# Chapter 15 : Concurrency Control

Edited by Radhika Sukapuram

Database System Concepts, 6th Ed.

# 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.)

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

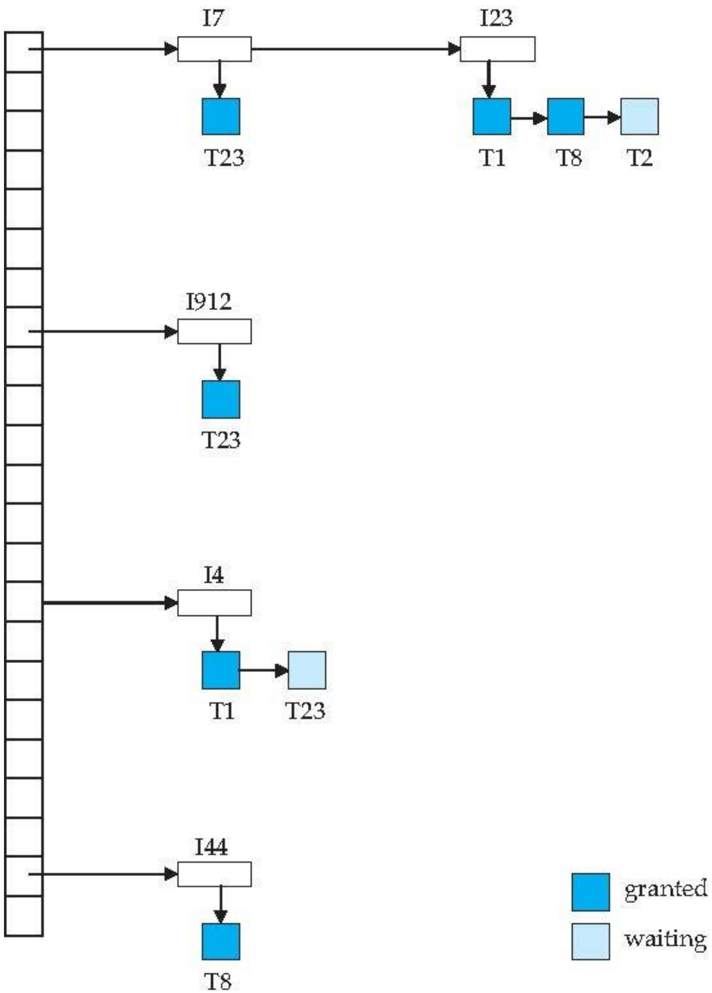
  **if** $T_i$ has a **lock-X** on *D*<br>
    **then**<br>
      write(*D*)<br>
   **else begin**<br>
       if necessary wait until no other transaction has any lock on *D*,<br>
       if $T_i$ has a **lock-S** on *D*<br>
         **then**<br>
           **upgrade** lock on *D* to **lock-X**<br>
        **else**<br>
           grant $T_i$ a **lock-X** on *D*<br>
        write(*D*)<br>
    **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 message (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




- Dark blue rectangles indicate granted locks; light blue 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

# Deadlock Handling

- System is deadlocked if there exists a set of waiting transactions $\{T_0, T_1, \ldots, T_n\}$ such that $T_0$ is waiting for an item that $T_1$ holds, $T_1$ is waiting for an item that $T_2$ holds , and ..., $T_{n-1}$ is waiting for a data item that $T_n$ holds and $T_n$ is waiting for a data item that $T_0$ holds.
- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state. A prevention strategy :
  - Require that each transaction locks all its data items before it begins execution (pre-declaration). All items are locked in one step or none are locked
    - Hard to know all data items before-hand
    - Data-item utilization may be low as they are locked for a long time

# More Deadlock Prevention Strategies

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.
- $T_i$ and $T_k$ request a data item held by $T_j$
- **wait-die** scheme — non-preemptive
  - i < j. older transaction waits for younger one to release data item
  - older means smaller timestamp
  - k > j : younger transactions never wait for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring the needed data item
- **wound-wait** scheme — preemptive
  - i < j: older transaction *wounds* (forces rollback of) a younger transaction instead of waiting for it.
  - k > j: younger transactions wait for older ones.
- Locking is used for concurrency control

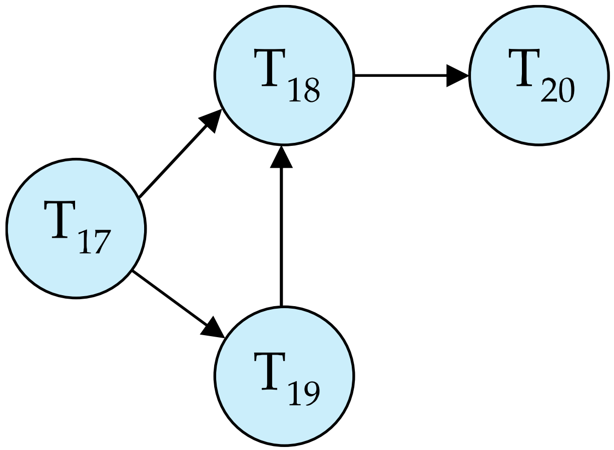
# Deadlock prevention (Cont.)

- Both in *wait-die* and in *wound-wait* schemes
  - a rolled back transaction is restarted with its original timestamp. Older transactions thus have precedence over newer ones,
    - starvation is hence avoided.
- **Timeout-Based Schemes:**
  - A transaction waits for a lock only for a specified amount of time.
  - If the lock has not been granted within that time, the transaction is rolled back and restarted
    - Thus, deadlocks are not possible
    - Simple to implement, but starvation is possible.
    - Also difficult to determine a good value for the timeout interval.

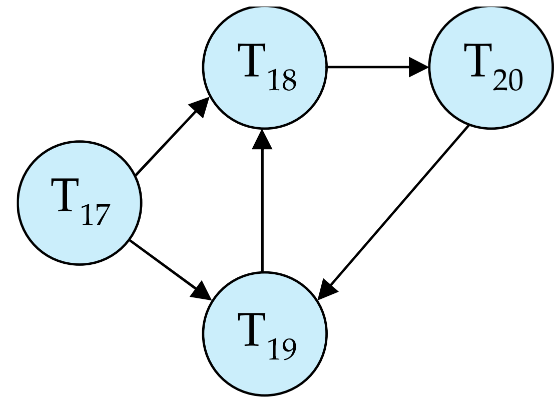
# Deadlock Detection

- 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 $T_i \rightarrow T_j$ 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

# Order of transactions

- Determine the order between transactions at execution time by the first lock that the transactions request in incompatible mode
- Select an ordering among transactions in advance. How?

# 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.
- 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.)

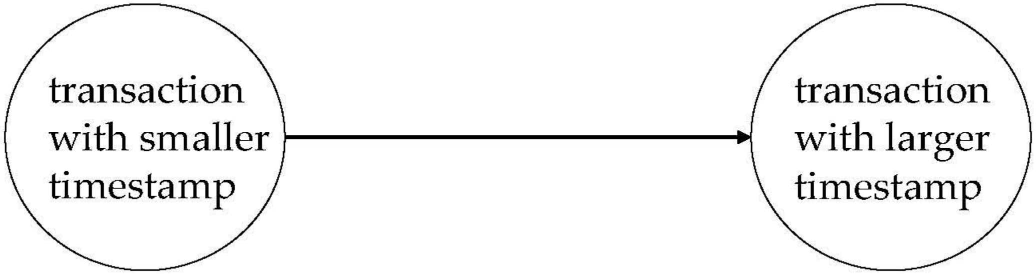
- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction $T_i$ issues a **read**($Q$)
  1. If TS($T_i$) < W-timestamp($Q$), then $T_i$ needs to read a value of $Q$ that was already overwritten.
     - Hence, the **read** operation is rejected, and $T_i$ is rolled back.
  2. If TS($T_i$) $\geq$ W-timestamp($Q$), then the **read** operation is executed, and R-timestamp($Q$) is set to **max**(R-timestamp($Q$), TS($T_i$)).

# Timestamp-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 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 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($T_i$).
- If a transaction is rolled back as a result of a read or write operation, the system assigns it a new timestamp and restarts it

# Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees conflict 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.

# Recoverability and Cascade Freedom

- Problem with timestamp-ordering protocol:
  - Suppose $T_i$ aborts, but $T_j$ has read a data item written by $T_i$
  - Then $T_j$ must abort; if $T_j$ had been allowed to commit earlier, the schedule is not recoverable.
  - Further, any transaction that has read a data item written by $T_j$ must abort
  - This can lead to cascading rollback --- that is, a chain of rollbacks

# One solution : Limited form of locking

- Wait for data to be committed before reading it
- A commit bit associated with each data item Q, C(Q) is set if and only if the most recent transaction to write Q has already committed
- 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($T_i$) $\geq$ W-timestamp(*Q*), then
     1. If C(Q) is false, the transaction waits until it becomes true or the transaction that wrote Q aborts
     2. If C(Q) is true, the **read** operation is executed, and R-timestamp(*Q*) is set to **max**(R-timestamp(*Q*), TS($T_i$)).
- Scheduler receives a request to commit $T_i$, C(Q) is set to true.
- Scheduler receives a request to abort $T_i$ or decides to rollback $T_i$
  - Any transaction waiting for this must repeat its attempt to read 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 write 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.
  - Allows some view-serializable schedules that are not conflict-serializable.
  - $read_{27}(Q), write_{28}(Q), write_{27}(Q)$
  - This schedule is not possible under 2PL or basic time-stamp ordering

# Limited form of locking with Thomas' Write Rule

- Rules for checking for Commit bit for read(Q) apply
- 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 an obsolete value of $Q$.
     - If C(Q) is true, the write is ignored and $T_i$ proceeds
     - If C(Q) is false, $T_i$ waits until C(Q) becomes true or the transaction that wrote Q aborts
  3. Otherwise, the **write** operation is executed, and W-timestamp($Q$) is set to TS($T_i$), C(Q) is set to false.
- Scheduler receives a request to commit $T_i$, C(Q) is set to true.
- Scheduler receives a request to abort $T_i$ or decides to rollback $T_i$
  - Any transaction waiting for this must repeat its attempt to read or write Q

Note: From Garcia-Molina et al

# End of Chapter 15