21121350
Database System

Lecture 13: Concurrency Control

Lu Chen (陈璐)
College of Computer Science
Zhejiang University
Spring & Summer 2023
lucheng@zju.edu.cn/18868818726

Outline

- Lock-Based Protocols
- *Timestamp-Based Protocols
- *Validation-Based Protocols
- Multiple Granularity
- *Multiversion Schemes
- Deadlock Handling
- *Insert and Delete Operations
- *Concurrency in Index Structures

Lock-Based Protocols

- Serializable schedule is the basis of the concurrent control.

- Data items can be locked in two modes:
  1. Exclusive (X) mode: Data item can be both read as well as written. X-lock is requested using lock-X instruction.
  2. Shared (S) mode: Data item can only be read. S-lock is requested using lock-S instruction.

- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-Based Protocols (Cont.)

- Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.
- Any number of transactions can hold shared locks on an item,
  - But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

Schedule With Lock Grants

- Grants omitted in rest of chapter
  - Assume grant happens just before the next instruction following lock request
- This schedule is not serializable (why?)
- A locking protocol is a set of rules followed by all transactions while requesting and releasing locks.
- Locking protocols enforce serializability by restricting the set of possible schedules.

T_1	T_2	concurrency-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

Pitfalls of Lock-Based Protocols

- Consider the partial schedule

T_3	T_4
lock-x (B)	
read (B)	
$B := B - 50$	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

- Neither T_3 nor T_4 can make progress — executing lock-S(B) causes T_4 to wait for T_3 to release its lock on B, while executing lock-X(A) causes T_3 to wait for T_4 to release its lock on A.
- Such a situation is called a deadlock.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

Pitfalls of Lock-Based Protocols (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.

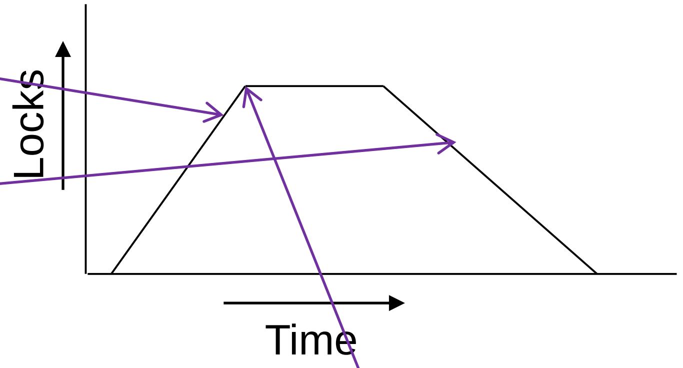
- Starvation is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.

- Concurrency control manager can be designed to prevent starvation.

The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.

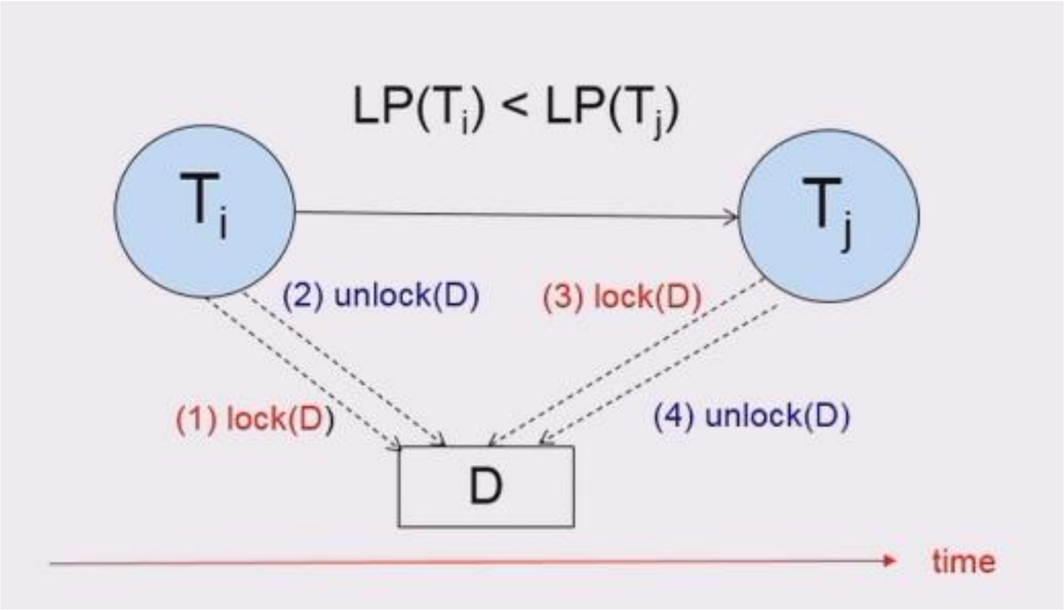
- Phase 1: Growing Phase
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: Shrinking Phase
  - Transaction may release locks
  - Transaction may not obtain locks



- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their lock points (i.e., the point where a transaction acquired its final lock).

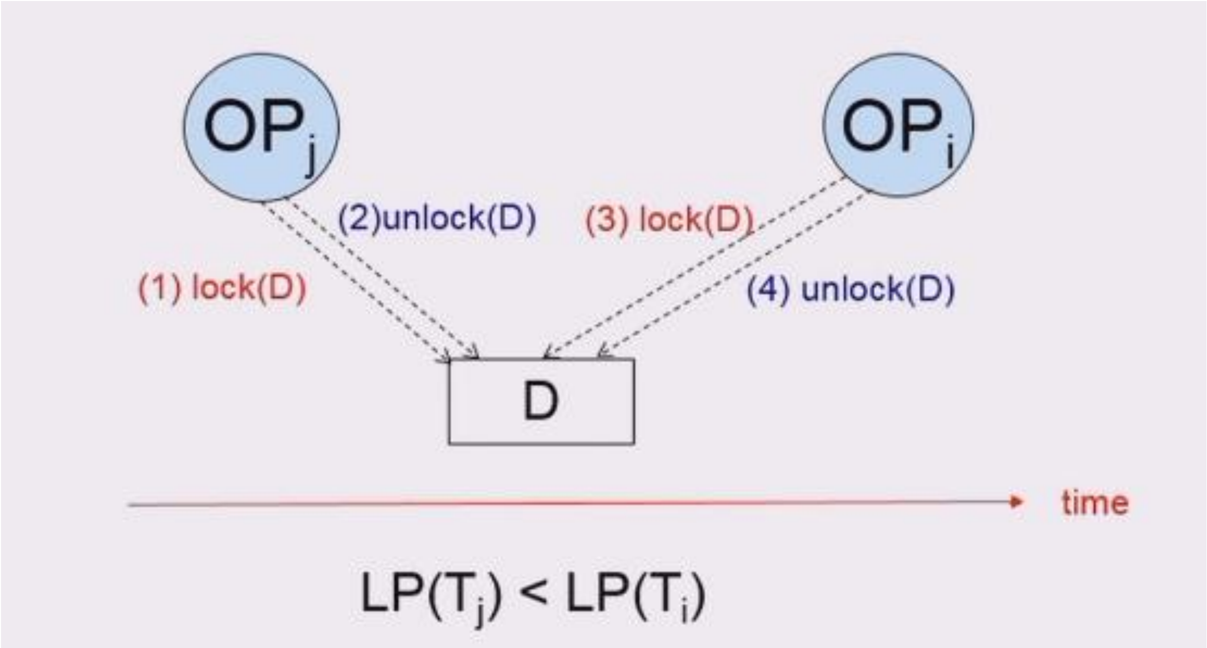
The Two-Phase Locking Protocol

- In the precedence graph corresponding to a schedule of a set of transactions T_1, T_2, … T_n, if there is an arc from T_i to T_j, then LP(T_i) < LP(T_j). (LP: Lock Point)



The Two-Phase Locking Protocol

- Let T_i is the transaction with minimum lock point (LP).
  If there is an operation OP_j of another transaction T_j that blocks an operation OP_i of T_i



The Two-Phase Locking Protocol (Cont.)

- Two-phase locking does not ensure freedom from deadlocks.
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called strict two-phase locking (严格两阶段封锁). Here a transaction must hold all its exclusive locks till it commits/aborts.
- Rigorous two-phase locking (强两阶段封锁) is even stricter: here all locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

The Two-Phase Locking Protocol (Cont.)

- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.

- However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:

  Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable.

Example

T1	T2	T3
Lock-X (B)		
read B		
	Lock-X (A)	
	read A	
	write A	
	UL (A)	
Lock-X (A)		
read A		
write B		
UL (B)		
		Lock-X (B)
		read B
write A		
UL (A)		
		write B
		UL (B)

1) Draw the precedence graph for the schedule.
2) Is it conflict serializable ?
3) Is it possible that the schedule is generated by the 2PL protocol ? Explain.

1) $T2 \xrightarrow{A} T1 \xrightarrow{B} T3$

2) < T2, T1, T3 >

3) Yes.

Exercise

Following is a schedule for transactions T1,T2,T3, and T4:

T1	T2	T3	T4
		write B	
			write C
			write A
write C			
	write C		
		write A	
write B			
write A			

a) Draw the precedence graph of the schedule
b) If the schedule is conflict serializable?
c) Can the schedule be generated by 2PL protocol?

Lock Conversions

- Two-phase locking with lock conversions:
  - First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S (unlock)
    - can release a lock-X (unlock)
    - can convert a lock-X to a lock-S (downgrade)
- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.

Automatic Acquisition of Locks

- A transaction T_i issues the standard read/write instruction, without explicit locking calls.
- The operation **read**(D) is processed as:

if T_i has a lock on D **then**
    read(D)
else begin
        if necessary wait until no other transaction has a **lock-X** on D **then**
            grant T_i a **lock-S** on D;
            read(D)
    end

Automatic Acquisition of Locks (Cont.)

- **write**(*D*) is processed as:

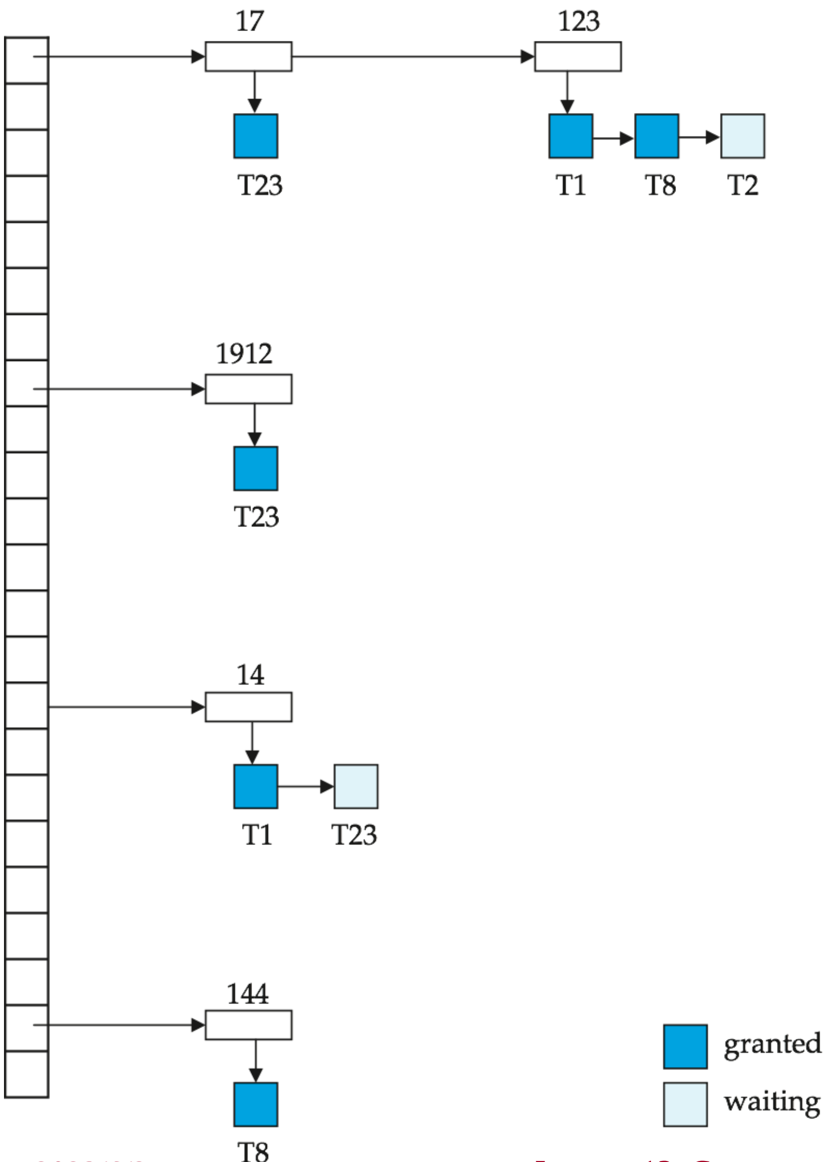
  if T_i has a **lock-X** on *D* **then**
  write(*D*)
  else begin
  if necessary wait until no other trans. has any lock on *D*,
  if T_i has a **lock-S** on *D* **then**
  upgrade lock on *D* to **lock-X**
  else
  grant T_i a **lock-X** on *D*
  write(*D*)
  end;

- All locks are released after commit or abort

Implementation of Locking

- A lock manager can be implemented as a separate process to which transactions send lock and unlock requests.
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock).
- The requesting transaction waits until its request is answered.
- The lock manager maintains a data-structure called a lock table to record granted locks and pending requests.
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked.

Lock Table



- Black rectangles indicate granted locks, white ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager may keep a list of locks held by each transaction, to implement this efficiently

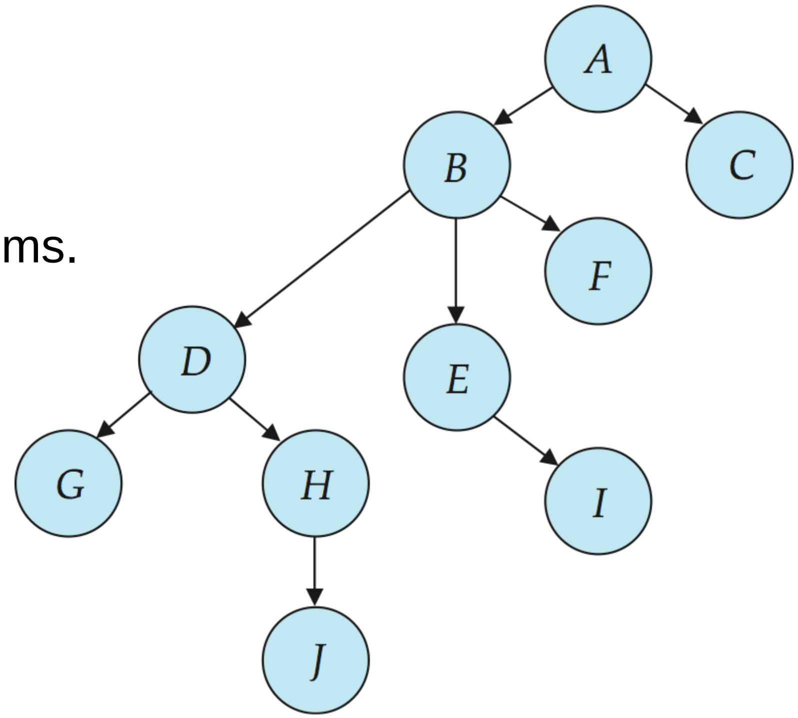
Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking.
- Impose a partial ordering (偏序) $\rightarrow$ on the set **D** = $\{d_1, d_2, \ldots, d_h\}$ of all data items.
  - If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j.
  - Implies that the set **D** may now be viewed as a directed acyclic graph, called a *database graph*.
- The tree-protocol is a simple kind of graph protocol.

Tree Protocol

A,B,C,D,E,F,G,H,I,J are data items.

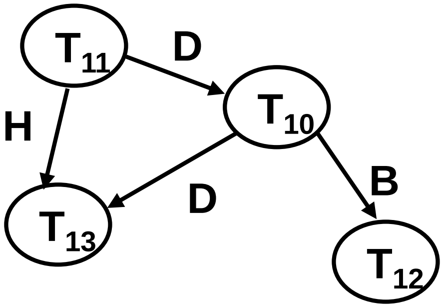
Every transaction Ti must conform the locking order.



1. Only exclusive locks are allowed.
2. The first lock by T_i may be on any data item. Subsequently, a data Q can be locked by T_i only if the parent of Q is currently locked by T_i.
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i .

Serializable Schedule Under the Tree Protocol

T_{10}	T_{11}	T_{12}	T_{13}
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock(D)		
lock-X(E)			
lock-X(D)			
unlock(B)			
unlock(E)			
		lock-X(B)	
		lock-X(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock (G)			

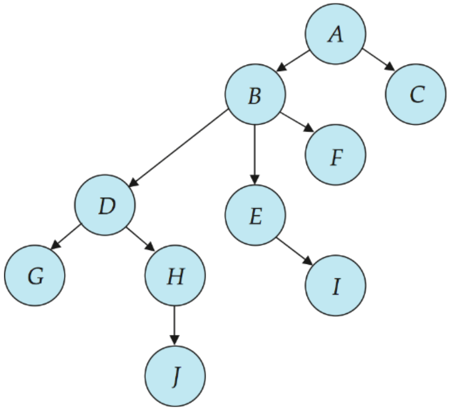


Precedence graph

Serializable Schedules:

$T_{11} \rightarrow T_{10} \rightarrow T_{12} \rightarrow T_{13}$,

或 $T_{11} \rightarrow T_{10} \rightarrow T_{13} \rightarrow T_{12}$,



Graph-Based Protocols (Cont.)

- Advantages:
  - The tree protocol ensures conflict serializability as well as freedom from deadlock.
  - Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.
    - Shorter waiting times, and increase in concurrency
    - Protocol is deadlock-free, no rollbacks are required
- Drawbacks
  - Protocol does not guarantee recoverability or cascade freedom
    - Need to introduce commit dependencies to ensure recoverability
  - Transactions may have to lock data items that they do not access.
    - Increased locking overhead, and additional waiting time (e.g., to access A, G)
    - Potential decrease in concurrency
- Schedules not possible under two-phase locking are possible under tree protocol, and vice versa.

Outline

- Lock-Based Protocols
- *Timestamp-Based Protocols
- *Validation-Based Protocols
- Multiple Granularity
- *Multiversion Schemes
- Deadlock Handling
- *Insert and Delete Operations
- *Concurrency in Index Structures

*Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j).
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data Q two timestamp values:
  - **W-timestamp**(Q) is the largest time-stamp of any transaction that executed **write**(Q) successfully.
  - **R-timestamp**(Q) is the largest time-stamp of any transaction that executed **read**(Q) successfully.

Timestamp-Based Protocols (Cont.)

- The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order.
- Suppose a transaction T_i issues a read(Q)
  - If TS(T_i) ≤ W-timestamp(Q), then T_i needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and T_i is rolled back.
  - If TS(T_i)≥ W-timestamp(Q), then the read operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).
- Suppose that transaction T_i issues write(Q).
  - If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced. Hence, the write operation is rejected, and T_i is rolled back.
  - If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. Hence, this write operation is rejected, and T_i is rolled back.
  - Otherwise, the write operation is executed, and W-timestamp(Q) is set to TS(T_i).

Example Use of the Protocol

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

T_1	T_2	T_3	T_4	T_5
				read (X)
	read (Y)			
read (Y)				
		write (Y)		
		write (Z)		
				read (Z)
	read (Z)			
	abort			
read (X)				
			read (W)	
		write (W)		
		abort		
				write (Y)
				write (Z)

Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be cascade-free, and may not even be recoverable.

Recoverability and Cascade Freedom

- Problem with timestamp-ordering protocol:
  - Suppose T_i aborts, but T_j has read a data item written by T_i
  - Then T_j must abort; if T_j had been allowed to commit earlier, the schedule is not recoverable.
  - Further, any transaction that has read a data item written by T_j must abort
  - This can lead to cascading rollback --- that is, a chain of rollbacks
- Solution:
  - A transaction is structured such that its writes are all performed at the end of its processing
  - All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
  - A transaction that aborts is restarted with a new timestamp

Thomas' Write Rule

- Modified version of the timestamp-ordering protocol in which obsolete write operations may be ignored under certain circumstances.
- When T_i attempts to write data item Q, if TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of {Q}. Hence, rather than rolling back T_i as the timestamp ordering protocol would have done, this {write} operation can be ignored.
- Otherwise this protocol is the same as the timestamp ordering protocol.
- Thomas' Write Rule allows greater potential concurrency. Unlike previous protocols, it allows some view-serializable schedules that are not conflict-serializable.

Outline

- Lock-Based Protocols
- *Timestamp-Based Protocols
- *Validation-Based Protocols
- Multiple Granularity
- *Multiversion Schemes
- Deadlock Handling
- Insert and Delete Operations
- *Concurrency in Index Structures

*Validation-Based Protocol

- Execution of transaction T_i is done in three phases.
  1. Read and execution phase: Transaction T_i writes only to temporary local variables
  2. Validation phase: Transaction T_i performs a ``validation test'' to determine if local variables can be written without violating serializability.
  3. Write phase: If T_i is validated, the updates are applied to the database; otherwise, T_i is rolled back.
- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
  - Assume for simplicity that the validation and write phase occur together, atomically and serially
    - i.e., only one transaction executes validation/write at a time.
- Also called as optimistic concurrency control since transaction executes fully in the hope that all will go well during validation

Validation-Based Protocol (Cont.)

- Each transaction T_i has 3 timestamps:
  - Start(T_i) : the time when T_i started its execution
  - Validation(T_i): the time when T_i entered its validation phase
  - Finish(T_i) : the time when T_i finished its write phase

- Serializability order is determined by timestamp given at validation time, to increase concurrency.
  - Thus, TS(T_i) is given the value of Validation(T_i).

- This protocol is useful and gives greater degree of concurrency if probability of conflicts is low.
  - Because the serializability order is not pre-decided, and
  - Relatively few transactions will have to be rolled back.

Validation Test for Transaction T_j

- If for all T_i with TS (T_i) < TS (T_j) either one of the following condition holds:
  - finish(T_i) < start(T_j)
  - start(T_j) < finish(T_i) < validation(T_j) and the set of data items written by T_i does not intersect with the set of data items read by T_j.

  then validation succeeds and T_j can be committed. Otherwise, validation fails and T_j is aborted.
- Justification: Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
  - The writes of T_j do not affect reads of T_i since they occur after T_i has finished its reads.
  - The writes of T_i do not affect reads of T_j since T_j does not read any item written by T_i.

Schedule Produced by Validation

- Example of schedule produced using validation

T_{25}	T_{26}
read (B)	
	read (B)
	$B := B$ 50
	read (A)
	$A := A + 50$
read (A)	
⟨ *validate* ⟩	
display ($A + B$)	
	⟨ *validate* ⟩
	write (B)
	write (A)

Outline

- Lock-Based Protocols
- *Timestamp-Based Protocols
- *Validation-Based Protocols
- Multiple Granularity
- *Multiversion Schemes
- Deadlock Handling
- *Insert and Delete Operations
- *Concurrency in Index Structures

Multiple Granularity

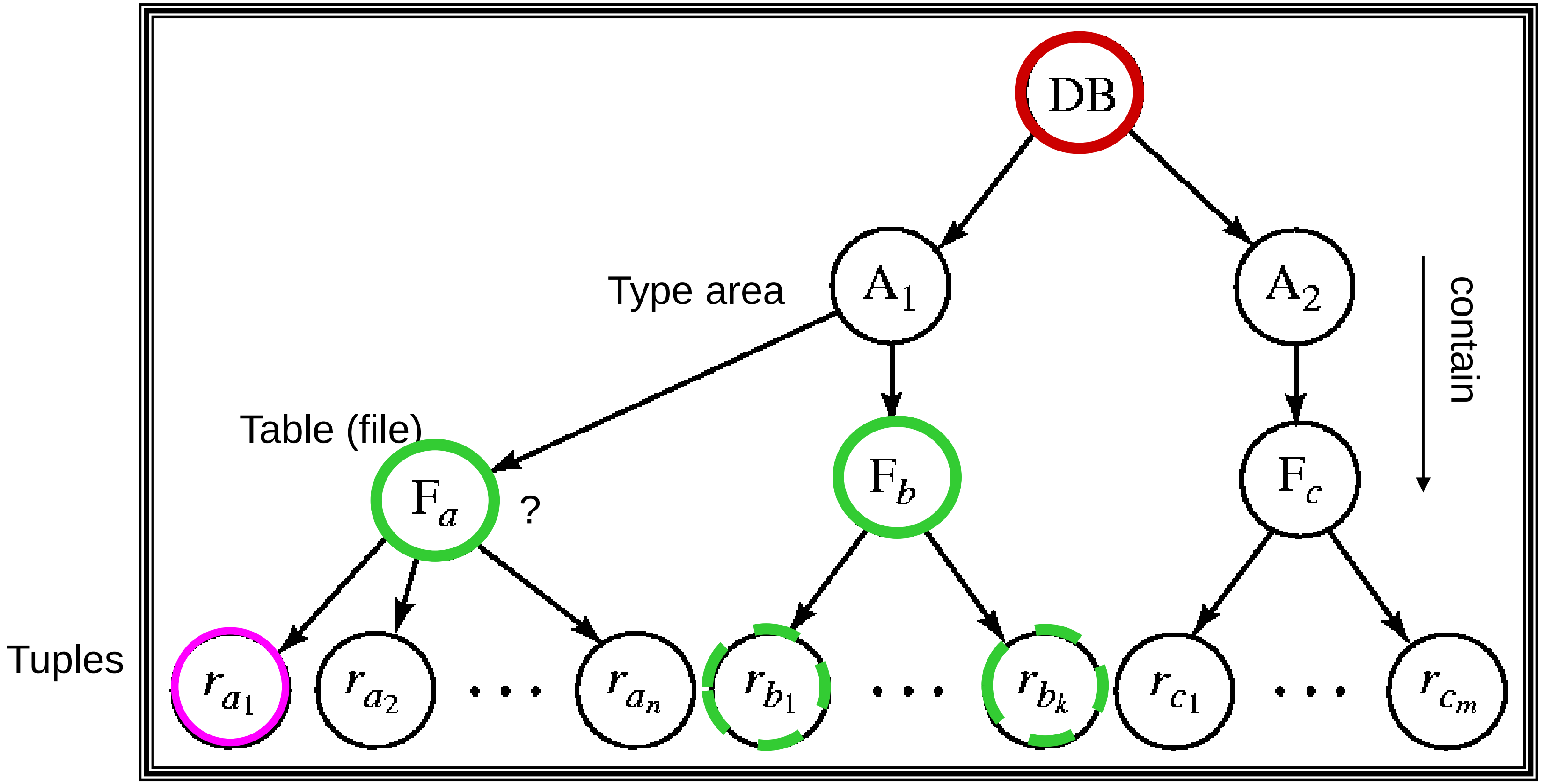
- For the convenience, allow data items to be locked in various sizes according to to the requirements---multiple granularity.

- Define a hierarchy of data granularities, where the small granularities are nested within larger ones, and can be represented graphically as a tree (but don't confuse with tree-locking protocol) (see next page)

- When a transaction locks a node in the tree explicitly, it implicitly locks all the node's descendents in the same mode.

- Granularity of locking (level in tree where locking is done):
  - Fine granularity (细粒度，lower in tree): high concurrency, high locking overhead
  - Coarse granularity (粗粒度，higher in tree): low locking overhead, low concurrency

Example of Granularity Hierarchy



The highest level in the example hierarchy is the entire database.
The levels below are of type *area*, *file* and *record* in that order.

Intention Lock

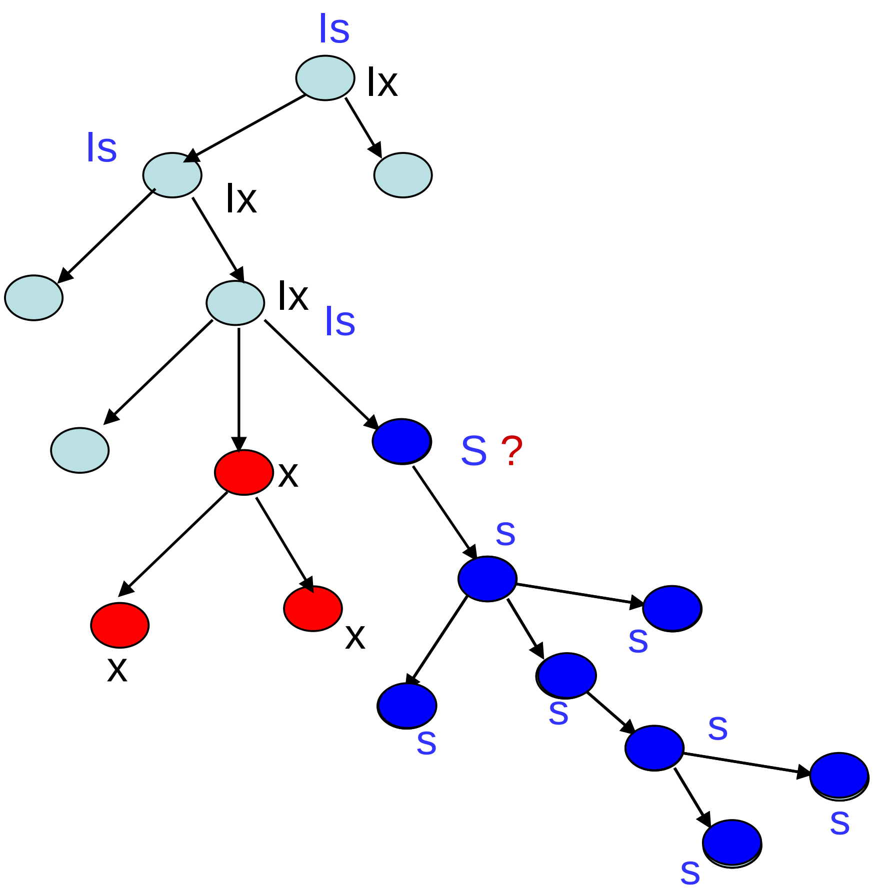
- Problem: T_1 locked r_{a1} in X-lock, T_2 locked F_b in S-lock. Now T_3 wishes to lock F_a in S-lock. T_4 wishes to lock the entire DB in S-lock. (search the entire tree!)

- Intention locks are put on all the ancestors of a node before that node is locked explicitly.

- Intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Intention Lock Modes

- There are three intention lock modes with multiple granularity:
  - Intention-shared (IS，共享型意向锁): indicates explicit locking at a lower level of the tree with shared locks.（表明其后代存在S锁)
  - Intention-exclusive (IX ， 排它型意向锁): indicates explicit locking at a lower level with exclusive locks。 (表明其后代存在X锁)
  - Shared and intention-exclusive (SIX，共享排它型意向锁): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.   SIX=S+IX


IS
IX
IS
IX
IX
IS
S ?
X
S
X
X
S
S
S
S
S
S

Compatibility Matrix

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

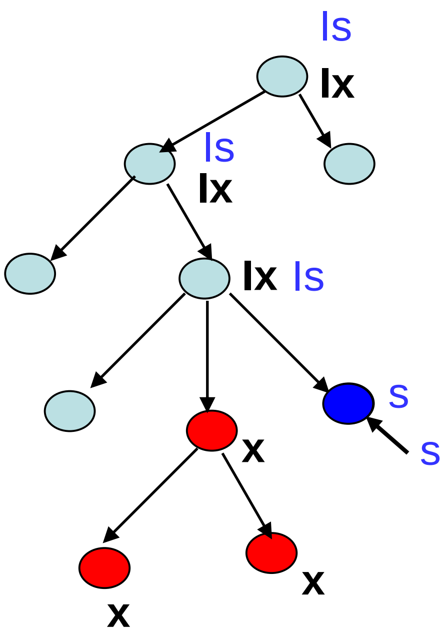
Compatibility Matrix with Intention Lock Modes

- The compatibility matrix for all lock modes is: （二个事务对同一数据对象加锁时的相容或冲突）

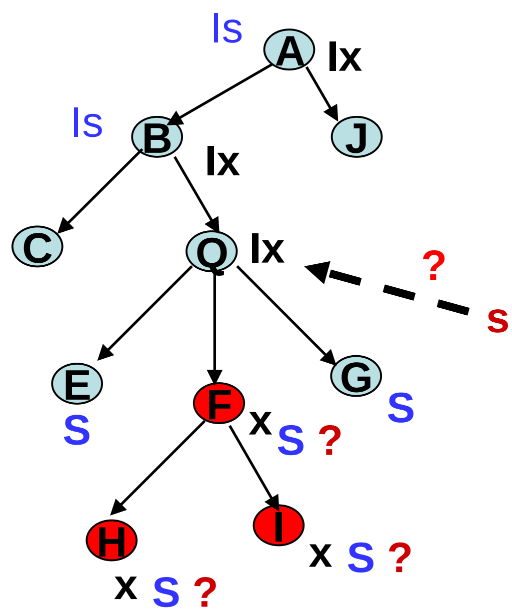
其他事务已加的锁

一个事务的加锁申请

	IS	IX	S	S IX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
S IX	✓	×	×	×	×
X	×	×	×	×	×

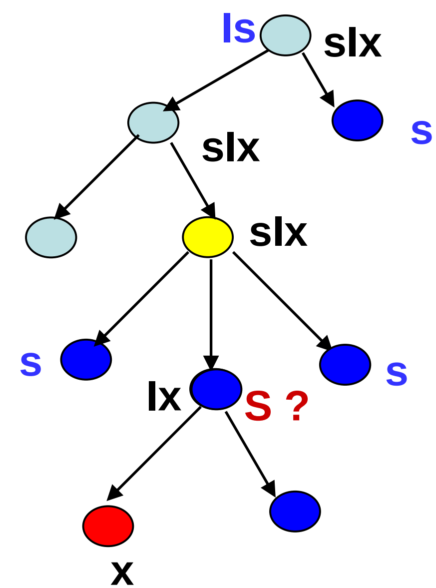


(A) Ix, Is兼容



To Lock-s (Q) ?

(B)Ix, s不兼容



(C) sIx

Multiple Granularity Locking Scheme

- Transaction T_i can lock a node Q, using the following rules:
  1. The lock compatibility matrix must be observed.
  2. The root of the tree must be locked first, and may be locked in any mode.
  3. A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
  4. A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
  5. T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
  6. T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.

  Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order. (加锁自顶向下，解锁自下而上，且遵守2PL协议)

  Advantages: 增强并发性，降低加锁开销。

Outline

- Lock-Based Protocols
- *Timestamp-Based Protocols
- *Validation-Based Protocols
- Multiple Granularity
- *Multiversion Schemes
- Deadlock Handling
- *Insert and Delete Operations
- *Concurrency in Index Structures

*Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency.
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- Each successful **write** results in the creation of a new version of the data item written.
- Use timestamps to label versions.
- When a **read**(*Q*) operation is issued, select an appropriate version of *Q* based on the timestamp of the transaction, and return the value of the selected version.
- **read**s never have to wait as an appropriate version is returned immediately.

Multiversion Timestamp Ordering

- Each data item Q has a sequence of versions <Q_1, Q_2,...., Q_m>. Each version Q_k contains three data fields:
  - **Content** -- the value of version Q_k.
  - **W-timestamp**(Q_k) -- timestamp of the transaction that created (wrote) version Q_k
  - **R-timestamp**(Q_k) -- largest timestamp of a transaction that successfully read version Q_k
- When a transaction T_i creates a new version Q_k of Q, Q_k's W-timestamp and R-timestamp are initialized to TS(T_i).
- R-timestamp of Q_k is updated whenever a transaction T_j reads Q_k, and TS(T_j) > R-timestamp(Q_k).

Multiversion Timestamp Ordering (Cont.)

- Suppose that transaction T_i issues a **read**(Q) or **write**(Q) operation. Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i).
  1. If transaction T_i issues a **read**(Q), then the value returned is the content of version Q_k.
  2. If transaction T_i issues a **write**(Q)
     1. if TS(T_i) < R-timestamp(Q_k), then transaction T_i is rolled back.
     2. if TS(T_i) = W-timestamp(Q_k), the contents of Q_k are overwritten
     3. else a new version of Q is created.
- Observe that
  - Reads always succeed.
  - A write by T_i is rejected if some other transaction T_j that (in the serialization order defined by the timestamp values) should read T_i's write, has already read a version created by a transaction older than T_i.
- Protocol guarantees serializability.

Multiversion Two-Phase Locking

- Differentiates between read-only transactions and update transactions

  Set Transaction read only;

  Set Transaction read wirte;
- Update transactions acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
  - Each successful write results in the creation of a new version of the data item written.
  - Each version of a data item has a single timestamp whose value is obtained from a counter ts-counter that is incremented during commit processing.
- Read-only transactions are assigned a timestamp by reading the current value of ts-counter before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads.

Multiversion Two-Phase Locking (Cont.)

- When an update transaction wants to read a data item:
  - It obtains a shared lock on it, and reads the latest version.
- When it wants to write an item
  - It obtains X lock on; it then creates a new version of the item and sets this version's timestamp to ∞.
- When update transaction T_i completes, commit processing occurs:
  - T_i sets timestamp on the versions it has created to **ts-counter** + 1
  - T_i increments **ts-counter** by 1
- Read-only transactions that start after T_i increments **ts-counter** will see the values updated by T_i.
- Read-only transactions that start before T_i increments the **ts-counter** will see the value before the updates by T_i.
- Only serializable schedules are produced.

Outline

- Lock-Based Protocols
- *Timestamp-Based Protocols
- *Validation-Based Protocols
- Multiple Granularity
- *Multiversion Schemes
- Deadlock Handling
- *Insert and Delete Operations
- *Concurrency in Index Structures

Deadlock Handling

- Consider the following two transactions:

  T_1: write (A) write(B) T_2: write(B) write(A)

- Schedule with deadlock

T_1	T_2
lock-X on A	
write (A)	
	lock-X on B
	write (B)
	wait for **lock-X** on A
wait for **lock-X** on B	

Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

How to handing?

- Deadlock prevention
- Deadlock detection and deadlock recovery

Deadlock prevention

- Deadlock prevention protocols ensure that the system will never enter into a deadlock state. Some prevention strategies:
  - 1) Require that each transaction locks all its data items before it begins execution (predeclaration) – conservative 2PL. (Either all or none are locked)
    - Disadvantages: bad concurrency, hard to predict
  - 2) Impose partial ordering of all data items and require that a transaction can lock data items only in the order (graph-based protocol). ---- therefore never form a cycle.

Deadlock prevention (Cont.)

- Both in wait-die and in wound-wait schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.

- **Timeout-Based Schemes**:
  - A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - Thus deadlocks are not possible
  - Simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

More Deadlock Prevention Strategies

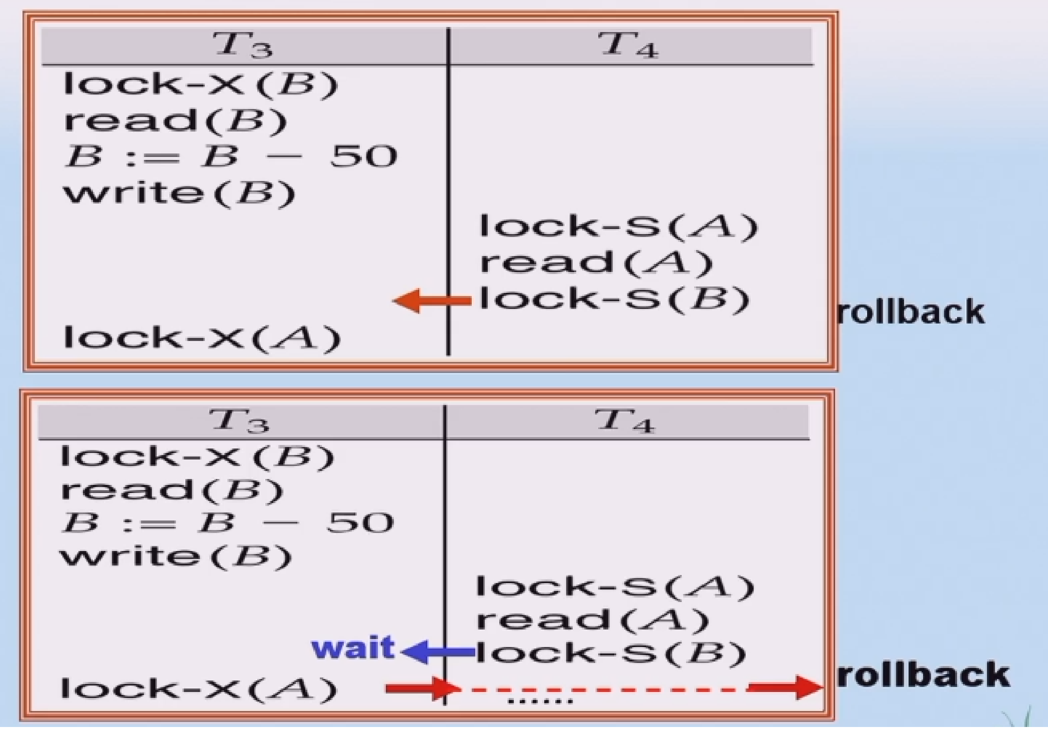
- Following schemes use transaction timestamps for the sake of deadlock prevention alone.

- Wait-die scheme — non-preemptive
  - Older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
  - A transaction may die several times before acquiring needed data item

- Wound-wait scheme — preemptive
  - Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
  - May be fewer rollbacks than *wait-die* scheme

More Deadlock Prevention Strategies

- Wait-die
- Wound-wait

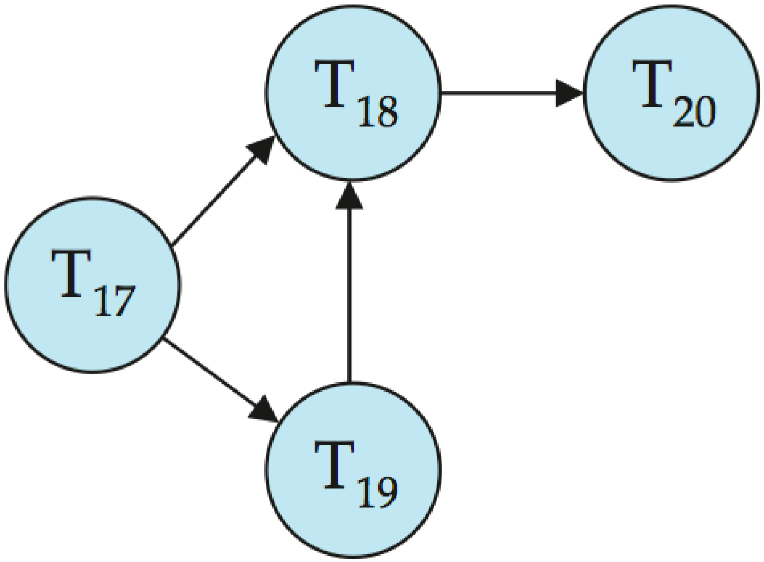


Deadlock Detection

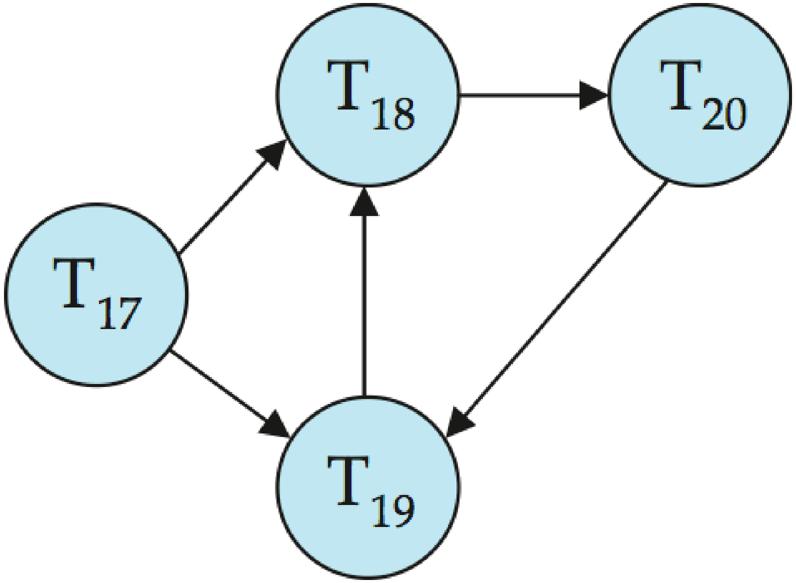
- Deadlocks can be described as a wait-for graph, which consists of a pair $G = (V,E)$,
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.

- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.
- When T_i requests a data item currently being held by T_j, then the edge T_i T_j is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.

Deadlock Detection (Cont.)



Wait-for graph without a cycle



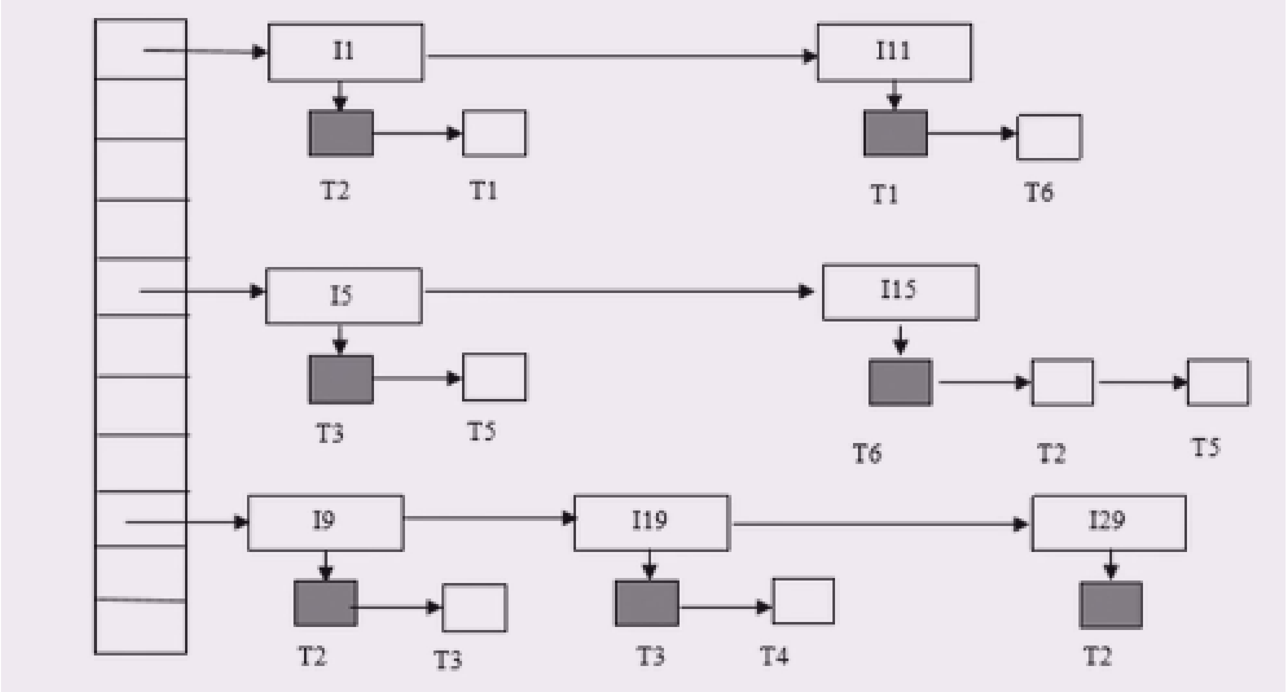
Wait-for graph with a cycle

Deadlock Recovery

- When deadlock is detected:
  - Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
  - Rollback -- determine how far to roll back transaction
    - Total rollback: Abort the transaction and then restart it.
    - Partial rollback: More effective to roll back transaction only as far as necessary to break deadlock.
  - Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

Exercise

- Following figure shows an example of a lock table:
  - Which transaction are involved in deadlock?
  - To break the deadlock, which transaction should be rolled back?



Outline

- Lock-Based Protocols
- *Timestamp-Based Protocols
- *Validation-Based Protocols
- Multiple Granularity
- *Multiversion Schemes
- Deadlock Handling
- *Insert and Delete Operations
- *Concurrency in Index Structures

Insert and Delete Operations

- If two-phase locking is used :
  - A delete operation may be performed only if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted.
  - A transaction that inserts a new tuple into the database is given an X-mode lock on the tuple

- Insertions and deletions can lead to the phantom phenomenon.
  - A transaction that scans a relation
    - (e.g., find sum of balances of all accounts in Perryridge)

    and a transaction that inserts a tuple in the relation
    - (e.g., insert a new account at Perryridge)

      (conceptually) conflict in spite of not accessing any tuple in common.
  - If only tuple locks are used, non-serializable schedules can result
    - E.g., the scan transaction does not see the new account, but reads some other tuple written by the update transaction

Insert and Delete Operations (Cont.)

- The transaction scanning the relation is reading information that indicates what tuples the relation contains, while a transaction inserting a tuple updates the same information.
  - The information should be locked.
- One solution:
  - Associate a data item with the relation, to represent the information about what tuples the relation contains.
  - Transactions scanning the relation acquire a shared lock in the data item.
  - Transactions inserting or deleting a tuple acquire an exclusive lock on the data item. (Note: locks on the data item do not conflict with locks on individual tuples.)
- Above protocol provides very low concurrency for insertions/ deletions.
- Index locking protocols provide higher concurrency while preventing the phantom phenomenon, by requiring locks on certain index buckets.

Index Locking Protocol

- Every relation must have at least one index. Access to a relation must be made only through one of the indices on the relation.
- A transaction T_i that performs a lookup must lock all the index buckets that it accesses, in S-mode.
- A transaction T_i may not insert a tuple t_i into a relation r without updating all indices to r.
- T_i must perform a lookup on every index to find all index buckets that could have possibly contained a pointer to tuple t_i, had it existed already, and obtain locks in X-mode on all these index buckets. T_i must also obtain locks in X-mode on all index buckets that it modifies.
- The rules of the two-phase locking protocol must be observed.

Outline

- Lock-Based Protocols
- *Timestamp-Based Protocols
- *Validation-Based Protocols
- Multiple Granularity
- *Multiversion Schemes
- Deadlock Handling
- *Insert and Delete Operations
- *Concurrency in Index Structures

*Concurrency in Index Structures

- Indices are unlike other database items in that their only job is to help in accessing data.
- Index-structures are typically accessed very often, much more than other database items.
  - Treating index-structures like other database items, e.g., by 2-phase locking of index nodes can lead to low concurrency.
- There are several index concurrency protocols where locks on internal nodes are released early, and not in a two-phase fashion.
  - It is acceptable to have nonserializable concurrent access to an index as long as the accuracy of the index is maintained.
    - In particular, the exact values read in an internal node of a B^+-tree are irrelevant so long as we land up in the correct leaf node.

Concurrency in Index Structures (Cont.)

- Example of index concurrency protocol:
- Use **crabbing** instead of two-phase locking on the nodes of the B^+-tree, as follows. During search/insertion/deletion:
  - First lock the root node in shared mode.
  - After locking all required children of a node in shared mode, release the lock on the node.
  - During insertion/deletion, upgrade leaf node locks to exclusive mode.
  - When splitting or coalescing requires changes to a parent, lock the parent in exclusive mode.
- Above protocol can cause excessive deadlocks
  - Searches coming down the tree deadlock with updates going up the tree
  - Can abort and restart search, without affecting transaction
- Better protocols are available; see Section 16.9 for one such protocol, the B-link tree protocol
  - Intuition: release lock on parent before acquiring lock on child
    - And deal with changes that may have happened between lock release and acquire

Q & A

Your questions and suggestions are expected for me.

Thanks a lot!

Questions?

Exercises: 18.1, 18.2, 18.7, and 18.18
(see Pages 899-902)