# COMP 3311
# DATABASE MANAGEMENT SYSTEMS

## LECTURE 20
## CONCURRENCY CONTROL: LOCK-BASED PROTOCOLS

# CONCURRENCY CONTROL: OUTLINE

Lock-based Protocols

- Two-phase Locking Protocols
- Deadlock Handling
- Graph-based Protocols

Timestamp-based Protocols

- Timestamp-ordering Protocols
- Validation-based Protocols

Multiversion Schemes

Snapshot Isolation

# LOCKING

**Lock**: a mechanism to control concurrent access to a data item.

- A data item *Q* can be locked in one of two modes:
  1. shared-mode (shared lock)
     - Can *only* read Q. A shared lock is requested using a lock-s(*Q*) instruction.
  2. exclusive-mode (exclusive lock)
     - Can *both* read *and* write Q. An exclusive lock is requested using a lock-x(*Q*) instruction.
  - A data item *Q* is unlocked using an unlock(*Q*) instruction.

- A transaction must make a lock request to the concurrency-control manager before accessing a data item.
  - A transaction can proceed only after a request is granted.
  - A transaction should not request a lock until it is needed.

☞ **The concurrency control manager should allow *only* conflict-serializable schedules.**

# LOCK-COMPATIBILITY MATRIX

| Lock Mode | Shared (S) | Exclusive (X) |
|---|---|---|
| Shared (S) | true (grant) | false (deny) |
| Exclusive (X) | false (deny) | false (deny) |

**Shared** - can *only* read
**Exclusive** - can read and write

- A transaction may be granted a lock on a data item *Q* if the requested lock is *compatible* with locks already held on *Q* by other transactions.
- Any number of transactions can hold shared locks on a data item *Q*.
- If any transaction holds an exclusive lock on the data item, then no other transaction may hold <u>any</u> lock on the data item.
- If a lock cannot be granted, the requesting transaction is made to wait until all incompatible locks held by other transactions have been released. The lock is then granted.

# LOCKING EXAMPLE

Start with A=100, B=200

| | $T_1$ | $T_2$ | |
|---|---|---|---|
| | read(B) | | |
| | B := B - 50 | | |
| B=150 | write(B) | | |
| | read(A) | | |
| | A := A - 50 | | |
| A=50 | write(A) | | |
| | | read(A) | |
| | | read(B) | |
| | | display(A+B) | 200 |

Locking requires a set of rules, called a locking protocol, that all transactions follow when requesting and releasing locks. The rules restrict the set of possible schedules.

| | $T_1$ | | $T_2$ | CC Manager |
|---|---|---|---|---|
| | lock-x(B) | | | grant-X(B) |
| | read(B) | | | |
| | B := B - 50 | | | |
| B=150 | write(B) | | | |
| | unlock(B) | | | |
| | | | lock-s(A) | grant-S(A) |
| | | A=100 | read(A) | |
| | | | unlock(A) | |
| | | | lock-s(B) | grant-S(B) |
| | | B=150 | read(B) | |
| | | | unlock(B) | |
| | | | display(A+B) | 250 **INCORRECT!** |
| | lock-x(A) | | | grant-X(A) |
| | read(A) | | | |
| | A := A - 50 | | | |
| | write(A) | | | |
| | unlock(A) | | | |

Serializability may not be ensured if a transaction unlocks a data item immediately after its final access of that data item but before it commits.

# DEADLOCKS

- A system is deadlocked if there is a set of transactions such that every transaction is waiting for another transaction in the set.

- Neither $T_1$ nor $T_2$ can make progress — executing lock-s(B) causes $T_2$ to wait for $T_1$ to release its lock on B, while executing lock-x(A) causes $T_1$ to wait for $T_2$ to release its lock on A.

| $T_1$ | $T_2$ |
|---|---|
| lock-x(B) | |
| read(B) | |
| write(B) | |
| | lock-s(A) |
| | read(A) |
| | lock-s(B) |
| lock-x(A) | |

☞ **Such a situation is called a deadlock.**

☞ **To handle a deadlock, either $T_1$ or $T_2$ must be rolled back and its locks released.**

☞ **A rollback requires all transaction operations to be undone.**

☞ **A locking protocol needs to be able to handle deadlocks.**

# STARVATION

Starvation (waiting forever) is possible if the concurrency control manager is badly designed.

– A transaction may be waiting for a lock-x on an item, while a sequence of other transactions request, and are granted, a lock-s on the same item.

| $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ |
|---|---|---|---|---|
| lock-s(A) | | | | |
| | lock-x(A) | | | |
| | | lock-s(A) | | |
| | | | lock-s(A) | |
| | *wait* | | | lock-s(A) |
| unlock(A) | | | | |

☞ **A locking protocol needs to be able to handle starvation.**

# TWO-PHASE LOCKING (2PL) PROTOCOL

**Phase 1: Growing Phase**

A transaction may obtain locks but may not release any locks.

**Phase 2: Shrinking Phase**

A transaction may release locks but may not obtain any new locks.

Using 2PL, transactions can be serialized in the order of their lock points (i.e., the point where a transaction acquired its final lock).

☞ **If a schedule is executed by 2PL, then it must be conflict serializable.**

☞ **If a schedule is conflict serializable, then it may or may not be executed by 2PL.**

☞ **Not all conflict serializable schedules are allowed by 2PL.**

# LOCK CONVERSIONS

- Allow shared locks to be upgraded to exclusive locks and exclusive locks to be downgraded to shared locks.

**Phase 1: Growing Phase** (request or upgrade locks)

– A transaction can acquire a lock-s on a data item.

– A transaction can acquire a lock-x on a data item.

– A transaction can convert a lock-s to a lock-x (upgrade) on a data item.

**Phase 2: Shrinking Phase** (release or downgrade locks)

– A transaction can release a lock-s on a data item.

– A transaction can release a lock-x on a data item.

– A transaction can convert a lock-x to a lock-s (downgrade) on a data item.

☞ **Schedules are cascadeless if exclusive–locks are held until the end of the transaction.**

# STRICT AND RIGOROUS TWO-PHASE LOCKING

**Under 2PL, cascading roll-back is possible.**

## Strict Two-phase Locking

- Requires that all exclusive-mode locks be held until a transaction commits (shared-mode locks can be released anytime).
- Ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits.

## Rigorous Two-phase Locking

- Requires that all locks (both shared-mode and exclusive-mode) be held until a transaction commits.
- The transactions can be serialized in the order in which they commit.

☞ **Strict and rigorous 2PL schedules are cascadeless.**

# IMPLEMENTATION OF LOCKING

- A lock manager can be implemented as a separate process to which transactions send lock and unlock requests.

  ☞ **Transactions should not request locks until they are needed.**

- 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).

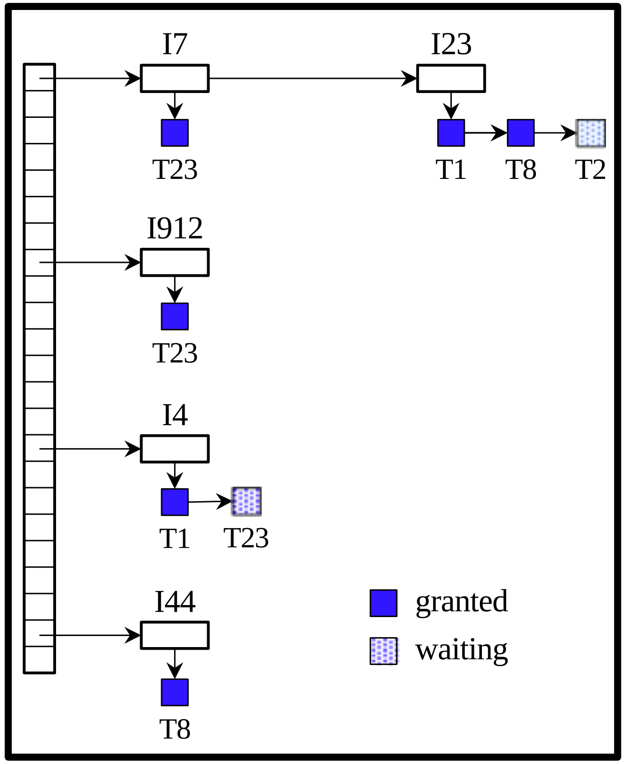
- The requesting transaction waits until its request is answered.

- 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 EXAMPLE

- Dark blue rectangles indicate granted locks, light blue ones indicate waiting requests.
- The lock table also records the type of lock granted or requested.
- A 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 waiting requests being checked to see if they can now be granted.




- If a transaction aborts, all waiting or granted requests of the transaction are deleted.
  - ➢ The lock manager also may keep a list of locks held by each transaction, to implement this efficiently.

☞ **This algorithm guarantees freedom from starvation.**

# DEADLOCK HANDLING

Start with A=100, B=200

| | $T_1$ | $T_2$ | |
|---|---|---|---|
| | read(B) | | |
| | B := B - 50 | | |
| B=150 | write(B) | | |
| | read(A) | | |
| | A := A - 50 | | |
| A=50 | write(A) | | |
| | | read(A) | |
| | | read(B) | |
| | | display(A+B) | 200 |

| | $T_1$ | $T_2$ | CC Manager |
|---|---|---|---|
| | lock-x(B) | | grant-X(B) |
| | read(B) | | |
| | B := B - 50 | | |
| B=150 | write(B) | | |
| | | lock-s(A) | grant-S(A) |
| | A=100 | read(A) | |
| | | lock-s(B) | wait-S(B) (for $T_1$) |
| | lock-x(A) | | wait-X(A) (for $T_2$) |
| | | **deadlock!** | |

☞ **2PL permits deadlocks.**

**The *potential* for deadlock exists in most locking protocols.**
**Deadlocks are a necessary evil of locking protocols.**

# DEADLOCK PREVENTION

- A deadlock prevention protocol ensures that the system will *never* enter a deadlock state.

## Strategy 1: Order lock requests

– Require each transaction to lock all its data items *before* it begins execution (pre-declaration).

- Often hard to predict, beforehand, what data items need to be locked.
- Locked data items may be unused for a long time.

– Impose a partial/total ordering of all data items and require that a transaction can *lock data items only in the order specified by the partial/total order* (e.g., tree protocol).

## Strategy 2: Preemption and/or rollback

– Preempt and/or rollback a transaction when needed.

– Use transaction timestamps to control preemption and rollback.

# DEADLOCK PREVENTION (CONT'D)

## Wait-die Scheme

- An older transaction may wait for a younger one to release a data item.
- A younger transaction never waits for an older one; it is rolled back instead.
- A transaction may die several times before acquiring the needed data item.

## Wound-wait Scheme

- An older transaction ***wounds*** (forces the rollback of) a younger transaction instead of waiting for it.
- A younger transaction may wait for an older one to release a data item.
- There may be fewer rollbacks than in the wait-die scheme.

- In both schemes, a rolled back transaction is restarted with its <u>original timestamp</u> so that older transactions have precedence over younger ones, thus, avoiding starvation.

# DEADLOCK PREVENTION (cont'd)

## Timeout-Based Schemes

- A transaction waits for a lock only for a specified amount of time.
- After a pre-defined waiting period, the transaction is rolled back.
- Simple to implement; but starvation is possible.
- Often difficult to determine a good value of the timeout interval.

# DEADLOCK DETECTION

- Deadlocks can be detected using a wait-for graph $G = (V, E)$, where

  $V$ 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 into the wait-for graph.

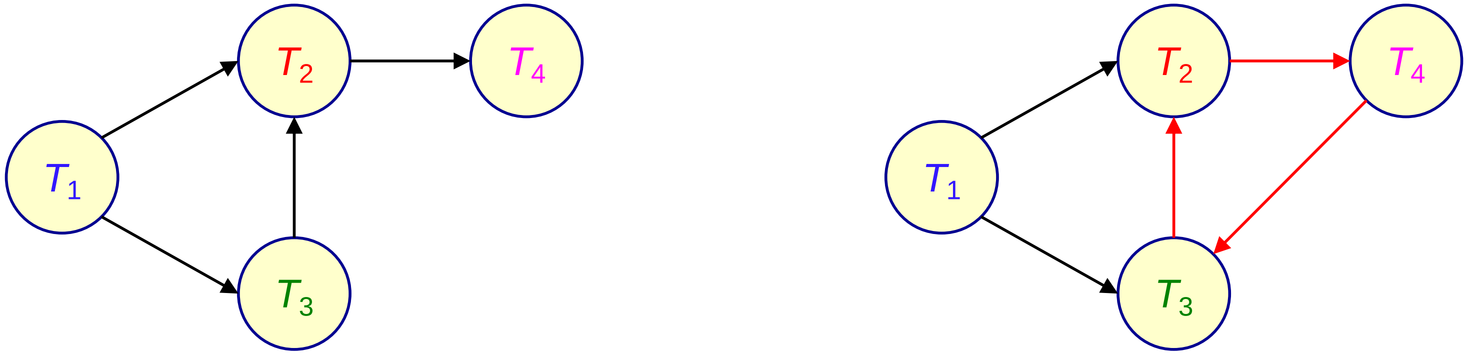
  – This edge is removed only when $T_j$ is no longer holding a data item needed by $T_i$.

☞ **The system is in a deadlock state *if and only if* the wait-for graph has a cycle.**

☞ **The system must invoke a deadlock-detection algorithm periodically to look for cycles in the wait-for graph.**

# DEADLOCK DETECTION (CONT'D)

## Example




Wait-for graph without a cycle.

Wait-for graph with a cycle.

☞ **This is not the same as a precedence graph!**

# DEADLOCK RECOVERY

## Victim Selection

– Select as a victim the transaction that will incur minimum cost.

## Rollback

– Need to determine how far to roll back the transaction.

- Total rollback: abort the transaction and then restart it.
- Partial rollback: roll back the transaction only as far as necessary to break the deadlock. (Requires the system to maintain additional information.)

## Starvation

– Can happen if the same transaction is always chosen as the victim.

– Can include the number of rollbacks in the cost factor to avoid starvation.

# CONCURRENCY CONTROL: OUTLINE

✓ Lock-based Protocols

- ✓ Two-phase Locking Protocols
- ✓ Deadlock Handling
- ➡ **Graph-based Protocols**

Timestamp-based Protocols

- Timestamp-ordering Protocols
- Validation-based Protocols

Multiversion Schemes

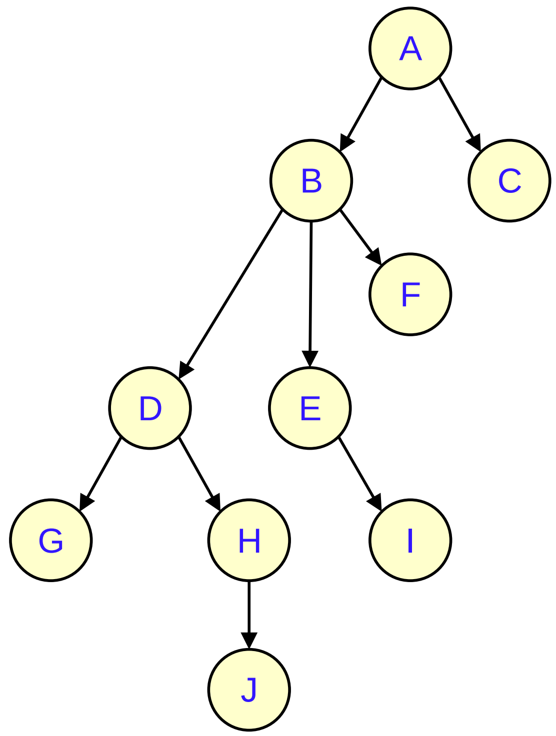
Snapshot Isolation

# GRAPH-BASED PROTOCOLS

- For protocols that are not two phase, additional information is needed about how each transaction accesses the database.
- The simplest such protocols require knowledge about the order in which the data items will be accessed.
- This knowledge can be acquired by imposing a partial ordering on the set **D** = $\{d_1, d_2, \ldots, d_h\}$ of all data items.
  - If $d_i$ precedes $d_j$ in the ordering, then any transaction accessing both $d_i$ and $d_j$ *must* access $d_i$ *before* accessing $d_j$.
  - The set **D** may be viewed as a directed acyclic graph, called a *database graph*.
- For simplicity, we consider only those graphs that are rooted trees.

# TREE PROTOCOL

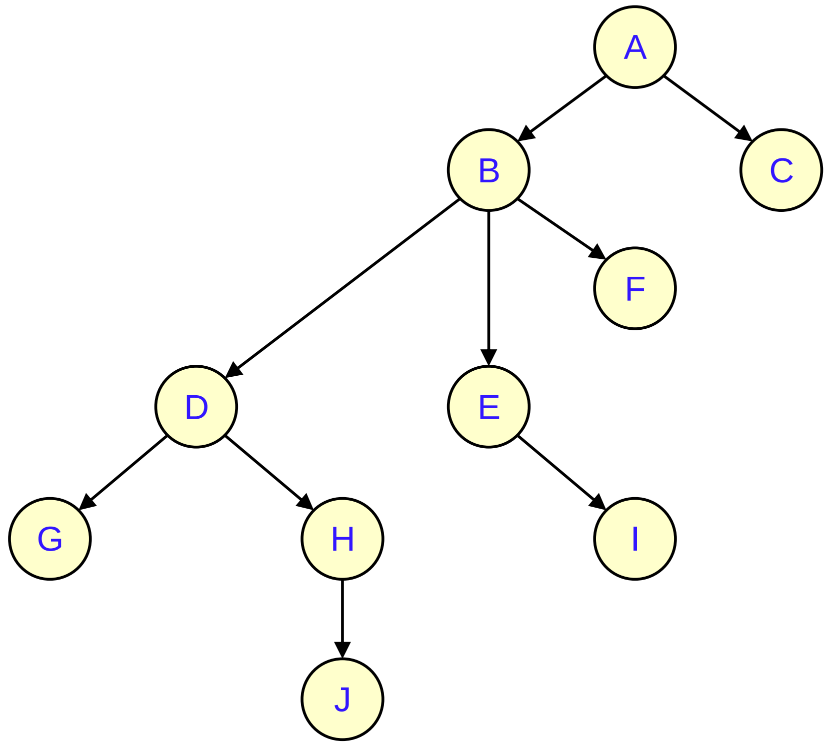
1. Only lock-x instructions are allowed.
2. The first lock by $T_i$ may be on any data item.
3. Subsequently, a data item $Q$ can be locked by $T_i$ *only if* the parent of $Q$ is currently locked by $T_i$.
4. Data items may be unlocked at any time.
5. A data item that has been unlocked by $T_i$ cannot be locked again by $T_i$.

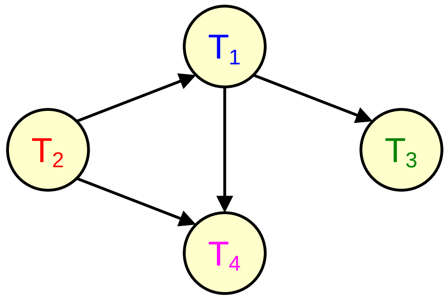



☞ **All legal schedules under the tree protocol are conflict serializable.**

# TREE PROTOCOL: SERIALIZABLE SCHEDULE

| $T_1$ | $T_2$ | $T_3$ | $T_4$ |
|---|---|---|---|
| lock-X(B) | | | |
| | lock-X(D) | | |
| | lock-X(H) | | |
| | unlock(D) | | |
| lock-X(E) | | | |
| lock-X(D) | | | |
| unlock(B) | | | |
| unlock(E) | | | |
| | | lock-X(B) | |
| | | lock-X(E) | |
| | unlock(H) | | |
| lock-X(G) | | | |
| unlock(D) | | | |
| | | | lock-X(D) |
| | | | lock-X(H) |
| | | | unlock(D) |
| | | | unlock(H) |
| | | unlock(E) | |
| | | unlock(B) | |
| unlock(G) | | | |




Precedence Graph

Equivalent serial schedule:

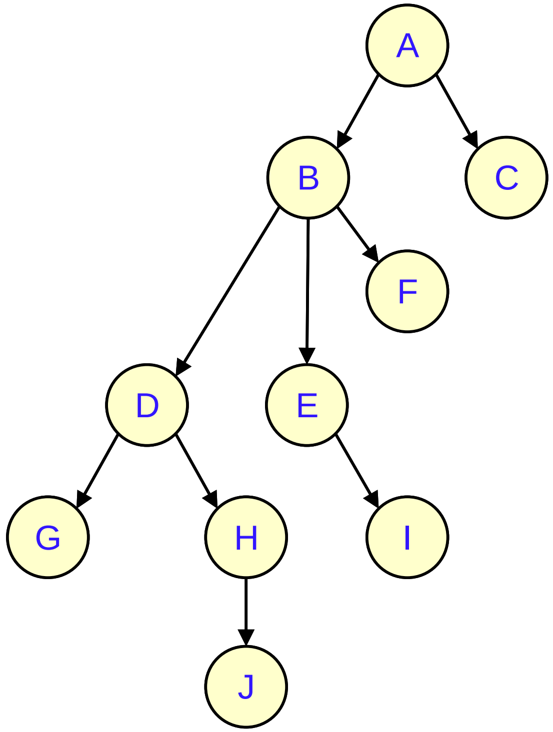
$T_2$, $T_1$, $T_4$, $T_3$

or

$T_2$, $T_1$, $T_3$, $T_4$

# TREE PROTOCOL: PROPERTIES

- The tree protocol
  - ensures conflict serializability.
  - is deadlock free.
- Unlocking may occur earlier than in the two-phase locking protocol.
  - Thus, there may be shorter waiting times and an increase in concurrency.
- However, a transaction may have to lock data items that it does not access.
  - Increased locking overhead and additional waiting time.
  - Potential decrease in concurrency.




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