



# Database Systems

## Lecture19 – Chapter 18: Concurrency Control



Beomseok Nam (남범석)

[bnam@skku.edu](mailto:bnam@skku.edu)



# Outline

- Lock-Based Protocols
  - Two Phase Locking Protocol
  - Graph-based Protocol
  - Multiple-Granularity Protocol



## 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
- 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.

## 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

# 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( $A$ )	
	read( $A$ )	grant-S( $A, T_2$ )
	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 called a **deadlock**.
  - To handle a deadlock one of  $T_3$  or  $T_4$  must be rolled back and its locks released.



## Deadlock (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

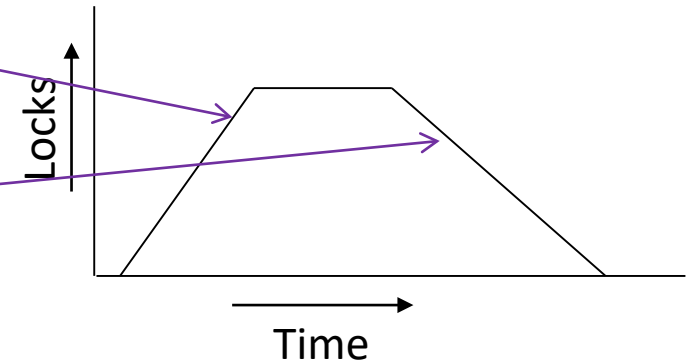
- A protocol which ensures conflict-serializable schedules.

- 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.
- 2PL is not a necessary condition for conflict-serializable schedules
- 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 (Cont.)

- 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.
  - **Rigorous two-phase locking:** a transaction must hold *all* locks till commit/abort.
    - Transactions can be serialized in the order in which they commit.
    - **Ensures recoverability and avoids cascading roll-backs**
- Most databases implement rigorous two-phase locking, *but refer to it as simply two-phase locking*
- **Two-phase locking *does not* ensure freedom from deadlocks**



# Lock Conversions

- 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

## 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 (Cont.)

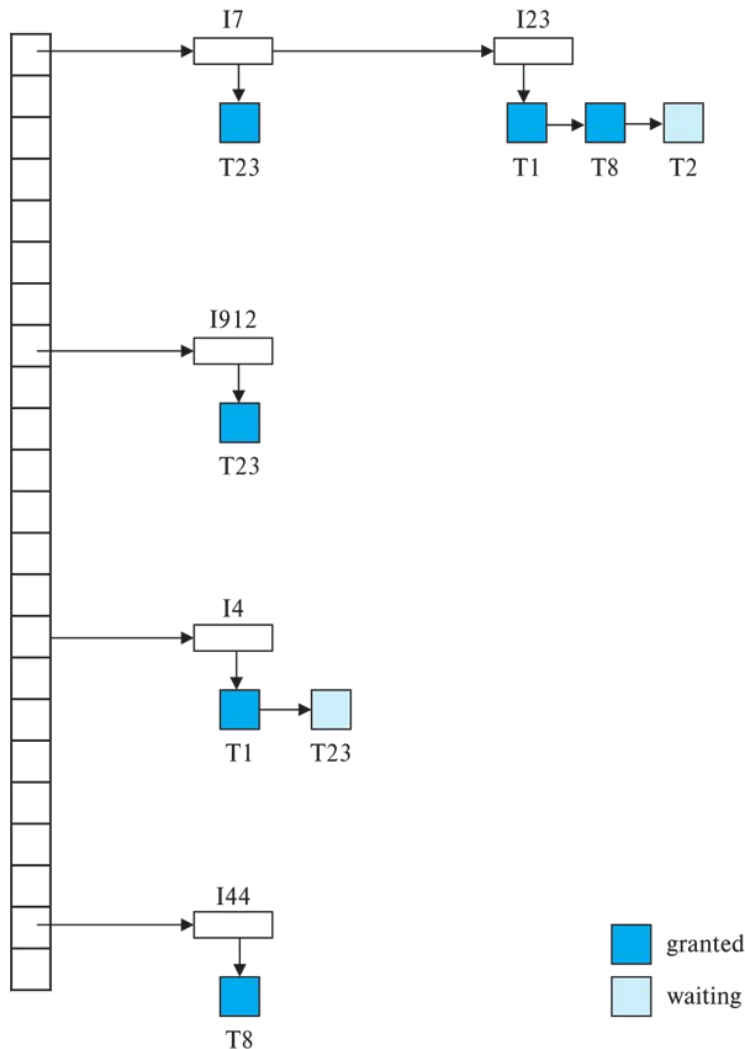
- The operation **write**( $D$ ) is processed as:
  - if**  $T_i$  has a **lock-X** on  $D$ 
    - then**
      - write( $D$ )
    - else begin**
      - if necessary wait until no other trans. has any lock on  $D$ ,
      - if  $T_i$  has a **lock-S** on  $D$ 
        - then**
          - upgrade** 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
- 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

# Lock Table



- Blue rectangles indicate granted locks, light colored 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

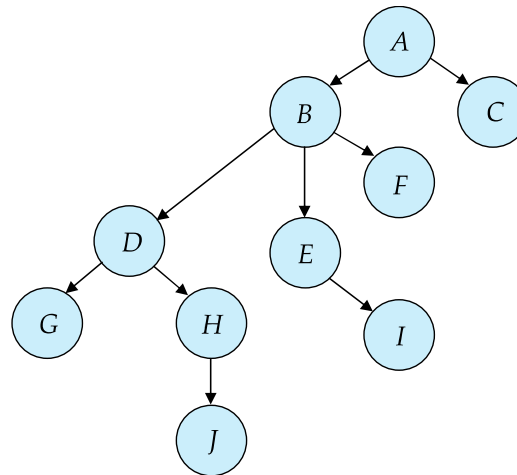
## Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering  $\rightarrow$  on the set  $\mathbf{D} = \{d_1, d_2, \dots, d_h\}$  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  $\mathbf{D}$  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 that has been locked and unlocked by  $T_i$  cannot subsequently be relocked by  $T_i$ .



## Graph-Based Protocols (Cont.)

- 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
    - potential decrease in concurrency
- Schedules not possible under two-phase locking are possible under the tree protocol, and vice versa.

# 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-X( $A$ )	lock-S( $A$ ) read( $A$ ) lock-S( $B$ )



# Deadlock Prevention

- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies:
  - Require that each transaction **locks all its data items before it begins execution** (pre-declaration).
  - **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).

## Deadlock Prevention (Cont.)

- **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 a lock
- **wound-wait** scheme — preemptive
  - Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it.
  - Younger transactions may wait for older ones.
  - Fewer rollbacks than *wait-die* scheme.
- 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 thus avoided.



## Deadlock Prevention (Cont.)

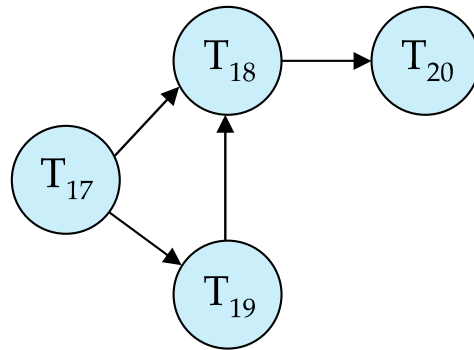
### ■ Timeout-Based Schemes:

- 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

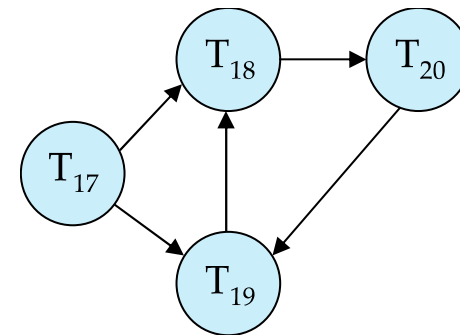
# Deadlock Detection

## ▪ 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 victim transaction only as far as necessary to release locks that another transaction in cycle is waiting for
- Starvation can happen (why?)
  - One solution: oldest transaction in the deadlock set is never chosen as victim





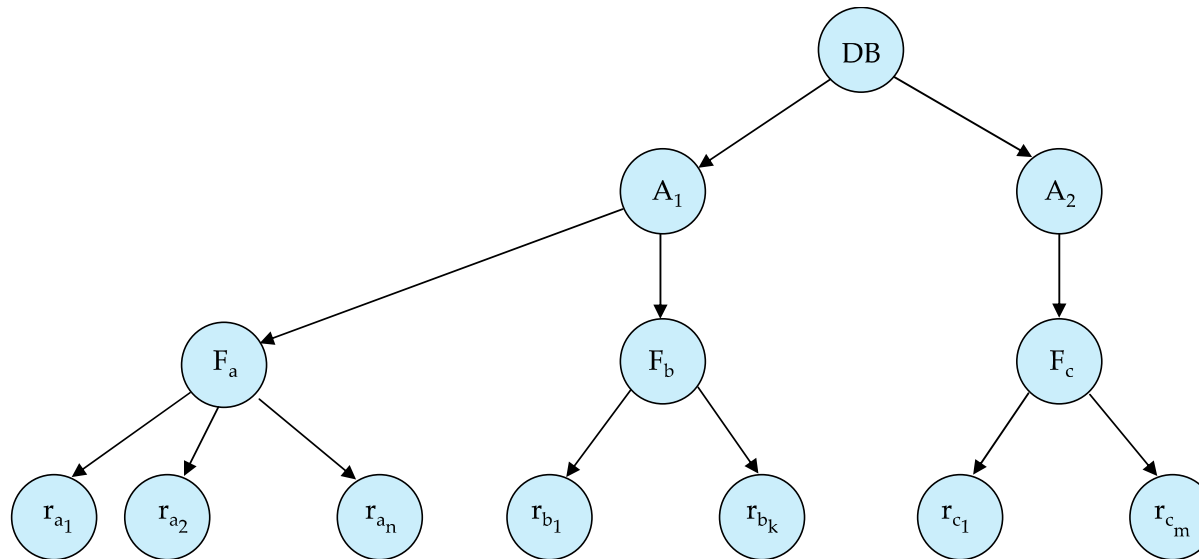
# Multiple Granularity

- 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 (but don't confuse with tree-locking protocol)
- 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

- In addition to S and X lock modes, there are 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

- 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.
- Lock granularity escalation:** in case there are too many locks at a particular level, switch to higher granularity S or X lock



## Insert/Delete Operations and Predicate Reads

- Locking rules for insert/delete operations
  - An exclusive lock must be obtained on an item before it is deleted
  - A transaction that inserts a new tuple into the database automatically given an X-mode lock on the tuple
- Ensures that
  - reads/writes conflict with deletes
  - Inserted tuple is not accessible by other transactions until the transaction that inserts the tuple commits

# Phantom Phenomenon

- Example of **phantom phenomenon**.
  - A transaction T1 that performs **predicate read** (or scan) of a relation
    - **select count(\*)**  
**from** *instructor*  
**where** *dept\_name* = 'Physics'
  - and a transaction T2 that inserts a tuple while T1 is active but after predicate read
    - **insert into instructor values** ('11111', 'Feynman', 'Physics', 94000)
  - (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 instructor, but may read some other tuple written by the update transaction
- Can also occur with updates
  - E.g. update Wu's department from Finance to Physics

# Insert/Delete Operations and Predicate Reads

- **Another Example:**
  - not possible in serializable schedule

T1	T2
Read(instructor where dept_name='Physics')	Insert Instructor in Physics
	Insert Instructor in Comp. Sci.
	Commit
Read(instructor where dept_name='Comp. Sci.')	





# Handling Phantoms

- There is a conflict at the data level
  - The transaction performing predicate read or scanning the relation is reading information that indicates what tuples the relation contains
  - The transaction inserting/deleting/updating a tuple updates the same information.
  - The conflict should be detected, e.g. by locking the information.
- 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 To Prevent Phantoms

- **Index locking protocol** to prevent phantoms
  - Every relation must have at least one index.
  - A transaction can access tuples only after finding them through one or more indices on the relation
  - A transaction  $T_i$  that performs a lookup must **lock all the index leaf nodes that it accesses**, in S-mode
    - Even if the leaf node does not contain any tuple satisfying the index lookup (e.g. for a range query, no tuple in a leaf is in the range)
  - A transaction  $T_i$  that inserts, updates or deletes a tuple  $t_i$  in a relation  $r$ 
    - Must update all indices to  $r$
    - Must obtain **exclusive locks on all index leaf nodes** affected by the insert/update/delete
  - The rules of the **two-phase locking protocol** must be observed
- ***Guarantees that phantom phenomenon won't occur***

## Next-Key Locking to Prevent Phantoms

- Index-locking protocol to prevent phantoms locks entire leaf node
  - Can result in poor concurrency if there are many inserts
- **Next-key locking protocol:** provides higher concurrency
  - Lock all values that satisfy index lookup (match lookup value, or fall in lookup range)
  - Also lock next key value in index
    - even for inserts/deletes
  - Lock mode: S for lookups, X for insert/delete/update
- Ensures detection of query conflicts with inserts, deletes and updates

Consider B+-tree leaf nodes as below, with query predicate  $7 \leq X \leq 16$ .

Check what happens with next-key locking when inserting: (i) 15 and (ii) 7

