Concurrency Control

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.

- 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.

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 — if A and B get updated in-between the read of A and B, the displayed sum would be wrong.

- 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)	

- 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).

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.
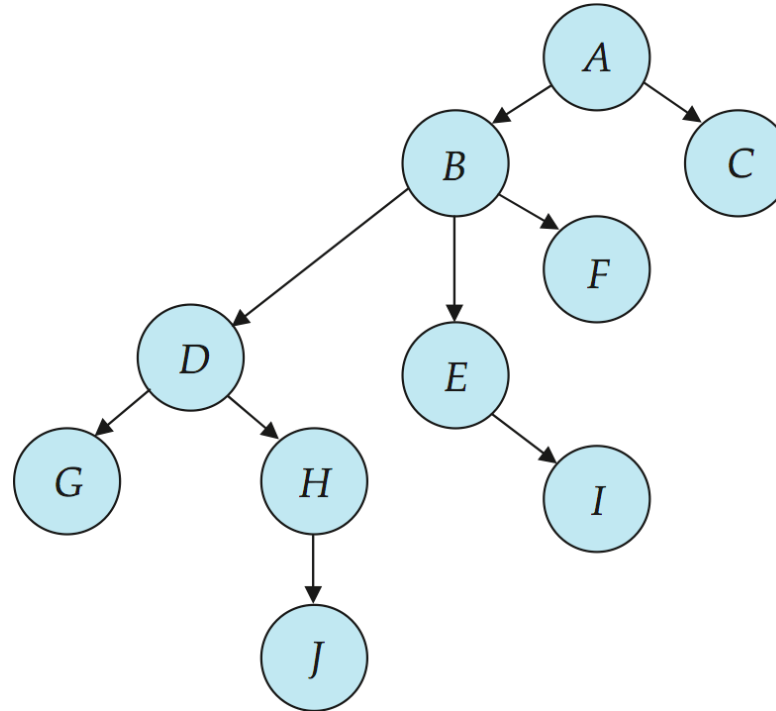
Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking.

- Impose a partial ordering $\rightarrow$ on the set **D** = {$d_1, d_2, ..., d_h$} 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

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.

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

- Schedules not possible under two-phase locking are possible under tree protocol, and vice versa.

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 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.

■ **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 (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) of 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.
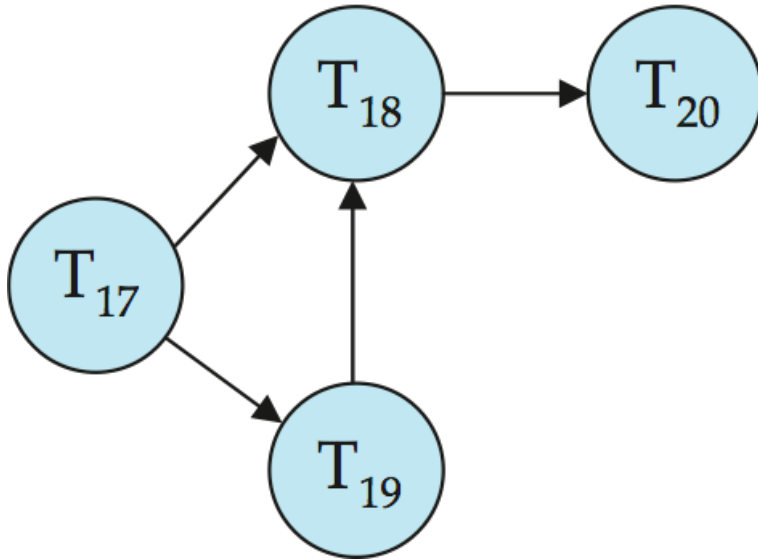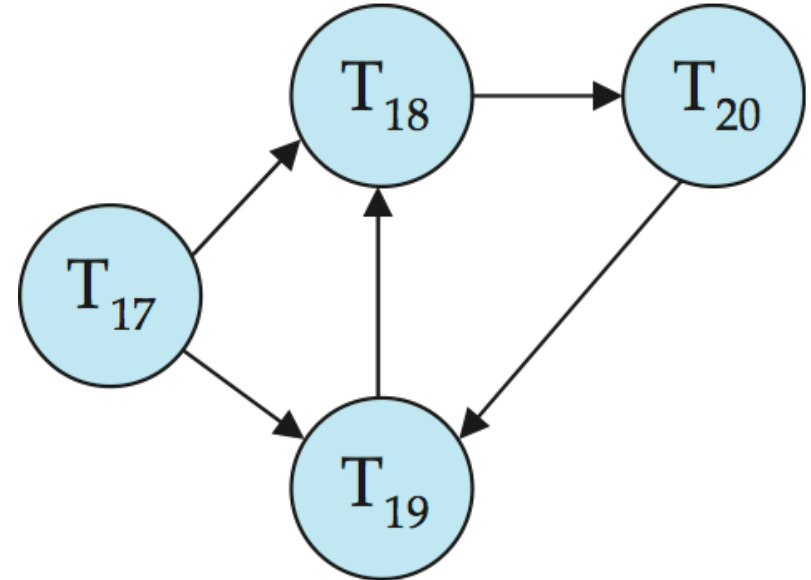
Deadlock Detection

- 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.

- 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.

    - 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
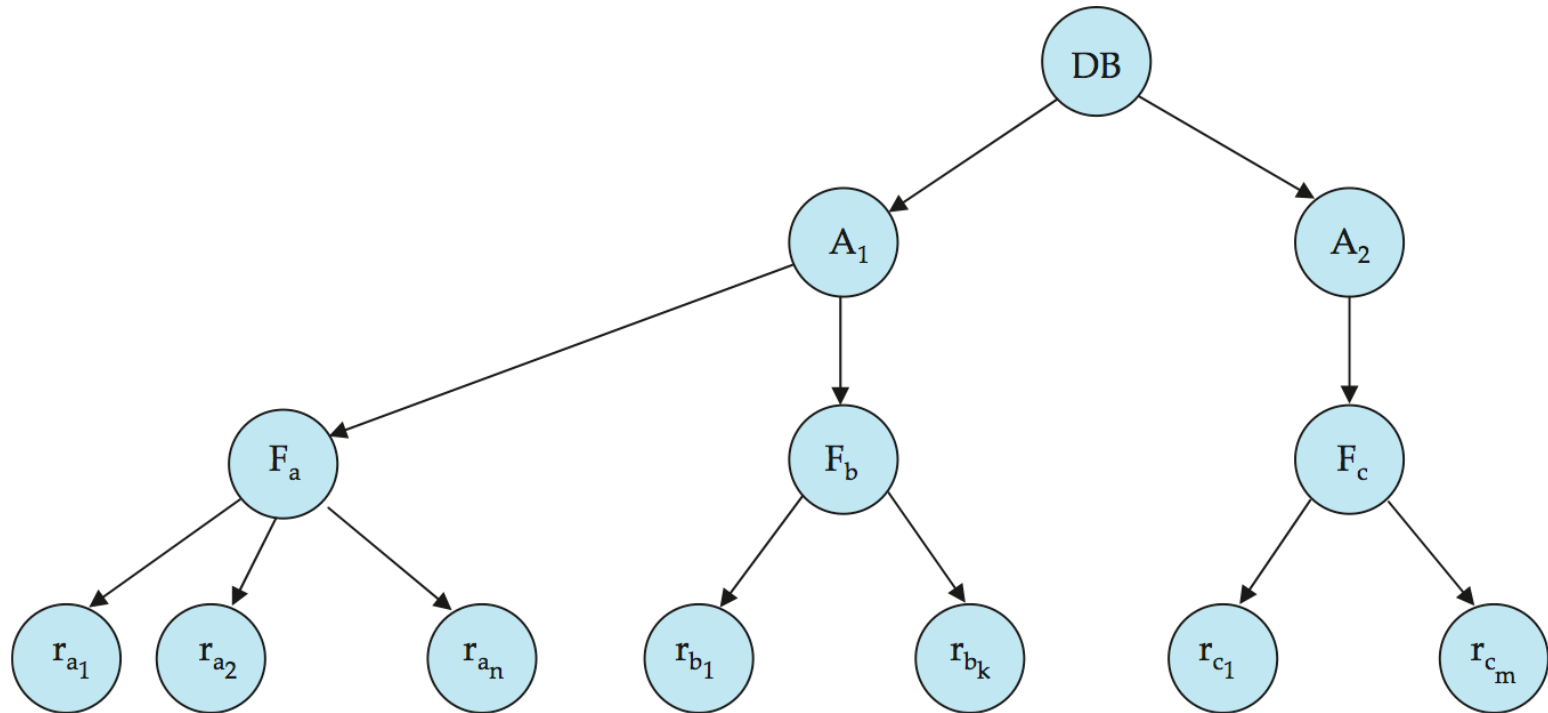
Multiple Granularity

- 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 descendents 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*

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.