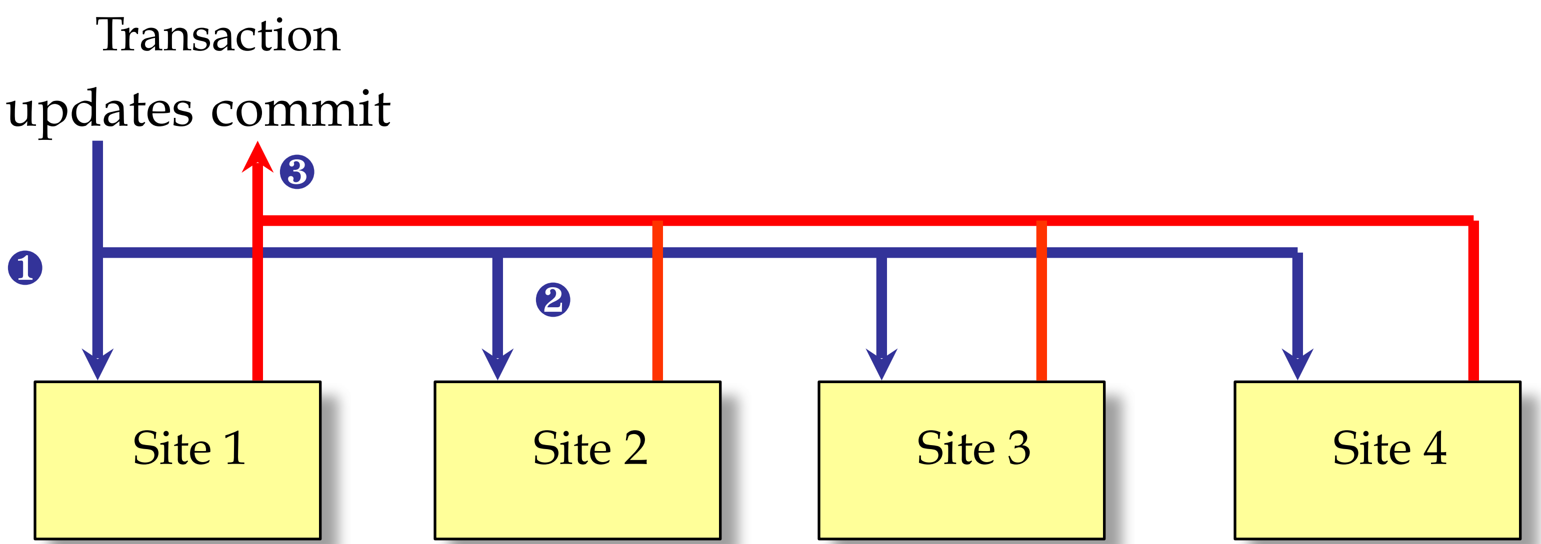
Update Management Strategies

- Depending on when the updates are propagated
  - Eager
  - Lazy
- Depending on where the updates can take place
  - Centralized
  - Distributed

	Centralized	Distributed
Eager		
Lazy		

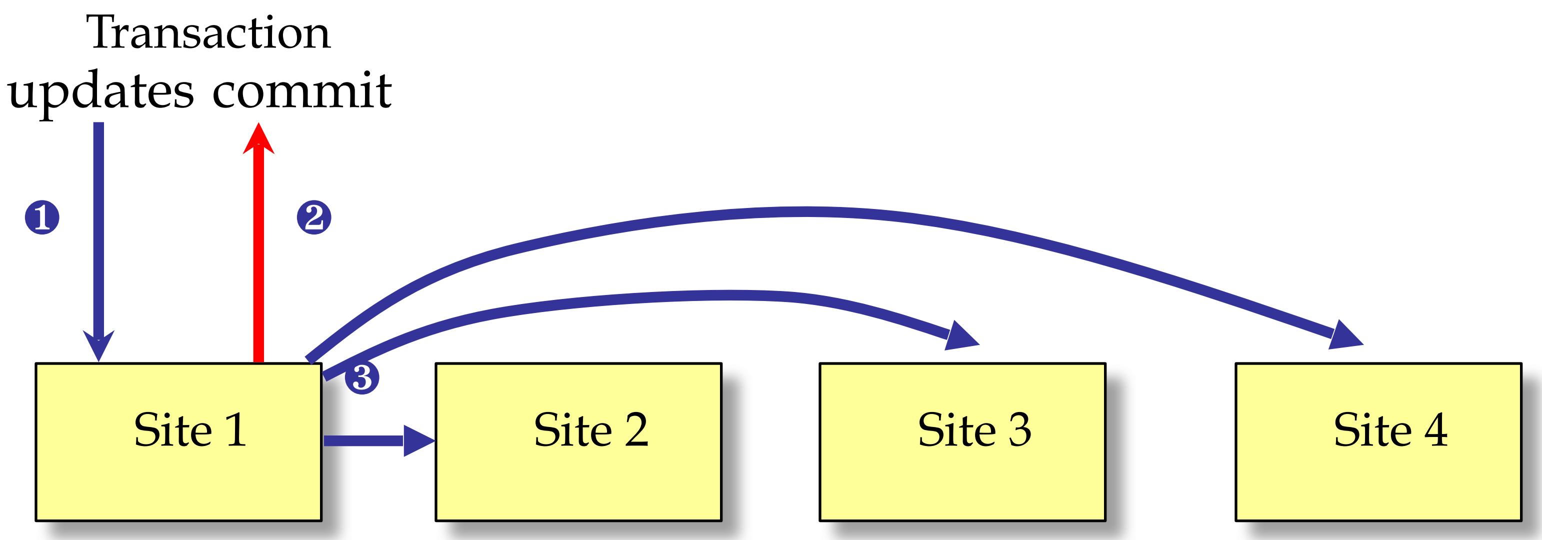
Eager Replication

- Changes are propagated within the scope of the transaction making the changes. The ACID properties apply to all copy updates.
  - Synchronous
  - Deferred
- ROWA protocol: Read-one/Write-all



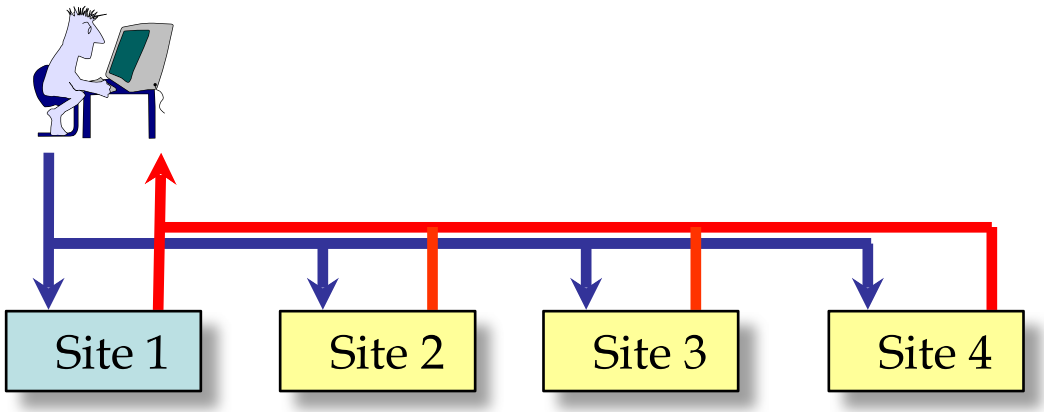
Lazy Replication

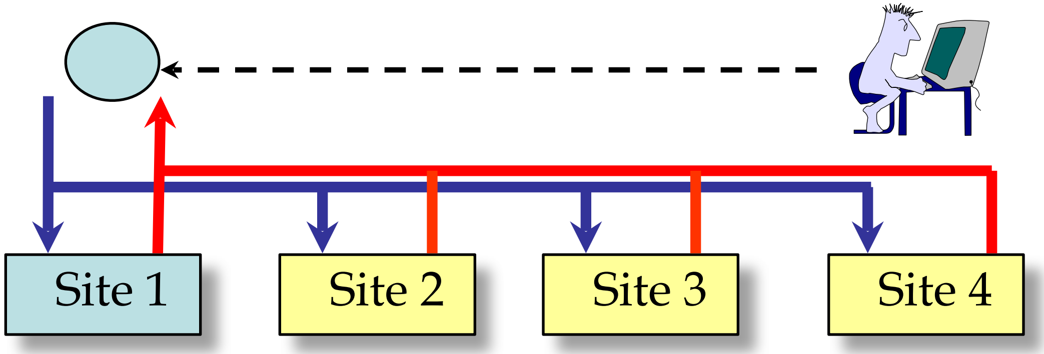
- Lazy replication first executes the updating transaction on one copy. After the transaction commits, the changes are propagated to all other copies (refresh transactions)
- While the propagation takes place, the copies are mutually inconsistent.
- The time the copies are mutually inconsistent is an adjustable parameter which is application dependent.



Centralized

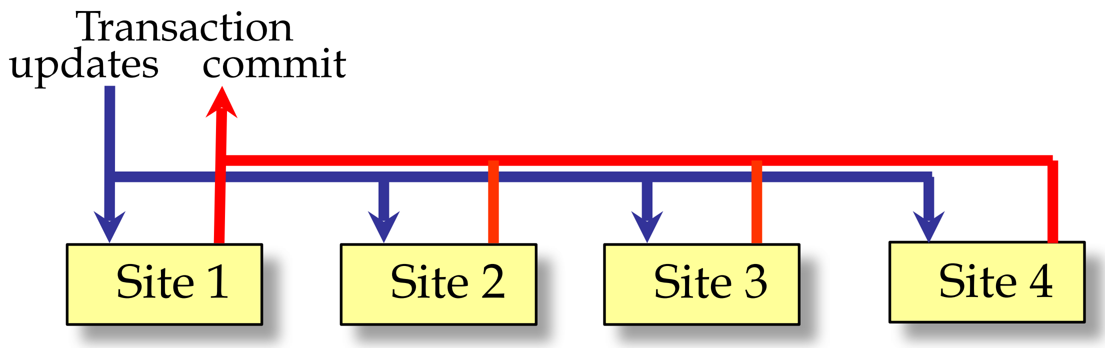
- There is only one copy which can be updated (the master), all others (slave copies) are updated reflecting the changes to the master.

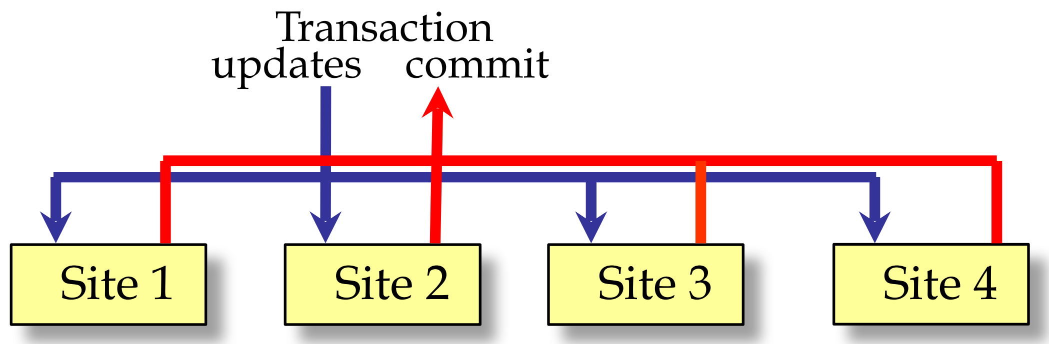




Distributed

- Changes can be initiated at any of the copies. That is, any of the sites which owns a copy can update the value of the data item.





Forms of Replication

Eager

- Pros
- Cons

Lazy

- Pros
- Cons

Centralized

- Pros
- Cons

Distributed

- Pros
- Cons

Forms of Replication

Eager

- \+ No inconsistencies (identical copies)
- \+ Reading the local copy yields the most up to date value
- \+ Changes are atomic
- – A transaction has to update all sites
  - – Longer execution time
  - – Lower availability

Lazy

- \+ A transaction is always local (good response time)
- – Data inconsistencies
- – A local read does not always return the most up-to-date value
- – Changes to all copies are not guaranteed

Centralized

- \+ No inter-site synchronization is necessary (it takes place at the master)
- \+ There is always one site which has all the updates
- – The load at the master can be high
- – Reading the local copy may not yield the most up-to-date value

Distributed

- \+ Any site can run a transaction
- \+ Load is evenly distributed
- – Copies need to be synchronized

Replication Protocols

The previous ideas can be combined into 4 different replication protocols:

	Centralized	Distributed
Eager	Eager centralized	Eager distributed
Lazy	Lazy centralized	Lazy distributed

Replication Strategies

	Centralized	Distributed
Eager	+ Updates do not need to be coordinated + No inconsistencies - Longest response time - Only useful with few updates - Local copies can only be read	+ No inconsistencies + Elegant (symmetrical solution) - Long response times - Updates need to be coordinated
Lazy	+ No coordination necessary + Short response times - Local copies are not up to date - Inconsistencies	+ No centralized coordination + Shortest response times - Inconsistencies - Updates can be lost (reconciliation)

Questions