Chapter 15:

Concurrency Control

Database System Concepts, 7th Ed.

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

- **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

Schedule With Lock Grants

- From now on, grants are not explicitly indicated
  - Assume grant happens just before the next instruction following lock request
- 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)		

Deadlock

- 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)	

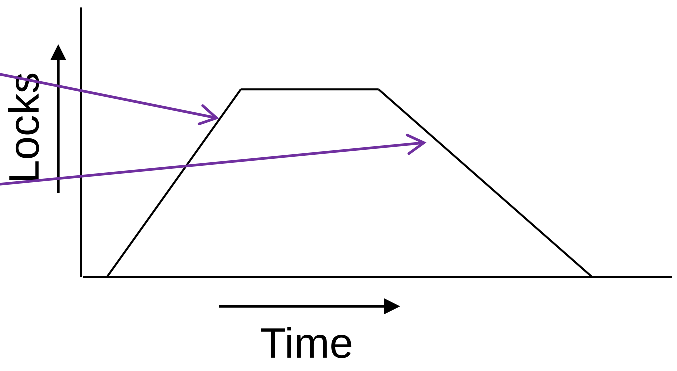
- Neither T_3 nor T_4 can make progress — executing **lock-S**(*B*) causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X**(*A*) causes T_3 to wait for T_4 to release its lock on *A*
- Such a situation is a **deadlock** situation
  - To handle a deadlock one of T_3 or T_4 must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

Deadlock

- The potential for deadlock exists in most locking protocols
- **Starvation** occurs when a transaction cannot complete its task for an indefinite period of time while other transactions continue
  - 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

- 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

- Two-phase locking protocol with **lock conversions**:
  - Growing 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**)
  - Shrinking Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (**downgrade**)
- This protocol ensures serializability

The Two-Phase Locking Protocol

- Extensions to basic two-phase locking needed to ensure recoverability of freedom from cascading roll-back
  - **Strict two-phase locking:** a transaction must hold all its exclusive locks till it commits/aborts
    - Ensures recoverability and avoids cascading rollbacks (avoids dirty reads – no one can read the updated data item before commit)
  - **Rigorous two-phase locking**: a transaction must hold *all* locks till commit/abort
    - Avoids nonrepeatable reads (no one can update the item when a shared lock is held on it)
    - Transactions can be serialized in the order in which they commit
- Most databases implement rigorous two-phase locking

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
           grant T_i a  lock-S on D;
           read(D)
         end
```

Automatic Acquisition of Locks

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

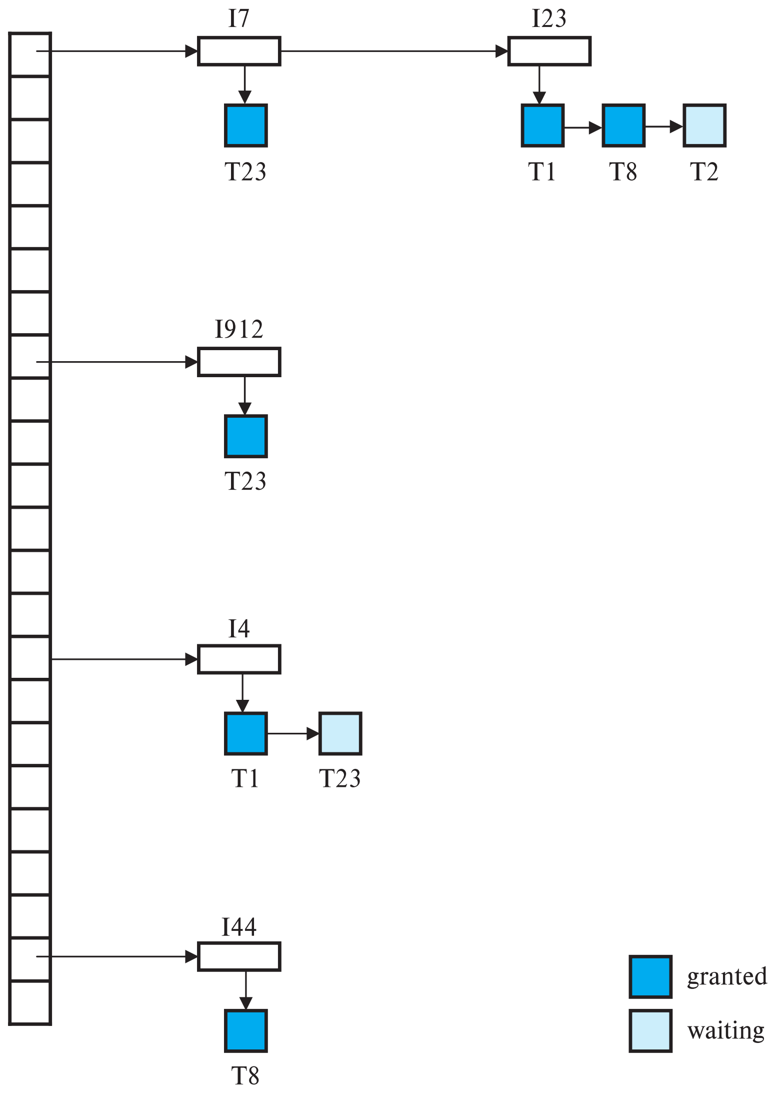
```
if $T_i$ has a lock-X on $D$
  then
    write($D$)
  else begin
      if necessary, wait until no other transaction has any lock on $D$,
      if $T_i$ has a lock-S on $D$
          then
            upgrade lock on $D$ to lock-X
          else
            grant $T_i$ a lock-X on $D$
          write($D$)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

- All locks are released after commit or abort

Implementation of Locking

- A **lock manager** can be implemented as a separate process
- Transactions can send lock and unlock requests as messages
- 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 an in-memory data-structure called a **lock table** to record granted locks and pending requests
    - Uses a hash table, hashed on the name of the data item, to find the linked list for a data item

Lock Table



- Data items are prefixed by I, and transactions by T
- Darker rectangles indicate granted locks, lighter 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

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

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)	

Deadlock Prevention

- **wait-die** scheme — non-preemptive
  - Older transaction may wait for younger one to release data item
  - Younger transactions never wait for older ones; the younger transactions are rolled back instead
  - A transaction may die several times before acquiring a lock
  - For example, suppose that transactions T14, T15, and T16 have timestamps 5, 10, and 15, respectively. If T14 requests a data item held by T15, then T14 will wait. If T16 requests a data item held by T15, then T16 will be rolled back.

Deadlock Prevention

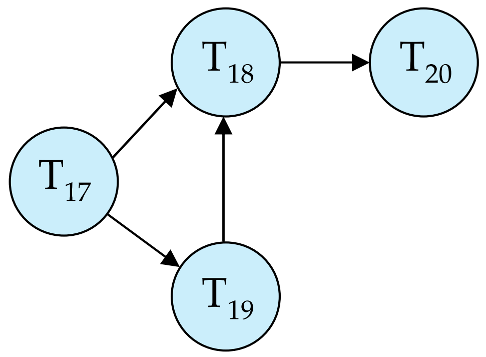
- **wound-wait** scheme — preemptive
  - Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it
  - Younger transactions may wait for older ones
  - For transactions T14, T15, and T16 above with respective timestamps 5, 10, and 15, if T14 requests a data item held by T15, then the data item will be preempted from T15, and T15 will be rolled back. If T16 requests a data item held by T15, then T16 will wait
- In both schemes, a rolled back transactions is restarted with its original timestamp
  - Ensures that older transactions have precedence over newer ones, and starvation is avoided

Deadlock Prevention

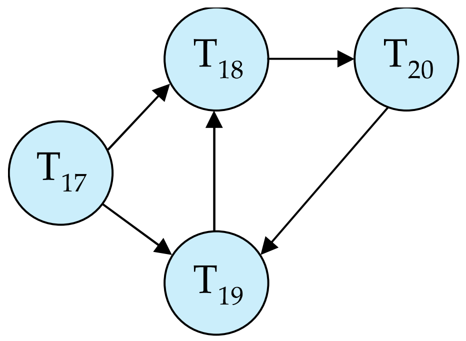
- **Timeout-Based Schemes**
  - A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back
  - Ensures that deadlocks get resolved by timeout if they occur
  - Simple to implement
  - But may roll back transaction unnecessarily in absence of deadlock
    - Difficult to determine good value of the timeout interval
  - Starvation is also possible

Deadlock Detection

- **Wait-for graph**
  - *Vertices:* transactions
  - *Edge from $T_i \rightarrow T_j$*. : if T_i is waiting for a lock held in conflicting mode by T_j
- The system is in a deadlock state if and only if the wait-for graph has a cycle
- Invoke a deadlock-detection algorithm periodically to look for cycles



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

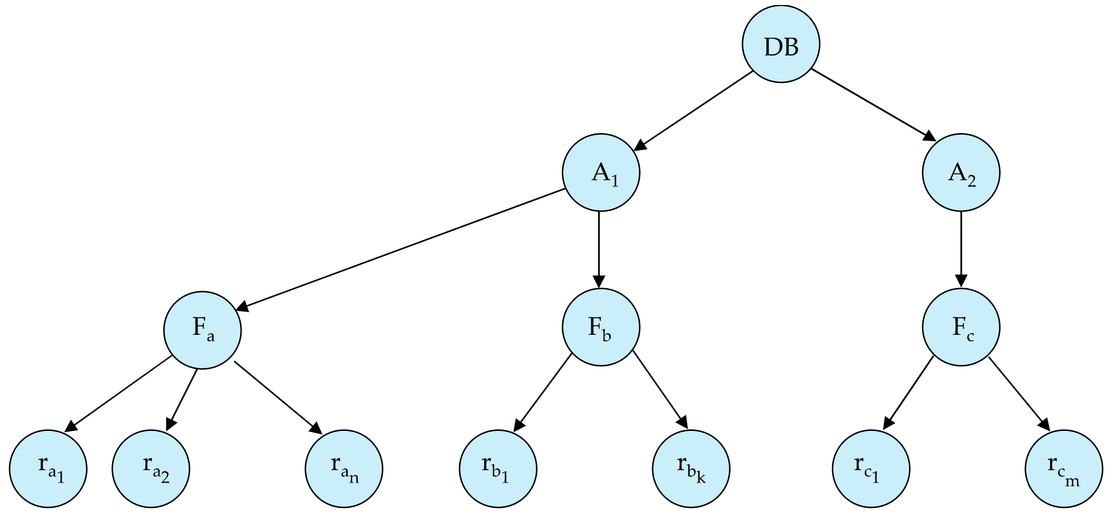
- When deadlock is detected
  - Some transaction will have to rolled back (made a **victim**) to break deadlock cycle
    - Select that transaction as victim that will incur minimum cost
  - Rollback -- determine how far to roll back transaction
    - **Total rollback**: Abort the transaction and then restart it
    - **Partial rollback**: Roll back the victim transaction only as far as necessary to release locks that another transaction in cycle is waiting for
      - The deadlock detection mechanism should decide which locks the selected transaction needs to release in order to break the deadlock. The selected transaction must be rolled back to the point where it obtained the first of these locks, undoing all actions it took after that point
- Starvation can happen
  - Same transaction is always chosen as a victim
  - One solution: oldest transaction in the deadlock set is never chosen as victim

Multiple Granularity

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
- Can be represented graphically as a tree
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendants in the same mode
- Granularity of locking (level in tree where locking is done):
  - **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 levels, starting from the coarsest (top) level are
  - *database*
  - *area*
  - *file*
  - *record*



Intention Lock Modes

- Suppose a transaction wishes to lock the entire database
  - To do so, it must lock the root of the hierarchy
  - How does the system determine if the root node can be locked:
    - One possibility is for it to search the entire tree, which is far from efficient
- A more efficient way to gain this knowledge is to introduce a new class of lock modes, called intention lock modes
  - If a node is locked in an intention mode, explicit locking is done at a lower level of the tree (that is, at a finer granularity).
- Intention locks are put on all the ancestors of a node before that node is locked explicitly
  - Thus, a transaction does not need to search the entire tree to determine whether it can lock a node successfully
- A transaction wishing to lock a node Q must traverse a path in the tree from the root to Q
  - While traversing the tree, the transaction locks the various nodes in an intention mode

Intention Lock Modes

- In addition to S and X lock modes, there are three additional lock modes with multiple granularity:
  - ***intention-shared*** (IS): indicates that a shared lock(s) will be requested on some descendant node(s)
  - ***intention-exclusive*** (IX): indicates that an exclusive lock(s) will be requested on some descendant node(s)
  - ***shared and intention-exclusive*** (SIX): that the current node is locked in shared mode, but an exclusive lock(s) will be requested on some descendant node(s)
- 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

- The compatibility matrix for all lock modes is:

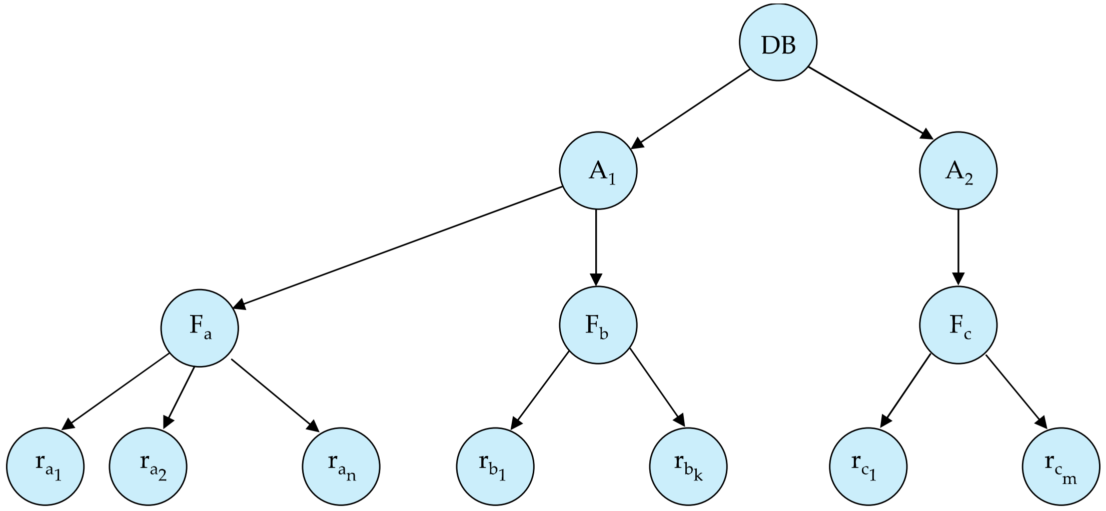
	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

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

Multiple Granularity Locking Scheme

- Suppose that transaction T_{21} reads record r_{a2} in file F_a. Then, T_{21} needs to lock the database, area A_1, and F_a in IS mode (and in that order), and finally to lock r_{a2} in S mode
- Suppose that transaction T_{22} modifies record r_{a9} in file F_a. Then, T_{22} needs to lock the database, area A_1, and file F_a (and in that order) in IX mode, and finally to lock r_{a9} in X mode
- Suppose that transaction T_{23} reads all the records in file F_a. Then, T_{23} needs to lock the database and area A_1 (and in that order) in IS mode, and finally to lock F_a in S mode
- Suppose that transaction T_{24} reads the entire database. It can do so after locking the database in S mode
- Note that transactions T_{21}, T_{23}, and T_{24} can access the database concurrently. Transaction T_{22} can execute concurrently with T_{21}, but not with either T_{23} or T_{24}



Timestamp-Based Protocol

- Each transaction T_i is issued a timestamp TS(T_i) when it enters the system
  - Each transaction has a *unique* timestamp
  - Newer transactions have timestamps strictly greater than earlier ones
  - Timestamp could be based on a logical counter
    - Real time may not be unique
    - Can use logical counter to ensure uniqueness
- Timestamp-based protocols manage concurrent execution such that

  time-stamp order = serializability order

Timestamp-Ordering Protocol

The **timestamp ordering (TSO) 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
- Imposes rules on read and write operations to ensure that
  - Any conflicting operations are executed in timestamp order
  - Out of order operations cause transaction rollback

Timestamp-Ordering Protocol

- Suppose a transaction T_i issues a **read**(Q)

1. If TS(T_i) $\leq$ **W**-timestamp(Q), then T_i attempts to read a future value of Q
   - Hence, the **read** operation is rejected, and T_i is rolled back
2. If TS(T_i) $\geq$ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to

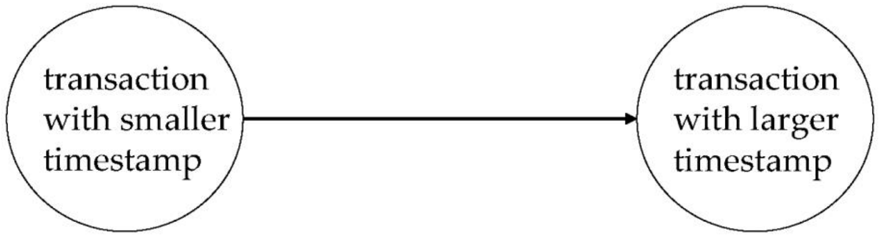
   max(R-timestamp(Q), TS(T_i))

Timestamp-Ordering Protocol

- 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 that 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 (produce) an obsolete (superseded) 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)

Correctness of Timestamp-Ordering Protocol

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


transaction with smaller timestamp
transaction with larger timestamp

Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.

Example of Schedule Under TSO

Assume that initially:
R-TS(A) = W-TS(A) = 0
R-TS(B) = W-TS(B) = 0
Assume TS(T_{25}) = 25 and
TS(T_{26}) = 26

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($A + B$)

- The top schedule is valid under TSO because T_{25} carries out only Reads, and it does not read any future values written by T_{26} for the same date item

Assume:
R-TS(Q)=W-TS(Q)=0
TS(T_{27}) = 27
TS(T_{28}) = 28

T_{27}	T_{28}
read(Q)	
	write(Q)
write(Q)	

- T_{27} needs to roll back because it is attempting to produce an obsolete value for Q in its write(Q)

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 produce an obsolete value of {Q}
  - 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