Lecture 13
Concurrency Control

Shuigeng Zhou

June 11, 2014

School of Computer Science
Fudan University

Outline

- Lock-Based Protocols
- Graph-Based Protocols
- Timestamp-Based Protocols
- Multiple Granularity
- Deadlock Handling
- Insert and Delete Operations

Lock-Based Protocols

Lock-Based Protocols

- A **lock** is a mechanism to control concurrent access to a data item
- 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
- If a lock cannot be granted, the requesting transaction is made to **wait** till all incompatible locks have been released. The lock is then granted.

Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

T_2: **lock-S**(*A*);
read (*A*);
unlock(*A*);
lock-S(*B*);
read (*B*);
unlock(*B*);
display(*A*+*B*)

- Locking as above is not sufficient to guarantee serializability.
- A **locking protocol** is a set of rules
  - followed by all transactions while requesting and releasing locks.
  - Locking protocols restrict the set of possible schedules.

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)	

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.
- **Starvation** (饥饿) is also possible if concurrency control manager is badly designed. Eg.
  - 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. (livelock: 活锁)
  - 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 （两阶段加锁协议）
- 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**
  - Lock point: the point in the schedule where the transaction has obtained its final lock (the end of the growing phase)

J. D. Ullman. Principles of Databse and Knowledge-base Systems. 1988

The Two-Phase Locking Protocol (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks
- Example

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)	

The Two-Phase Locking Protocol (Cont.)

- Cascading roll-back is possible under two-phase locking

T_5	T_6	T_7
lock-x (A)		
read (A)		
lock-s (B)		
read (B)		
write (A)		
unlock (A)		
	lock-x (A)	
	read (A)	
	write (A)	
	unlock (A)	
		lock-s (A)
		read (A)

  - To avoid this, follow a modified protocol called **strict two-phase locking**.
  - A transaction must hold all its exclusive locks till it commits/aborts.

The Two-Phase Locking Protocol (Cont.)

- **Rigorous two-phase locking** 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
- Lock conversion mechanism

T8: read(a1);
read(a2);
...
read(an);
write(a1);
T9: read(a1);
read(a2);
display(a1+a2)

T_8	T_9
lock-s (a_1)	
	lock-s (a_1)
lock-s (a_2)	
	lock-s (a_2)
lock-s (a_3)	
lock-s (a_4)	
	unlock-s(a_1)
	unlock-s(a_2)
lock-s (a_n)	
upgrade (a_1)	

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**
    - can release a **lock-X**
    - can convert a **lock-X** to a **lock-S** (**downgrade**)
- This protocol assures conflict serializability.

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

Automatic Acquisition of Locks (Cont.)

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

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

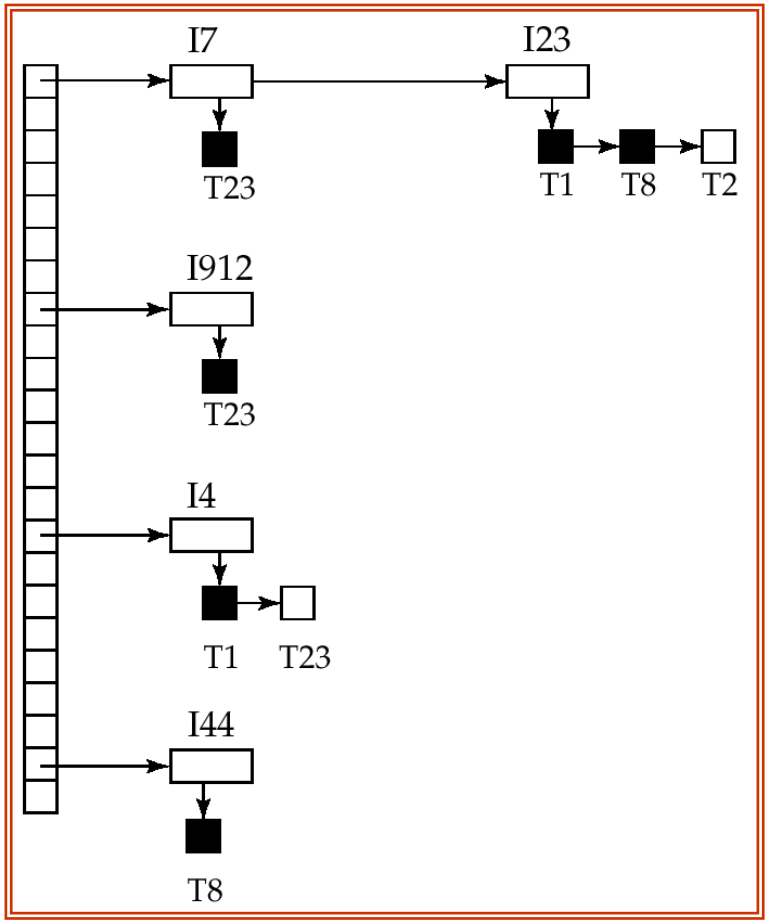
- All locks are released after commit or abort

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

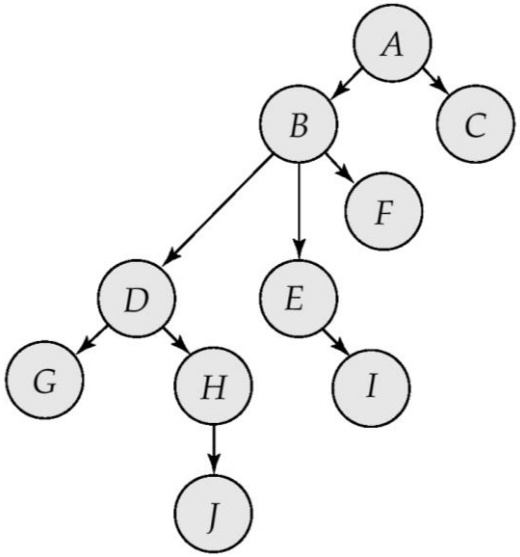


Graph-Based Protocols

Graph-Based Protocols

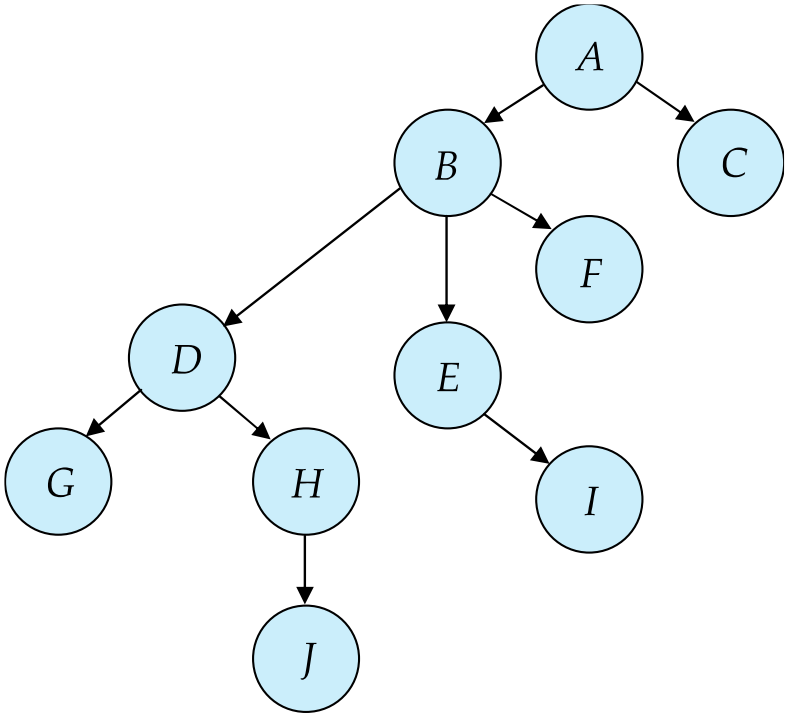
- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering $\rightarrow$ on the set **D** = $\{d_1, d_2, \ldots, d_h\}$ of all data items.
  - If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j.
  - Implies that 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



- Only exclusive locks are allowed.
  - 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.
  - Data items may be unlocked at any time.
  - A data item cannot be relocked by T_i.

Tree Protocol (Cont.)



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*)			

Tree Protocol (Cont.)

- The tree protocol ensures conflict serializability as well as freedom from deadlock.
- Unlocking may occur earlier than in the two-phase locking protocol.
  - shorter waiting times, and increase in concurrency
  - protocol is deadlock-free, no rollbacks are required
  - the abort of a transaction can still lead to cascading rollbacks.
- However, may have to lock data items that it does not access.
  - increased locking overhead, and additional waiting time
  - potential decrease in concurrency

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)
  1. 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.
  2. If TS(T_i)≥ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), TS(T_i)).

Timestamp-Based Protocols (Cont.)

- Suppose that transaction T_i issues **write**(Q).
  1. 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 the value would never be produced.
     - Hence, the **write** operation is rejected, and T_i is rolled back.
  2. 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.
  3. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

Timestamp-Based Protocols (Cont.)

- Example

T25: read(B);
read(A);
display(A+B);

T26: read(B);
B:=B-50;
write(B);
read(A);
A:=A+50;
write(A);
display(A+B);

T_{25}	T_{26}
read (B)	
	read (B)
	$B := B - 50$
	write (B)
read (A)	
	read (A)
display ($A + B$)	
	$A := A + 50$
	write (A)
	display ($A + B$)

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.

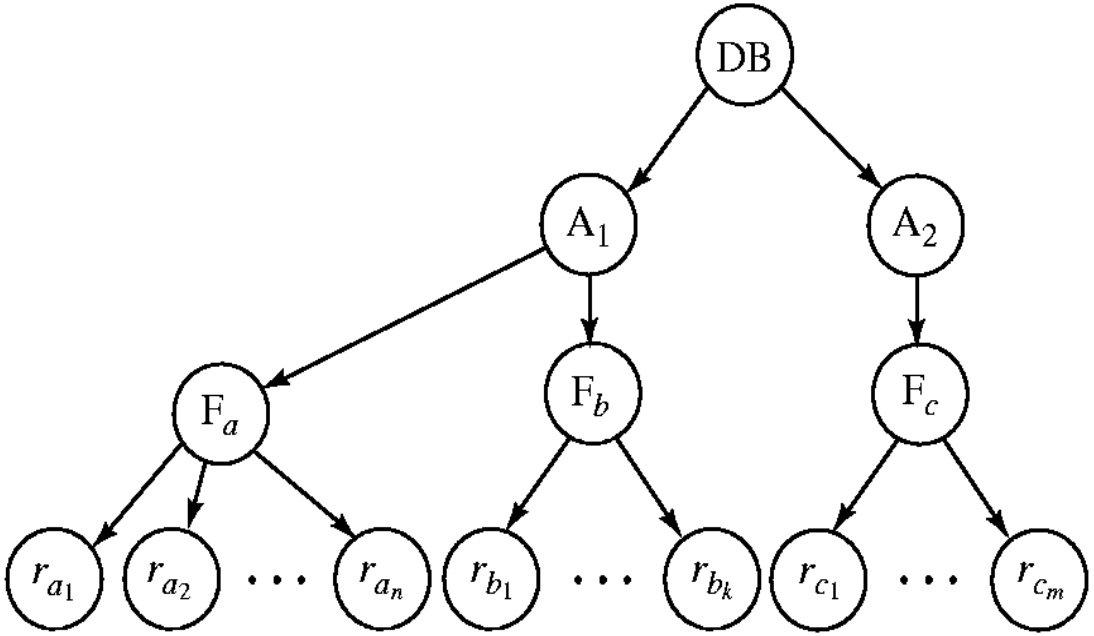
Multiple Granularity

Multiple Granularity

- Allow data items to be of various sizes and define a hierarchy of data granularities
- Can be represented graphically as a tree
- 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:
  - *fine granularity* (lower in tree): high concurrency, high locking overhead
  - *coarse granularity* (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（意向锁） Modes

- Three additional lock modes with multiple granularity:
  - ***intention-shared*** (IS)
    - indicates explicit locking at a lower level of the tree but only with shared locks
  - ***intention-exclusive*** (IX)
    - indicates explicit locking at a lower level with exclusive or shared locks
  - ***shared and intention-exclusive*** (SIX)
    - 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
- intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Compatibility Matrix with Intention Lock Modes

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

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.

Deadlock Handling

Deadlock Handling

- Consider the following two transactions:

  T_1: write(X) T_2: write(Y)

  write(Y) write(X)

- Schedule with deadlock

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

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
- Two strategies: deadlock prevention vs. deadlock detection
- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state.
  - Require that each transaction locks all its data items before it begins execution **(predeclaration).**
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order **(graph-based protocol).**

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) younger transaction instead of waiting for it. Younger transactions may wait for older ones.
  - may be fewer rollbacks than *wait-die* scheme.

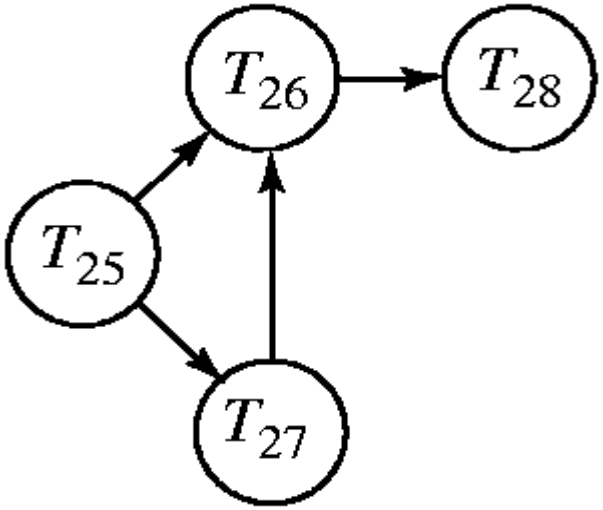
Deadlock prevention (Cont.)

- 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.

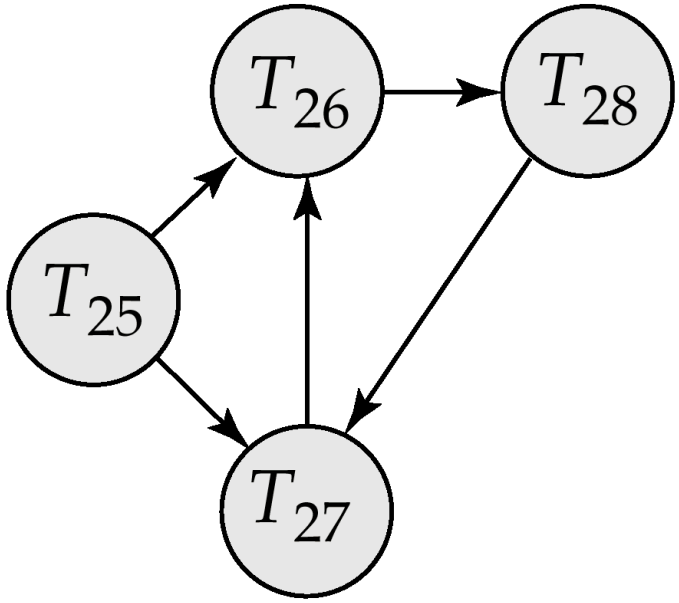
Deadlock Detection

- 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.
- The system is in a deadlock state iff 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.
  - 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

Insert and Delete Operations

Logical Error

- Logical error will happen if
  - Perform a read(Q) after delete(Q)
  - Perform a read(Q) before insert(Q)
  - To delete a nonexistent data item
- If two-phase locking is used :
  - A **delete** operation may be performed only if the transaction has an exclusive lock on the tuple.
  - A transaction that **inserts** a new tuple into the database is given an X-mode lock on the tuple.

Phantom Phenomenon

- Insertions and deletions can lead to the **phantom phenomenon (幻影现象)**.
  - A transaction that scans a relation and a transaction that inserts a tuple in the relation may conflict in spite of not accessing any tuple in common.

    Select sum(balance) from account where branch-name='Perryridge'

    Insert into account values(A-201, 'Perryridge', 900)
  - If only tuple locks are used, non-serializable schedules can result: the scan transaction may not see the new account, yet may be serialized after the insert transaction.

Solutions

- 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.
- 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, delete and update a tuple t_i into a relation r without updating all indices to r.
- The rules of the two-phase locking protocol must be observed.

Weak Levels of Consistency

Weak Levels of Consistency

- **Degree-two consistency:** differs from two-phase locking in that S-locks may be released at any time, and locks may be acquired at any time
  - X-locks must be held till end of transaction
  - Serializability is not guaranteed, programmer must ensure that no erroneous database state will occur.
- **Cursor stability:**
  - For reads, each tuple is locked, read, and lock is immediately released
  - X-locks are held till end of transaction
  - Special case of degree-two consistency

Weak Levels of Consistency in SQL

- SQL allows non-serializable executions
  - **Serializable**: is the default
  - **Repeatable read**: allows only committed records to be read, and repeating a read should return the same value
    - However, the phantom phenomenon need not be prevented
      - T1 may see some records inserted by T2, but may not see others inserted by T2
  - **Read committed**: same as degree two consistency, but most systems implement it as cursor-stability
  - **Read uncommitted**: allows even uncommitted data to be read

Homework

- Practice exercises
  - 16.2