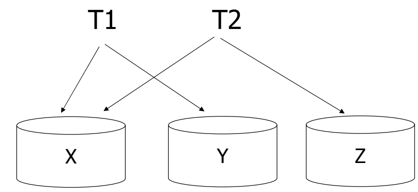
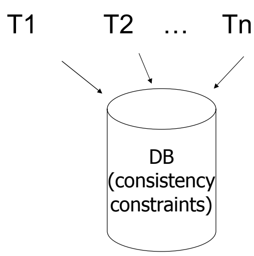
Concurrency Control in Distributed Databases

In distributed DB



In centralized (Local) DB



Example:

T1: Read(A)
$A \leftarrow A+100$
Write(A)
Read(B)
$B \leftarrow B+100$
Write(B)

T2: Read(A)
$A \leftarrow A\times 2$
Write(A)
Read(B)
$B \leftarrow B\times 2$
Write(B)

Constraint: A=B

Schedule A: Serial Schedule

T1	T2	A	B
		25	25
Read(A); $A \leftarrow A+100$			
Write(A);		125	
Read(B); $B \leftarrow B+100$;			
Write(B);			125
	Read(A);$A \leftarrow A\times 2$;		
	Write(A);	250	
	Read(B);$B \leftarrow B\times 2$;		
	Write(B);		250
		250	250

Schedule B

T1	T2	A	B
		25	25
Read(A); $A \leftarrow A+100$			
Write(A);		125	
	Read(A);$A \leftarrow A\times 2$;		
	Write(A);	250	
Read(B); $B \leftarrow B+100$;			
Write(B);			125
	Read(B);$B \leftarrow B\times 2$;		
	Write(B);		250
		250	250

Schedule C

T1	T2	A	B
		25	25
Read(A); A ← A+100 Write(A);		125	
	Read(A);A ← A×2; Write(A);	250	
	Read(B);B ← B×2; Write(B);		50
Read(B); B ← B+100; Write(B);			150
		250	150

Schedule D

Same as Schedule C but with new T2'

T1	T2'	A	B
		25	25
Read(A); A ← A+100 Write(A);		125	
	Read(A);A ← A×1; Write(A);	125	
	Read(B);B ← B×1; Write(B);		25
Read(B); B ← B+100; Write(B);			125
		125	125

What are good schedules?

- Want schedules that are "good", regardless of
  - initial state and
  - transaction semantics
- Only look at order of reads and writes

Example:

$S_B=r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$

Example:

S_B=r1(A)w1(A)r2(A)w2(A)r1(B)w1(B)r2(B)w2(B)

$T_1 \rightarrow T_2$ $T_1 \rightarrow T_2$

S_B'=$\underbrace{r_1(A)w_1(A)\ r_1(B)w_1(B)}_{T_1}\underbrace{r_2(A)w_2(A)r_2(B)w_2(B)}_{T_2}$

no cycles ⇒ S_B is "equivalent" to a serial schedule (in this case T_1,T_2)

Example (Cont)

Sc=$\underbrace{r_1(A)w_1(A)}r_2(A)w_2(A)\ r_2(B)w_2(B)\underbrace{r_1(B)w_1(B)}$

Example (Cont)

Sc=$\underbrace{r_1(A)w_1(A)}\underbrace{r_2(A)w_2(A)}\ \underbrace{r_2(B)w_2(B)}\underbrace{r_1(B)w_1(B)}$

$T_1 \rightarrow T_2$
Also, $T_2 \rightarrow T_1$

T1 T2

Sc cannot be rearranged into a serial schedule

Sc is not "equivalent" to any serial schedule

Sc is "bad"

Concepts

Transaction: sequence of $r_i(x)$, $w_i(x)$ actions

Conflicting actions: $r_1(A)$ / $w_2(A)$; $w_1(A)$ / $r_2(A)$; $w_1(A)$ / $w_2(A)$

Schedule: represents chronological order in which actions are executed

Serial schedule: no interleaving of actions or transactions

Serializable schedule: a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule

Definition

S_1, S_2 are <u>conflict equivalent</u> schedules if S_1 can be transformed into S_2 by a series of swaps on non-conflicting actions.

A schedule is <u>conflict serializable</u> if it is conflict equivalent to some serial schedule.

Precedence graph P(S) (S is schedule)

Nodes: transactions in S

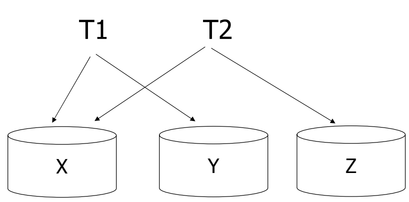
Arcs: $T_i \rightarrow T_j$ whenever

- $p_i(A)$, $q_j(A)$ are actions in S
- $p_i(A) <_S q_j(A)$
- at least one of p_i, q_j is a write

Theorem

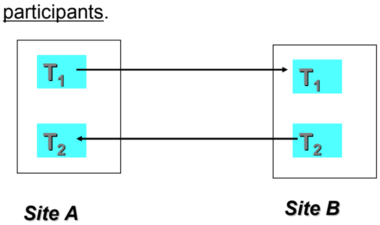
$P(S_1)$ acyclic $\Longleftrightarrow$ S_1 conflict serializable

In distributed DB



Distributed Transactions

- A distributed transaction T is initiated at one site and spawned subtransactions at several other sites. We distinguish the subtransaction at home site by calling it the <u>coordinator</u>, while the other subtransactions are the <u>participants</u>.



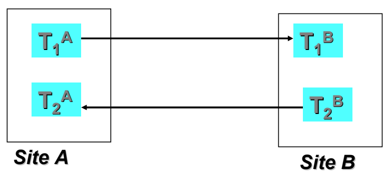
Distributed Transactions

T_1 in site A, denoted as T_1^A, is the coordinator.

T_1 in site B, denoted as T_1^B, is the participant.

T_2^B, is the coordinator. T_2^A, is the participant.

T_1^A waits for T_1^B, and T_2^B waits for T_2^A.



Example

X	Y

Node 1 Node 2

constraint: X=Y

	T_1		T_2
1	(T_1) a ← X	5	(T_2) c ← X
2	(T_1) X ← a+100	6	(T_2) X ← 2c
3	(T_1) b ← Y	7	(T_2) d ← Y
4	(T_1) Y ← b+100	8	(T_2) Y ← 2d

↓ Precedence relation

Schedule S1

Precedence: intra-transaction ↓
inter-transaction ↓

(node X)	(node Y)
1 (T_1) a ← X	
2 (T_1) X ← a+100	
5 (T_2) c ← X	3 (T_1) b ← Y
6 (T_2) X ← 2c	4 (T_1) Y ← b+100
	7 (T_2) d ← Y
	8 (T_2) Y ← 2d

If X=Y=0 initially, X=Y=200 at end (always good?)

Serializability in Distributed DBMS

- Somewhat more involved. Two types of schedules have to be considered:
  - local schedules
  - global schedule
- For global transactions (i.e., global schedule) to be serializable, two conditions are necessary:
  - Each local schedule should be serializable.
  - All sub-transactions of global transactions appear in the same order in the equivalent serial schedule at ALL sites.

Global Non-serializability

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

T_1:	Read(x)	T_2:	Read(x)
	$x \leftarrow x+5$		$x \leftarrow x*10$
	Write(x)		Write(x)
	Commit		Commit

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

$LH_1=\{R_1(x),W_1(x),R_2(x),W_2(x)\}$ $T_1 \rightarrow T_2$

$LH_2=\{R_2(x),W_2(x),R_1(x),W_1(x)\}$ $T_2 \rightarrow T_1$

Global Non-serializability (Cont)

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

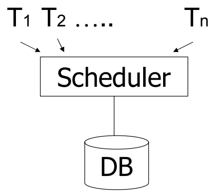
$LH_2=\{R_2(x),W_2(x),R_1(x),W_1(x)\}$

Assume x=1 initially. At site 1, x=60 after LH1.

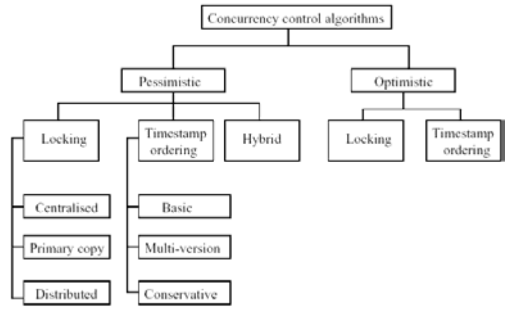
At site 2, x=15 after LH2. Violates the mutual consistency.

How to enforce serializable schedules?

prevent P(S) cycles from occurring

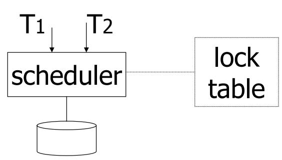


Classification of Concurrency control Mechanisms



A locking protocol

Two new actions:
lock (exclusive):
unlock:



Locking Rules in centralized db (2-phase locking)

- Well-formed transactions
- Legal schedulers
- Two-phase transactions

- These rules guarantee serializable schedules

Rules

Rule #1: Well-formed transactions

Ti: … li(A) … pi(A) … ui(A) ...

Rule #2: Legal scheduler

S = …….. li(A) ……….. ui(A) ……...

no lj(A)

Exercise:

- **What schedules are legal?**
  What transactions are well-formed?

  S1 = $l_1(A)l_1(B)r_1(A)w_1(B)l_2(B)u_1(A)u_1(B)$ $r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$

  S2 = $l_1(A)r_1(A)w_1(B)u_1(A)u_1(B)$ $l_2(B)r_2(B)w_2(B)l_3(B)r_3(B)u_3(B)$

  S3 = $l_1(A)r_1(A)u_1(A)l_1(B)w_1(B)u_1(B)$ $l_2(B)r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$

Exercise:

- **What schedules are legal?**
  What transactions are well-formed?

  S1 = $l_1(A)l_1(B)r_1(A)w_1(B)\mathbf{l_2(B)}u_1(A)u_1(B)$
  $r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$

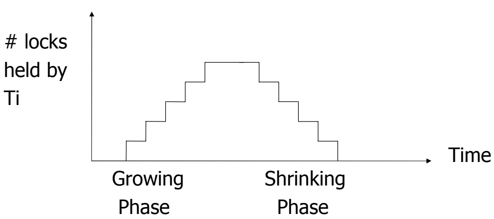
  S2 = $l_1(A)r_1(A)\mathbf{w_1(B)}u_1(A)u_1(B)$
  $l_2(B)r_2(B)w_2(B)l_3(B)r_3(B)u_3(B)$

  S3 = $l_1(A)r_1(A)u_1(A)l_1(B)w_1(B)u_1(B)$
  $l_2(B)r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$

Schedule F

T1	T2	A	B
		25	25
$l_1(A)$;Read(A)			
A←A+100;Write(A);$u_1(A)$		125	
	$l_2(A)$;Read(A)		
	A ← Ax2;Write(A);$u_2(A)$	250	
	$l_2(B)$;Read(B)		
	B ← Bx2;Write(B);$u_2(B)$		50
$l_1(B)$;Read(B)			
B ← B+100;Write(B);$u_1(B)$			150
		250	150

Two phase locking (2PL) for transactions



Schedule G

T1	T2
$l_1(A)$;Read(A)	
A ← A+100;Write(A)	
$l_1(B)$; $u_1(A)$	
	$l_2(A)$;Read(A)
	A ← Ax2;Write(A); $l_2(B)$ delayed
Read(B);B ← B+100	
Write(B); $u_1(B)$	
	$l_2(B)$; $u_2(A)$;Read(B)
	B ← Bx2;Write(B);$u_2(B)$;

Schedule H (T2 reversed)

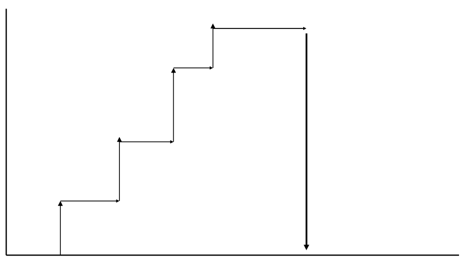
T1	T2
$l_1(A)$; Read(A)	$l_1(B)$;Read(B)
A ← A+100;Write(A)	B ← Bx2;Write(B)
$l_2(B)$ delayed	$l_2(A)$ delayed

Deadlocked transactions are rolled back

Theorem

2PL $\Rightarrow$ conflict serializable schedule

Strict 2PL



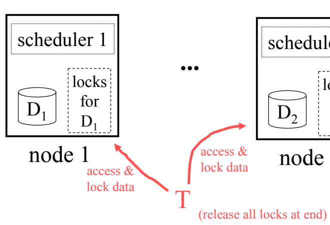
Locking-Based Algorithms

- That's all that is needed to ensure serializability!
- Beyond this is to improve concurrency
  - Locks are either read lock (also called *shared lock*) or write lock (also called *exclusive lock*)
  - Read locks and write locks are *conflicting* (or incompatible)

	rlock	*wlock*
rlock	*yes*	*no*
wlock	*no*	*no*

Locking in distributed DB

- **Just like in a centralized system**
- **But with multiple lock managers**



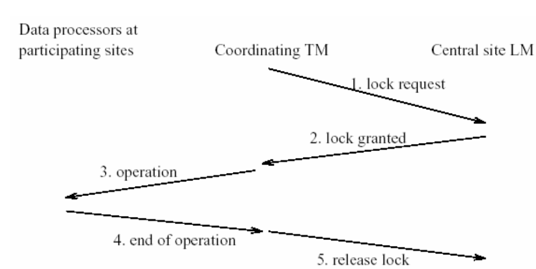
Distributed Locking

- **Look at three schemes:**
  - **Centralized locking**
  - **Primary Copy 2PL**
  - **Distributed 2PL**

Centralized Locking

- Single site that maintains all locking information.
- One lock manager for whole of DDBMS.
- Local transaction managers involved in global transaction request and release locks from lock manager.
- Or transaction coordinator can make all locking requests on behalf of local transaction managers.

Communication Structure of Centralised 2PL



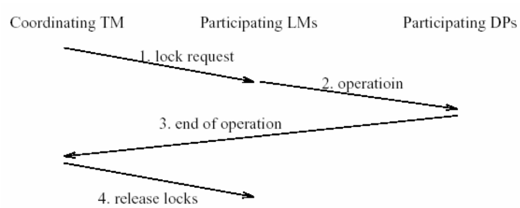
Primary Copy 2PL

- Lock managers distributed to a number of sites.
- Each lock manager responsible for managing locks for set of data items.
- For replicated data item, one copy is chosen as *primary copy*, others are *slave copies*
- Only need to write-lock primary copy of data item that is to be updated.
- Once primary copy has been updated, change can be propagated to slaves.

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*.

Communication Structure of Distributed 2PL



Dealing with Multiple Copies of Data

Three schemes which guarantee that conflicts are discovered at least in one site.

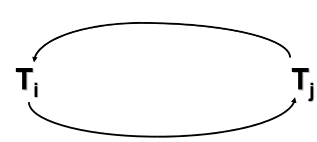
1. Read-lock-one, write-lock-all (ROWA). To read a data item A, a transaction may obtain a read-lock on any copy of A. To write on a data item A, a transaction must obtain write-locks on all copies of A.

Dealing with Multiple Copies of Data

2. The majority locking strategy. To read a data item A, a transaction may obtain read-locks on a majority of the copies of A. To write on a data item A, a transaction must also obtain write-locks on a majority of the copies of A.
3. Primary Copy Locking. All locks for a data item A are requested at the site of the primary copy.

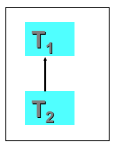
Deadlock and Wait for Graph

- A transaction is deadlocked if it is blocked and will remain blocked until there is intervention.
- Locking-based CC algorithms may cause deadlocks.
- Wait-for graph (WFG).

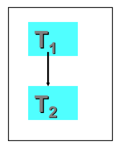


Local versus Global WFG

- Transaction T1 is initiated at site A and spawned subtransactions at site B. Transaction T2 is initiated at site B and spawned subtransactions at site A.
- Assume T_1 waits for a lock held by T_2 at site B, and T_2 waits for a lock held by T_1 at site A.



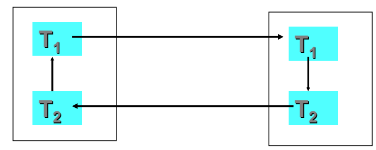
Site A (no cycle locally)



Site B (No cycle)

Local versus Global WFG

- Transaction T1 is initiated at site A and spawned subtransactions at site B. Transaction T2 is initiated at site B and spawned subtransactions at site A.
- Assume T_1 waits for a lock held by T_2 at site B, and T_2 waits for a lock held by T_1 at site A.



Site A ***Site B***

Deadlock Management

- Ignore
  - use *timeout*.
- Avoidance
  - detecting potential deadlocks in advance and taking action to ensure that deadlock will not occur.
- Detection and Recovery
  - Allowing deadlocks to form and then finding and breaking them.

Deadlock Detection

- Topologies for deadlock detection algorithms
  - Centralized
  - Hierarchical
  - Distributed

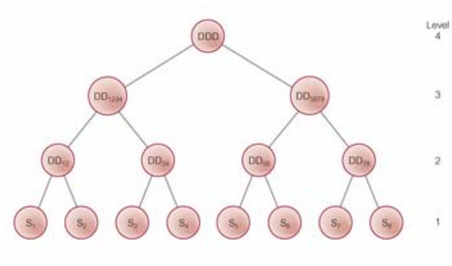
Centralized Deadlock Detection

- Single site appointed deadlock detection coordinator (DDC).
- Each scheduler periodically sends its local WFG to the central site
- DDC has responsibility of constructing and maintaining GWFG.
- If one or more cycles exist, DDC must break each cycle by selecting transactions to be rolled back and restarted.

Hierarchical Deadlock Detection

- Sites are organized into a hierarchy.
- Each site sends its LWFG to detection site above it in hierarchy.
- Reduces dependence on centralized detection site.

Hierarchical Deadlock Detection

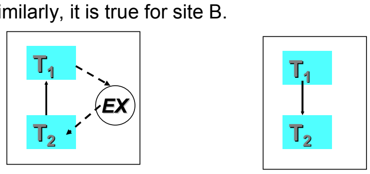


Distributed Deadlock Detection

- Sites cooperate in detection of deadlocks.
  - The local WFGs are formed at each site and passed on to other sites.
  - The WFGs are combined into a GWFGs.
  - To save cost, only when a potential deadlock exists then will the WFG be transmitted.

Local versus Global WFG

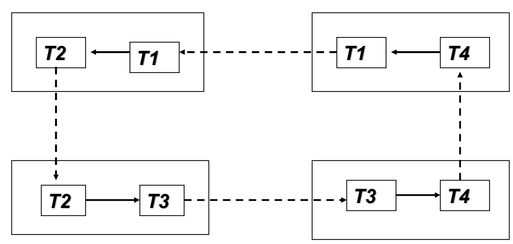
- There is a potential global deadlock found in site A, because the WFG in site A has a cycle involving the external node EX.
- Similarly, it is true for site B.



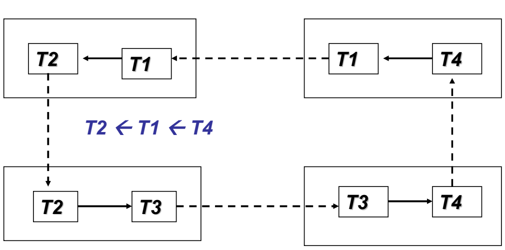
Site A ***Site B***

Distributed Deadlock Detection

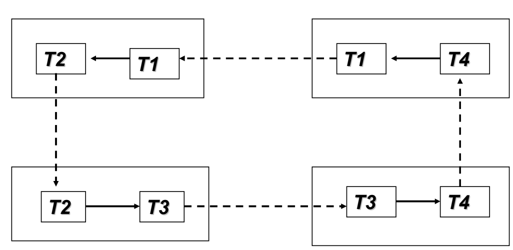
All potential global deadlock graphs discovered at site i are <u>forwarded</u> to the site j where there is a transaction T for which site i is waiting.



Distributed Deadlock Detection

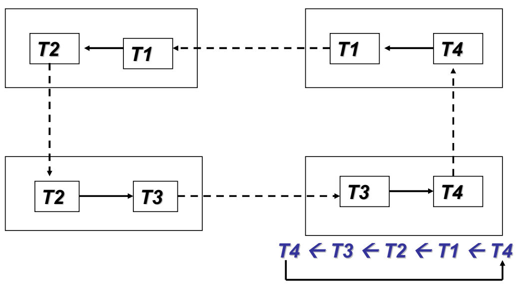


Distributed Deadlock Detection



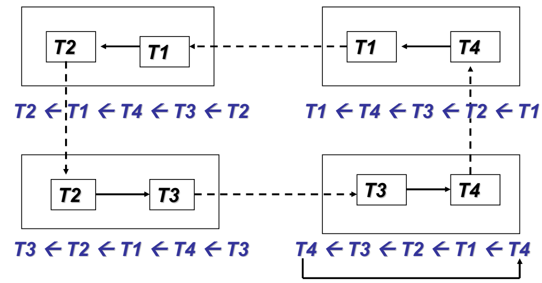
T3 ← T2 ← T1 ← T4

Distributed Deadlock Detection



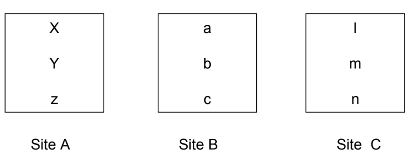
Distributed Deadlock Detection

Q: How to avoid the same deadlock being detected more than once?



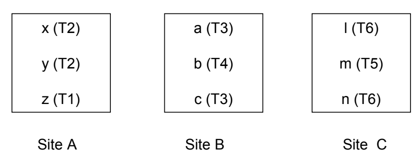
Example

- r6(l), r2(y), r5(m), r1(z), r1(x), r4(b), r3(a), r6(n), r3(c), r5(a),r1(y), r3(x), r6(m), r2(b), r4(n)



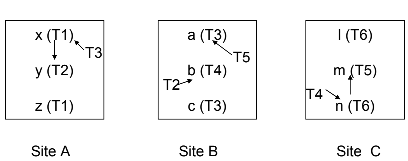
Example

- r6(l), r2(y), r5(m), r1(z), r1(x), r4(b), r3(a), r6(n), r3(c), r5(a),r1(y), r3(x), r6(m), r2(b), r4(n)



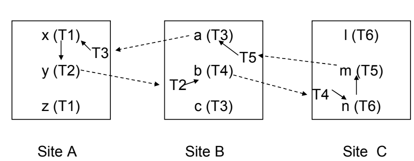
Example

- r6(l), r2(y), r5(m), r1(z), r1(x), r4(b), r3(a), r6(n), r3(c), r5(a),r1(y), r3(x), r6(m), r2(b), r4(n)



Example

- r6(l), r2(y), r5(m), r1(z), r1(x), r4(b), r3(a), r6(n), r3(c), r5(a),r1(y), r3(x), r6(m), r2(b), r4(n)



Size of Data Items

The ***size*** or ***granularity*** of the data item that can be locked in a single operation has effect on the performance of the concurrency control method.

Granule size can be:

Tuple –

Page (or bucket) –

Relation –

Size of Data Items (Cont)

Consider a transaction which is simply updating a tuple of a relation:

Tuple – the CC locks only that tuple

Relation – the CC locks the whole relation

Consider a transaction which is updating many tuples of a relation:

Tuple – the CC locks each individual tuple separately

Relation – the CC locks the entire relation

Size of Data Items (Cont)

Ideally, the CC should support mixed granularity with tuple, page and relation level locking.

DBMS will automatically upgrade locks from tuple to page to relation if a particular transaction is locking more than a certain percentage of the tuples or pages in the relation.

Summary

- Data sharing has to be managed to present inconsistencies or other anomalies
- Locking is the most widely used mechanism
- Deadlock has to be managed