CMSC 461, Database Management Systems
Spring 2018

# Lecture 22 – Concurrency Control Part 2

These slides are based on "Database System Concepts" 6th edition book (whereas some quotes and figures are used from the book) and are a modified version of the slides which accompany the book (http://codex.cs.yale.edu/avi/db-book/db6/slide-dir/index.html), in addition to the 2009/2012 CMSC 461 slides by Dr. Kalpakis

Dr. Jennifer Sleeman

https://www.csee.umbc.edu/~jsleem1/courses/461/spr18

# Logistics

- Phase 4 due 4/30/2018
- Homework 6 due 5/2/2018
- Final Project Plan 5/14/2018

Reminder: Presentation Slots

# Concurrency Control

Why do we need it?



Based on and image from "Database System Concepts" book and slides, 6th edition

# Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
  - *exclusive* *(X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  - *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.



Based on and image from "Database System Concepts" book and slides, 6th edition

# Lock-Based Protocols

## 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



Based on and image from "Database System Concepts" book and slides, 6th edition

# Lock-Based Protocols

- 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.

| | S | X |
|---|---|---|
| S | true | false |
| X | false | false |

Based on and image from "Database System Concepts" book and slides, 6th edition

# Lock-Based Protocols

What is a common problem we have with locking?

What happens to a transaction when it is starved?



Based on and image from "Database System Concepts" book and slides, 6th edition

# 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 ensures 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).



Based on and image from "Database System Concepts" book and slides, 6th edition

# The Two-Phase Locking Protocol

- 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.
- **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.

# What is a cascadeless schedule?

# The Two-Phase Locking Protocol

- 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



Based on and image from "Database System Concepts" book and slides, 6th edition

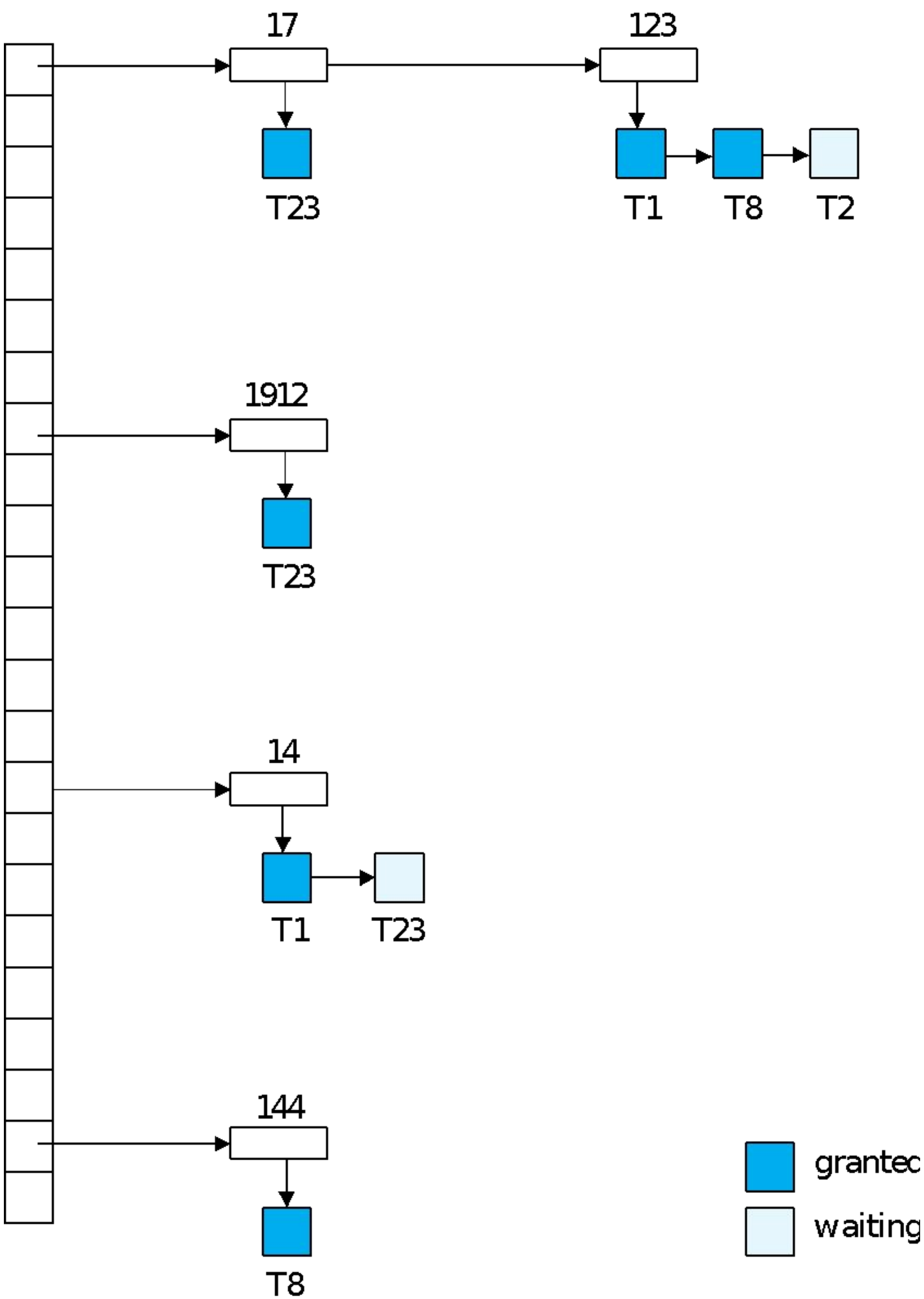
# Implementation of Locking

- 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 rollback, in case of  a deadlock)
- The requesting transaction waits until its request is answered

# Implementation of Locking

- 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

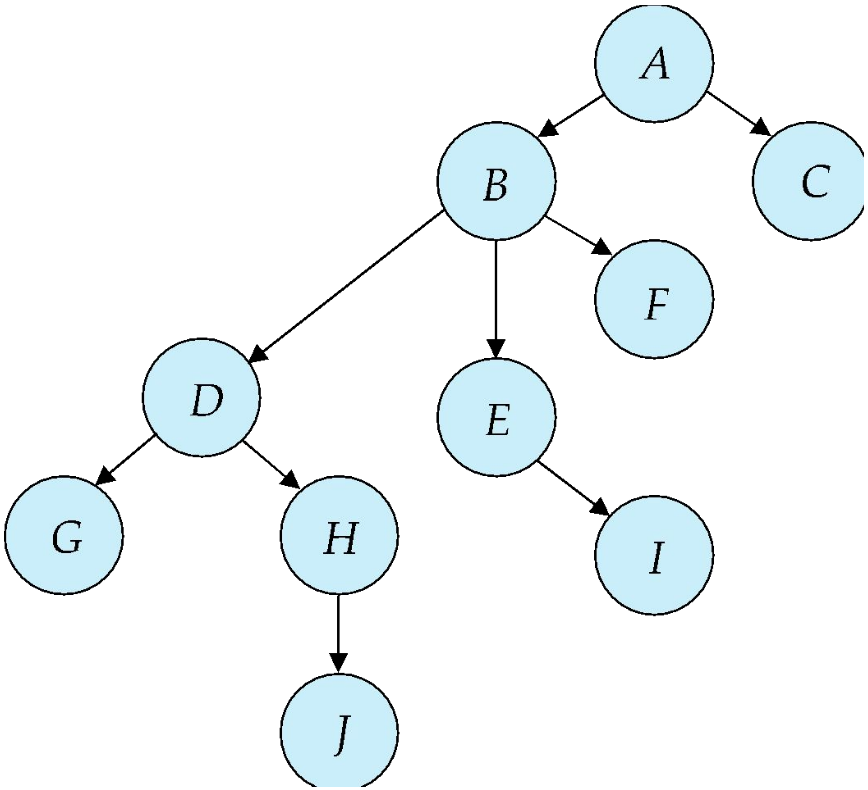
Based on and image from "Database System Concepts" book and slides, 6th edition

# 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

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 $T_i$






Based on and image from "Database System Concepts" book and slides, 6th edition

# Graph-Based Protocols

- 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

# Graph-Based Protocols

- 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.



Based on and image from "Database System Concepts" book and slides, 6th edition

# Deadlock Handling

- Consider the following two transactions:

  $T_1$: write ($X$) write($Y$)

  $T_2$: 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 | |



Based on and image from "Database System Concepts" book and slides, 6th edition

# 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 | Wound/Wait |
|---|---|---|
| O needs a resource held by Y | O waits | Y dies |
| Y needs a resource held by O | Y dies | Y waits |

- **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.



Based on and image from "Database System Concepts" book and slides, 6th edition

# More Deadlock Prevention Strategies

- 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.



Based on and image from "Database System Concepts" book and slides, 6th edition

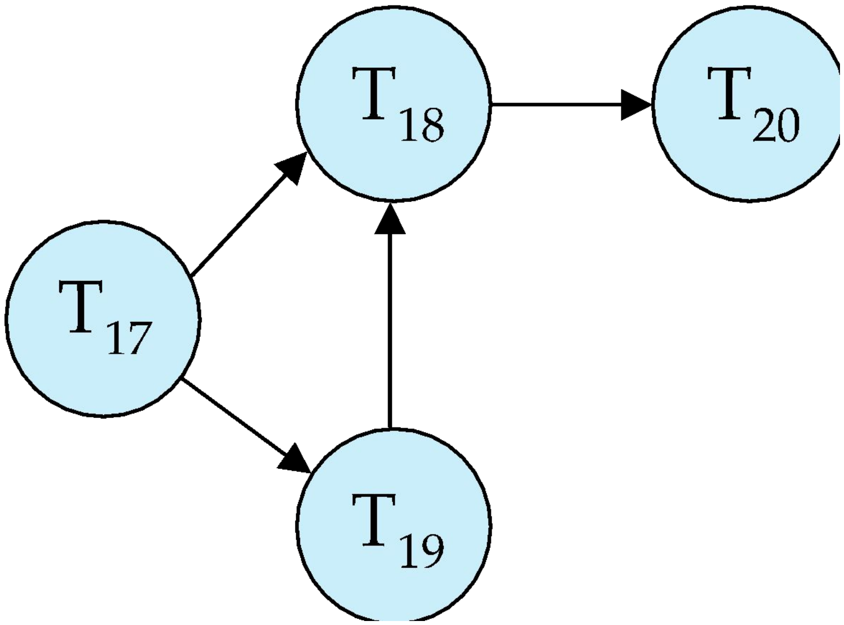
# Deadlock Detection

- 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.

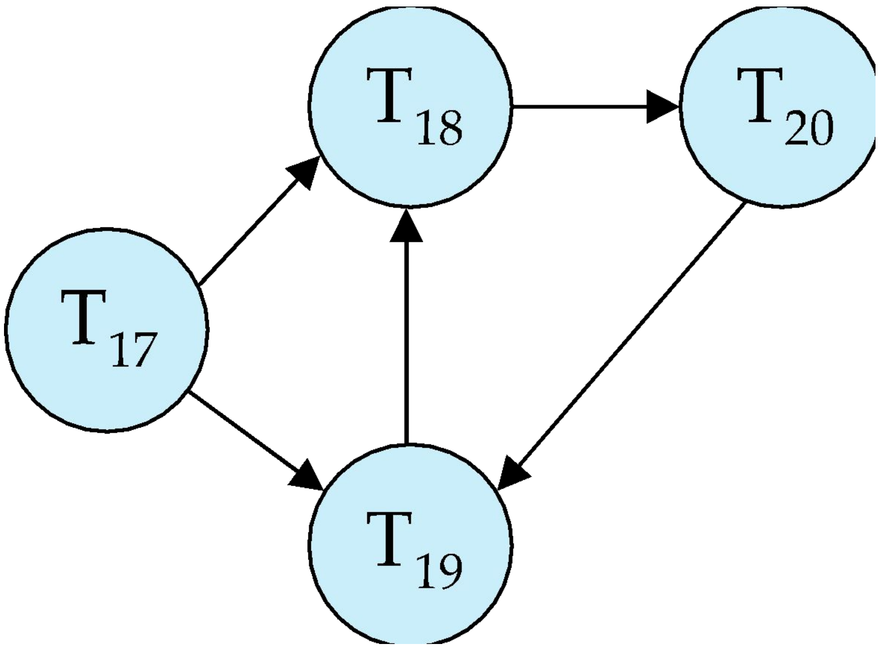
# Deadlock Detection

- 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.

# Is there a deadlock?



# Is there a deadlock?



# Deadlock Recovery

- When a 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

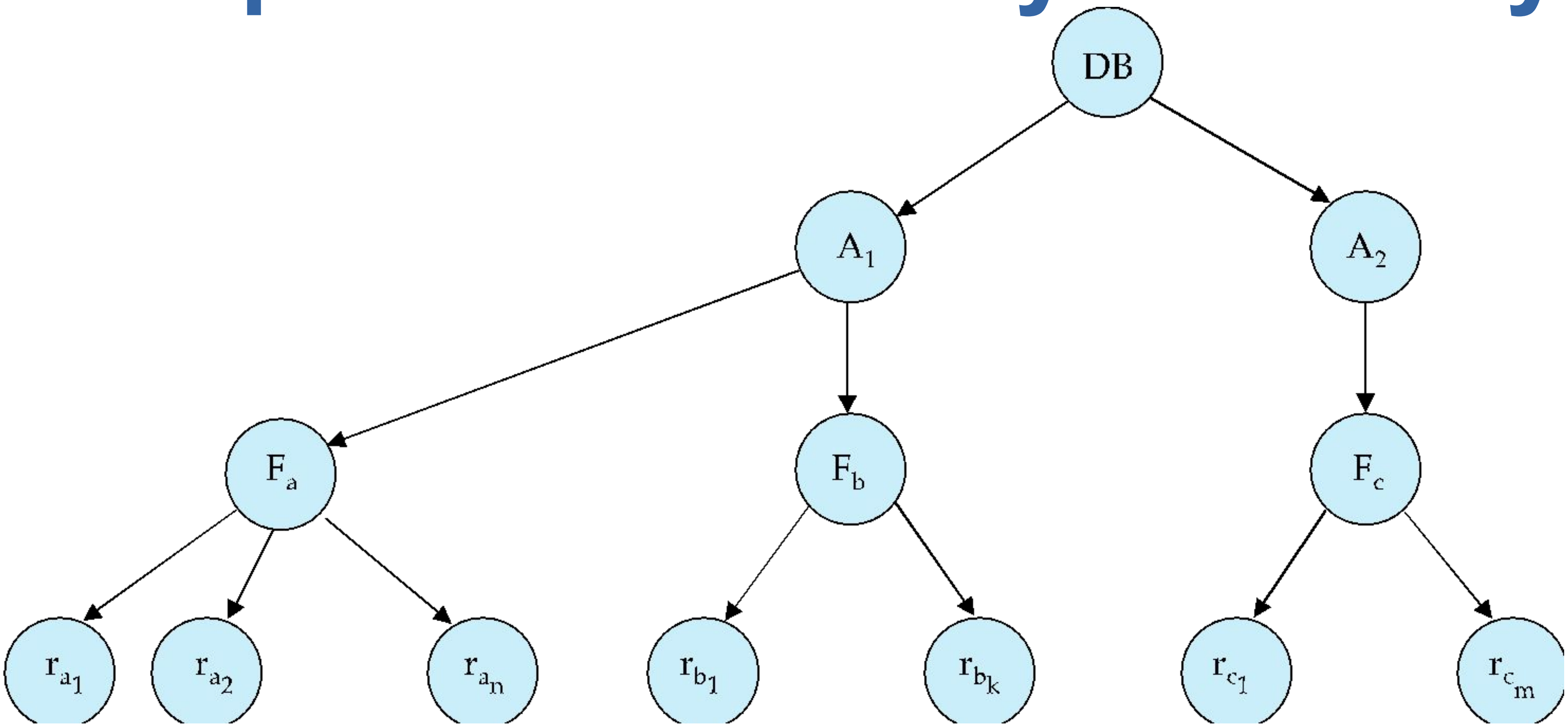
# Multiple Granularity

- 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)

# Multiple Granularity

- 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*



Based on and image from "Database System Concepts" book and slides, 6th edition

# 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 descendant nodes.



Based on and image from "Database System Concepts" book and slides, 6th edition

# 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 |



Based on and image from "Database System Concepts" book and slides, 6th edition

# 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

$$\text{TS}(T_i) < \text{TS}(T_j)$$

- The protocol manages concurrent execution such that the time-stamps determine the serializability order.

# Timestamp-Based Protocols

- 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

- 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**($Q$)
  - 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.
  - If TS($T_i$)≥ **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

- Suppose that transaction $T_i$ issues **write**(*Q*).
  - If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was needed previously, and the system assumed that that value would never be produced.
    - Hence, the write operation is rejected, and Ti is rolled back.
  - If TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of Q.
    - Hence, this write operation is rejected, and Ti is rolled back.
  - Otherwise, the  write operation is executed, and W-timestamp(Q) is set to TS(Ti).

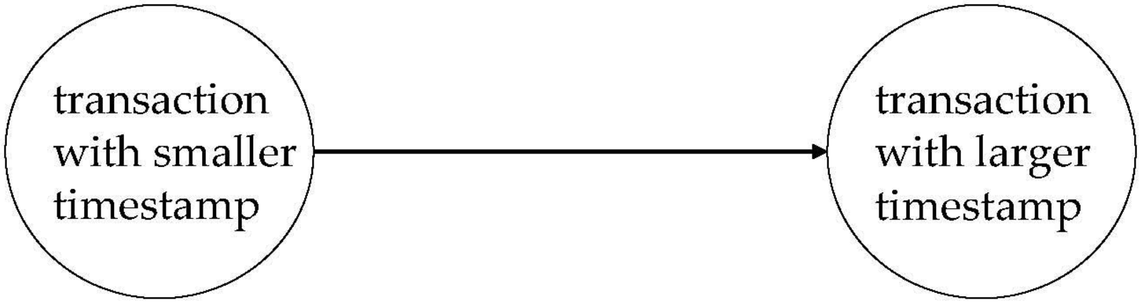
# Example Use of the Protocol

- 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.

# 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.

# 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.

# Validation-Based Protocol

- 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($T_i$).

# Validation-Based Protocol

- 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.

# Schedule Produced by Validation

Example of schedule produced using validation

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

Based on and image from "Database System Concepts" book and slides, 6th edition

# Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency.
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- Each successful **write** results in the creation of a new version of the data item written.
- Use timestamps to label versions.

# Multiversion Schemes

- When a **read**(*Q*) operation is issued, select an appropriate version of *Q* based on the timestamp of the transaction, and return the value of the selected version.
- **read**s never have to wait as an appropriate version is returned immediately.

# Multiversion Timestamp Ordering

- Each data item $Q$ has a sequence of versions <$Q_1$, $Q_2$,...., $Q_m$>. Each version $Q_k$ contains three data fields:
  - **Content** -- the value of version $Q_k$.
  - **W-timestamp**($Q_k$) -- timestamp of the transaction that created (wrote) version $Q_k$
  - **R-timestamp**($Q_k$) -- largest timestamp of a transaction that successfully read version $Q_k$
- when a transaction $T_i$ creates a new version $Q_k$ of $Q$, $Q_k$'s W-timestamp and R-timestamp are initialized to TS($T_i$).
- R-timestamp of $Q_k$ is updated whenever a transaction $T_j$ reads $Q_k$, and TS($T_j$) > R-timestamp($Q_k$).

# Multiversion Timestamp Ordering

- Suppose that transaction $T_i$ issues a **read**($Q$) or **write**($Q$) operation. Let $Q_k$ denote the version of $Q$ whose write timestamp is the largest write timestamp less than or equal to TS($T_i$).
  - If transaction $T_i$ issues a **read**($Q$), then the value returned is the content of version $Q_k$.
  - If transaction $T_i$ issues a **write**($Q$)
    - if TS($T_i$) < R-timestamp($Q_k$), then transaction $T_i$ is rolled back.
    - if TS($T_i$) = W-timestamp($Q_k$), the contents of $Q_k$ are overwritten
    - else a new version of $Q$ is created.

# Multiversion Timestamp Ordering

- Observe that
  - Reads always succeed
  - A write by $T_i$ is rejected if some other transaction $T_j$ that (in the serialization order defined by the timestamp values) should read $T_i$'s write, has already read a version created by a transaction older than $T_i$.
- Protocol guarantees serializability

# Multiversion Two-Phase Locking

- Differentiates between read-only transactions and update transactions
- *Update transactions* acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
  - Each successful **write** results in the creation of a new version of the data item written.
  - each version of a data item has a single timestamp whose value is obtained from a counter **ts-counter** that is incremented during commit processing.

# Multiversion Two-Phase Locking

- *Read-only transactions* are assigned a timestamp by reading the current value of **ts-counter** before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads.

# Multiversion Two-Phase Locking

- When an update transaction wants to read a data item:
  - it obtains a shared lock on it, and reads the latest version.
- When it wants to write an item
  - it obtains X lock on; it then creates a new version of the item and sets this version's timestamp to ∞.
- When update transaction $T_i$ completes, commit processing occurs:
  - $T_i$ sets timestamp on the versions it has created to **ts-counter** + 1
  - $T_i$ increments **ts-counter** by 1

# Multiversion Two-Phase Locking

- Read-only transactions that start after $T_i$ increments **ts-counter** will see the values updated by $T_i$.
- Read-only transactions that start before $T_i$ increments the
  **ts-counter** will see the value before the updates by $T_i$.
- Only serializable schedules are produced.

# MVCC: Implementation Issues

- Creation of multiple versions increases storage overhead
  - Extra tuples
  - Extra space in each tuple for storing version information
- Versions can, however, be garbage collected
  - E.g. if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp > 9, than Q5 will never be required again

# Research - Comparing Concurrency Schemes

| | Number of runs for Transactions | Transaction in each run | Committed Transaction | Rollback Transaction | Wait Transaction |
|---|---|---|---|---|---|
| 2PL | 100 | 10 | 180 | 370 | 550 |
| Timestamp | 100 | 10 | 288 | 712 | - |
| Optimistic | 100 | 10 | 333 | 677 | - |
| Multiversion | 100 | 10 | 666 | 334 | - |

Table 1
Average number of transaction for different methods of concurrency control

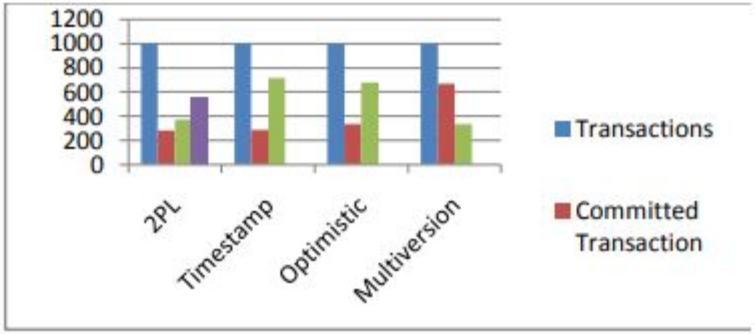



Figure 1 Comparison of all Techniques

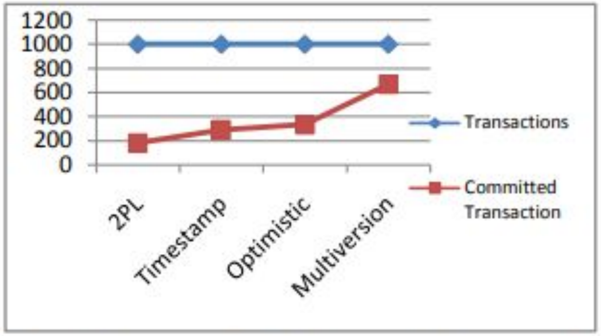



Figure 2 Average number of Commit transactions for different concurrency control methods

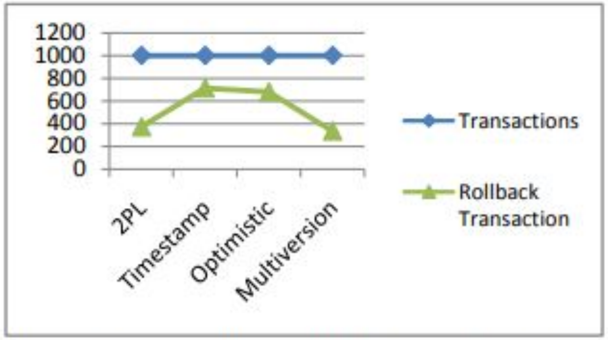



Figure.3.Average number of Rollback transactions for different concurrency control methods

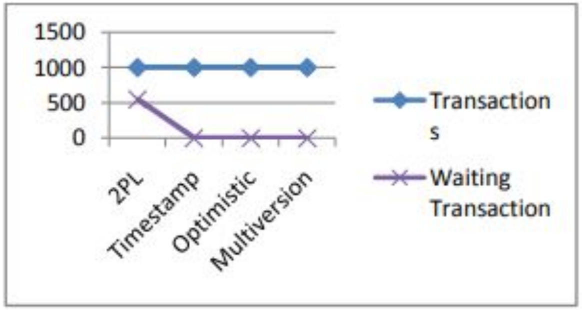



Figure 4.Average number of Wait transactions for different concurrency control methods

Source: https://www.ijarcce.com/upload/2015/march-15/IJARCCE%2060.pdf