CONCURRENCY CONTROL

Engr. Laraib Siddiqui

Concurrency

The term concurrency refers to the fact that DBMSs typically allow many transactions to access the same database at the same time.

In such system, some kind of control mechanism is clearly need to ensure that concurrent transactions do not interfere with each other.

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.

The potential for deadlock exists in most locking protocols.

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.

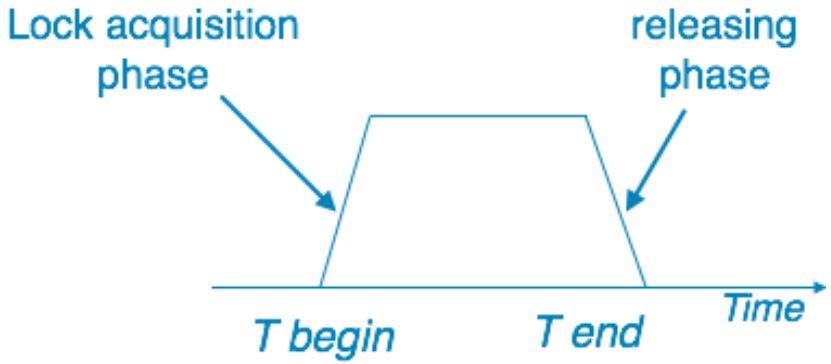
The Two-Phase Locking Protocol

Two phase locking protocol is one in which there are 2 phases that a transaction goes through. The first is the growing phase in which it is acquiring locks, the second phase is shrinking phase in which it is releasing locks. Once you have released a lock, you cannot acquire any more locks.

This protocol ensures a serializable schedule.

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

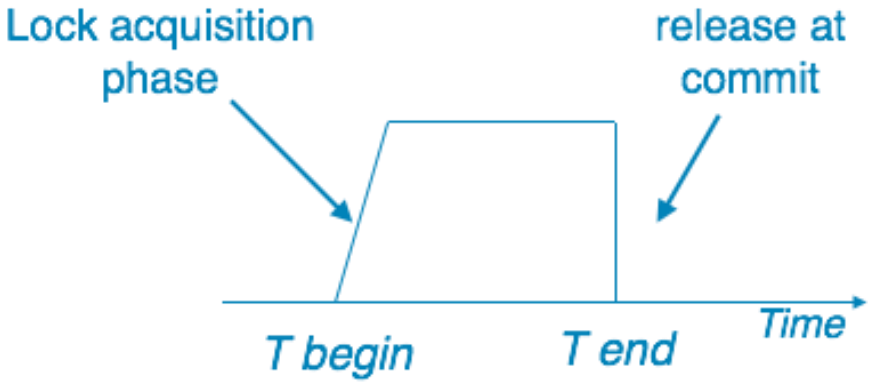
Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.



Strict Two-Phase Locking

The first phase of Strict-2PL is same as 2PL. After acquiring all the locks in the first phase, the transaction continues to execute normally. But in contrast to 2PL, Strict-2PL does not release a lock after using it. Strict-2PL holds all the locks until the commit point and releases all the locks at a time.

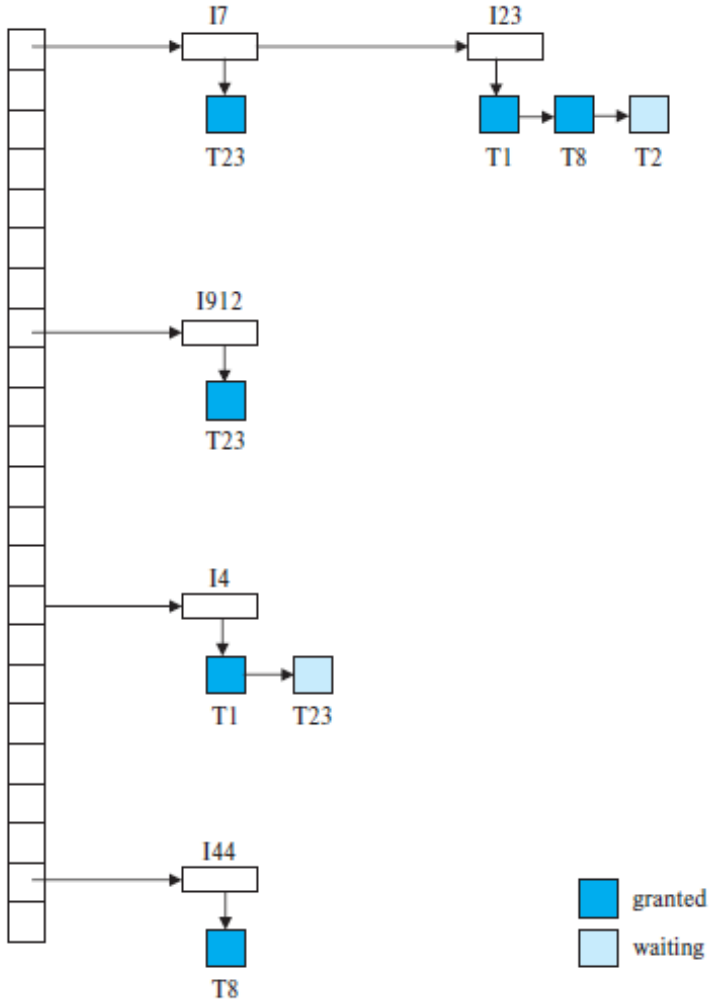
Strict-2PL does not have cascading abort as 2PL does.



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
- Each record of the linked list for a data item notes which transaction made the request, and what lock mode it requested.
- The record also notes if the request has currently been granted.

Lock Table



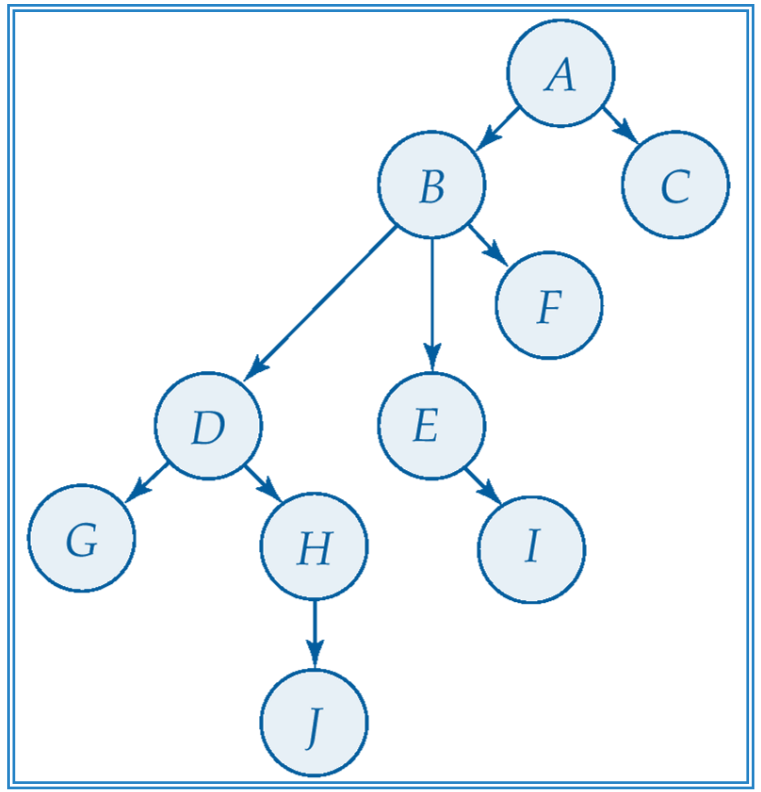
- When a lock request message arrives, it adds a record to the end of the linked list for the data item or creates a new linked list.
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible 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 **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 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
  - Transactions may have to lock data items that they do not access.
    - increased locking overhead, and additional waiting time
    - potential decrease in concurrency

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.

- The wait–die scheme is a nonpreemptive technique. When transaction ***Ti*** requests a data item currently held by ***Tj***, ***Ti*** is allowed to wait only if it has a timestamp smaller than that of ***Tj*** (i.e., ***Ti*** is older than ***Tj***). Otherwise, ***Ti*** is rolled back (dies).
  - 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.

- The wound–wait scheme is a preemptive technique. When transaction ***Ti*** requests a data item currently held by ***Tj***, ***Ti*** is allowed to wait only if it has a timestamp larger than that of *Tj* (i.e., *Ti* is younger than *Tj*). Otherwise, *Tj* is rolled back.
  - Example, with transactions ***T14***, ***T15***, and **T16**, 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.

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.

Timestamp-Based Protocols

The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order.

- **Timestamps** – Tell the order of arrival into system. We associate a unique fixed timestamp, denoted by TS(*Ti*). This timestamp is assigned by the database system before the transaction *Ti* starts execution.
- **Write-timestamp** (WTS) - denotes the largest/latest timestamp of any transaction that executed write(*Q*) successfully.
- **Read-timestamp** (*RTS*) - denotes the largest/latest timestamp of any transaction that executed read(*Q*) successfully.

Timestamp-Based Protocols (Cont.)

- Suppose a transaction Ti issues a read(Q)
  - If WTS(Q) > TS(Ti) , then rollback.
  - Otherwise execute R(Q) operation. Set RTS(Q) = Max { RTS(Q), TS(Ti) }
- Suppose a transaction Ti issues a write(Q)
  - If RTS(Q) > TS(Ti), then rollback.
  - If WTS(Q) > TS(Ti), then rollback.
  - Otherwise execute write(Q) operation. Set WTS(Q) = TS(Ti).

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