The picture can't be displayed.

# Chapter 15 : Concurrency Control

**Database System Concepts, 6th Ed.**

# Chapter 15: Concurrency Control

- Lock-Based Protocols

- Timestamp-Based Protocols

- Validation-Based Protocols

- Multiple Granularity

- Multiversion Schemes

- Insert and Delete Operations

- Concurrency in Index Structures

# 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.

$T_1$: lock-x($B$);
  read($B$);
  $B := B - 50$;
  write($B$);
  unlock($B$);
  lock-x($A$);
  read($A$);
  $A := A + 50$;
  write($A$);
  unlock($A$).

**Figure 15.2**  Transaction $T_1$.

$T_2$: lock-s($A$);
read($A$);
unlock($A$);
lock-s($B$);
read($B$);
unlock($B$);
display($A + B$).

**Figure 15.3** Transaction $T_2$.

$$T_3: \text{lock-x}(B);$$
$$\text{read}(B);$$
$$B := B - 50;$$
$$\text{write}(B);$$
$$\text{lock-x}(A);$$
$$\text{read}(A);$$
$$A := A + 50;$$
$$\text{write}(A);$$
$$\text{unlock}(B);$$
$$\text{unlock}(A).$$

**Figure 15.5** Transaction $T_3$ (transaction $T_1$ with unlocking delayed).

| $T_1$ | $T_2$ | concurreny-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) | | |

**Figure 15.4** Schedule 1.

$T_4$: lock-s(A);
        read(A);
        lock-s(B);
        read(B);
        display(A + B);
        unlock(A);
        unlock(B).

**Figure 15.6** Transaction $T_4$ (transaction $T_2$ with unlocking delayed).

# 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).

| T1 | T2 |
|---|---|
| Lock S(A) | |
| | Lock S(A) |
| Lock X(B) | |
| Unlock(A) | |
| | Lock S(D) |
| Unlock(B) | |
| | Unlock(A) |
| | Unlock (D) |

# The Two-Phase Locking Protocol (Cont.)

- Two-phase locking *does not* ensure freedom
  - Irrecoverabiltity
  - Deadlocks
  - cascading roll backs

- 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.

- **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

# The Two-Phase Locking Protocol (Cont.)

- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.

- However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:

  Given a transaction $T_i$ that does not follow two-phase locking, we can find a transaction $T_j$ that uses two-phase locking, and a schedule for $T_i$ and $T_j$ that is not conflict serializable.

# Lock Conversions

- **Two-phase locking with lock conversions:**
    - First 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)
    - Second Phase:
        - can release a lock-S
        - can release a lock-X
        - can convert a lock-X to a lock-S (downgrade)
- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.

# 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
    **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 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.

# Lock Table
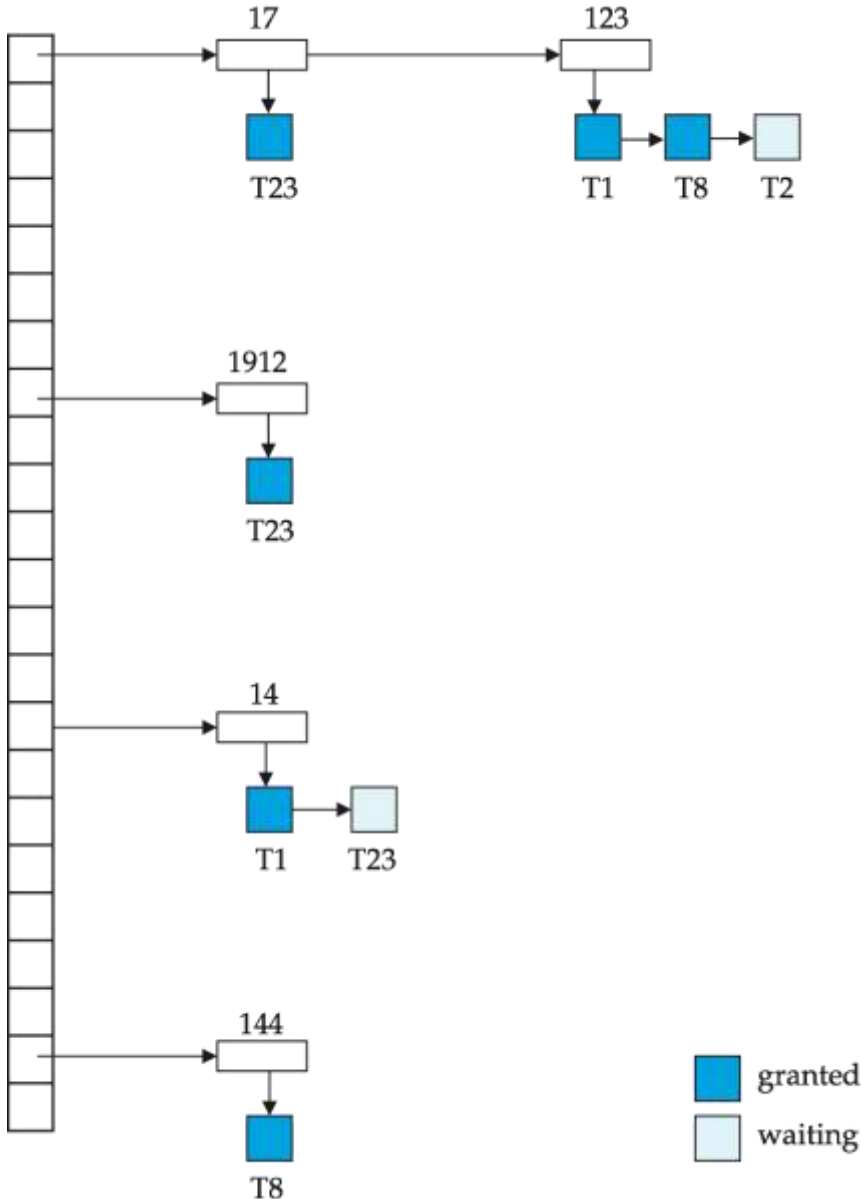


- Black rectangles indicate granted locks, white ones indicate waiting requests

- 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 **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
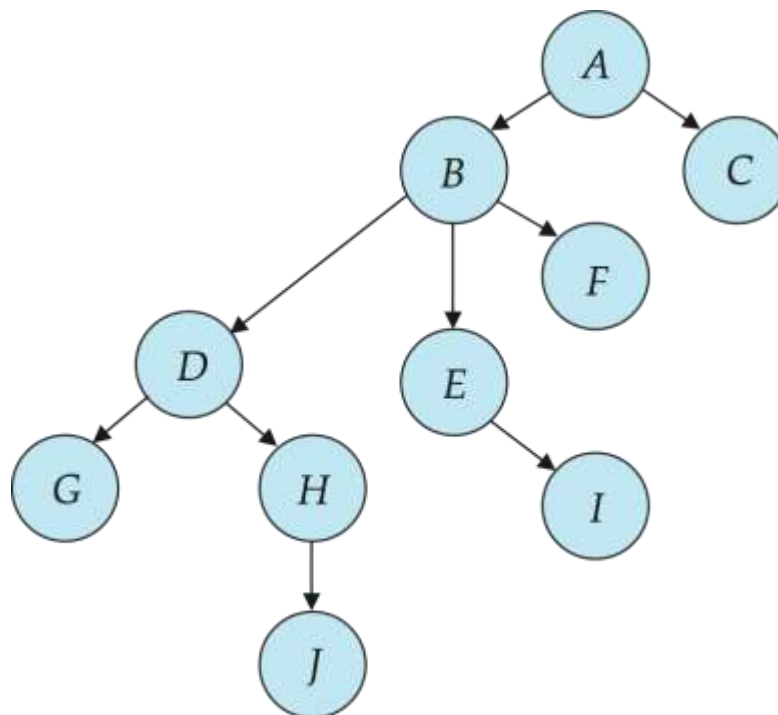


1. Only exclusive locks are allowed.

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

- 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 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<br>write (A) | |
| | **lock-X** on B<br>write (B)<br>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).
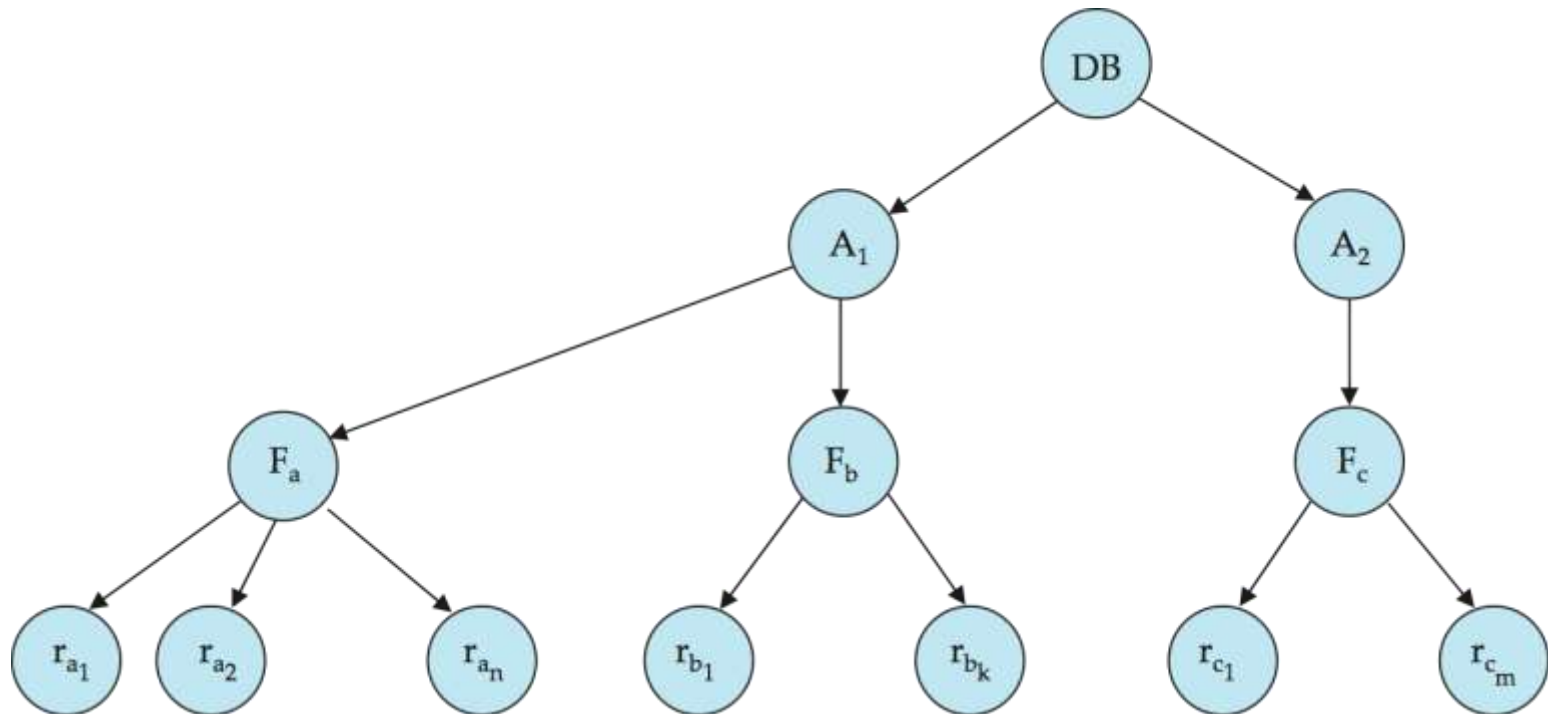
# Multiple Granularity

- 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*

# 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.

# Timestamp-Based Protocols

- Unique value assigned to every transaction

- Tells order when transaction enters in system

- RTS

- WTS

# Implementing TS

1. Use the value of the system clock as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.

2. Use a logical counter that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

# 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.

# 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 $TS(T_i) \leq$ **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 $TS(T_i)$.

$T_{25}$: read($B$);
read($A$);
display($A + B$).

$T_{26}$: read($B$);
$B := B - 50$;
write($B$);
read($A$);
$A := A + 50$;
write($A$);
display($A + B$).

| $T_{25}$ | $T_{26}$ |
|---|---|
| read($B$) | |
| | read($B$) |
| | $B := B - 50$ |
| | write($B$) |
| read($A$) | |
| | read($A$) |
| display($A + B$) | |
| | $A := A + 50$ |
| | write($A$) |
| | display($A + B$) |

**Figure 15.17** Schedule 3.

# Example Use of the Protocol

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

| $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ |
|---|---|---|---|---|
| | | | | read (X) |
| | read (Y) | | | |
| read (Y) | | | | |
| | | write (Y) | | |
| | | write (Z) | | |
| | | | | read (Z) |
| | read (Z) | | | |
| | abort | | | |
| read (X) | | | | |
| | | | read (W) | |
| | | write (W) | | |
| | | abort | | |
| | | | | write (Y) |
| | | | | write (Z) |

# Correctness of Timestamp-Ordering Protocol

■ The timestamp-ordering protocol guarantees 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

- Solution 1:

  - A transaction is structured such that its writes are all performed at the end of its processing

  - All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written

  - A transaction that aborts is restarted with a new timestamp

- Solution 2: Limited form of locking: wait for data to be committed before reading it

- Solution 3: Use commit dependencies to ensure recoverability

# 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 $TS(T_i) < W\text{-timestamp}(Q)$, 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.

# 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

# Validation-Based Protocol (Cont.)

- Each transaction $T_i$ has 3 timestamps:

  - Start($T_i$) : the time when $T_i$ started its execution

  - Validation($T_i$): the time when $T_i$ entered its validation phase

  - Finish($T_i$) : the time when $T_i$ finished its write phase

- Serializability order is determined by timestamp given at validation time, to increase concurrency.

  - Thus TS($T_i$) is given the value of Validation($T_i$).

- This protocol is useful and gives greater degree of concurrency if probability of conflicts is low.

  - because the serializability order is not pre-decided, and

  - relatively few transactions will have to be rolled back.

# Validation Test for Transaction $T_j$

- If for all $T_i$ with TS ($T_i$) < TS ($T_j$) either one of the following condition holds:

  - **finish**($T_i$) < **start**($T_j$)

  - **start**($T_j$) < **finish**($T_i$) < **validation**($T_j$) **and** the set of data items written by $T_i$ does not intersect with the set of data items read by $T_j$.

  then validation succeeds and $T_j$ can be committed.  Otherwise, validation fails and $T_j$ is aborted.

- *Justification*:  Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and

  - the writes of $T_j$ do not affect reads of $T_i$ since they occur after $T_i$ has finished its reads.

  - the writes of $T_i$ do not affect reads of $T_j$ since $T_j$ does not read any item written by $T_i$.

# Schedule Produced by Validation

- Example of schedule produced using validation

| $T_{25}$ | $T_{26}$ |
|---|---|
| read $(B)$ | |
| | read $(B)$ |
| | $B := B - 50$ |
| | read $(A)$ |
| | $A := A + 50$ |
| read $(A)$ | |
| $\langle$ validate $\rangle$ | |
| display $(A + B)$ | |
| | $\langle$ validate $\rangle$ |
| | write $(B)$ |
| | write $(A)$ |