# Chapter 18 : Concurrency Control

Database System Concepts, 7th Ed.

## Outline

- 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.

# Schedule With Lock Grants

- Grants omitted in rest of chapter
  - Assume grant happens just before the next instruction following lock request
- This schedule is not serializable (why?)
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks.
- Locking protocols enforce serializability by restricting the set of possible schedules.

| $T_1$ | $T_2$ | concurrency-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($A$) | | |

# Deadlock

- 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($A$) | |

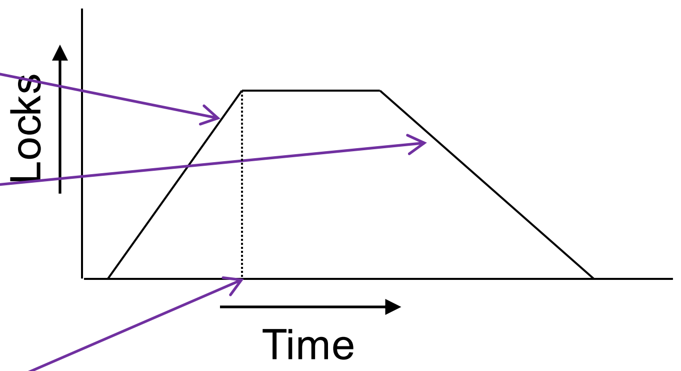
- Neither $T_3$ nor $T_4$ can make progress — executing **lock-S**(*B*) causes $T_4$ to wait for $T_3$ to release its lock on *B*, while executing **lock-X**(*A*) causes $T_3$ to wait for $T_4$ to release its lock on *A*.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of $T_3$ or $T_4$ must be rolled back and its locks released.

## Deadlock (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

- 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).



# The Two-Phase Locking Protocol (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks
- Extensions to basic two-phase locking needed to ensure recoverability of freedom from cascading roll-back
  - **Strict two-phase locking:** a transaction must hold all its exclusive locks till it commits/aborts.
    - Ensures recoverability and avoids cascading roll-backs
  - **Rigorous two-phase locking**: a transaction must hold *all* locks till commit/abort.
    - Transactions can be serialized in the order in which they commit.
- Most databases implement rigorous two-phase locking, *but refer to it as simply two-phase locking*

## The Two-Phase Locking Protocol (Cont.)

- Two-phase locking is not a necessary condition for serializability
  - There are conflict serializable schedules that cannot be obtained if the two-phase locking protocol is used.
- In the absence of extra information (e.g., ordering of access to data), two-phase locking is necessary 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.*

| $T_1$ | $T_2$ |
|---|---|
| lock-X($B$) | |
| read($B$) | |
| $B := B - 50$ | |
| write($B$) | |
| unlock($B$) | |
| | lock-S($A$) |
| | read($A$) |
| | unlock($A$) |
| | lock-S($B$) |
| | read($B$) |
| | unlock($B$) |
| | display($A + B$) |
| lock-X($A$) | |
| read($A$) | |
| $A := A + 50$ | |
| write($A$) | |
| unlock($A$) | |

## Locking Protocols

- Given a locking protocol (such as 2PL)
  - A schedule S is **legal** under a locking protocol if it can be generated by a set of transactions that follow the protocol
  - A protocol **ensures** serializability if all legal schedules under that protocol are serializable

# Lock Conversions

- Two-phase locking protocol with lock conversions:
  - Growing 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**)
  - Shrinking Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (**downgrade**)
- This protocol ensures serializability

## 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.)

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

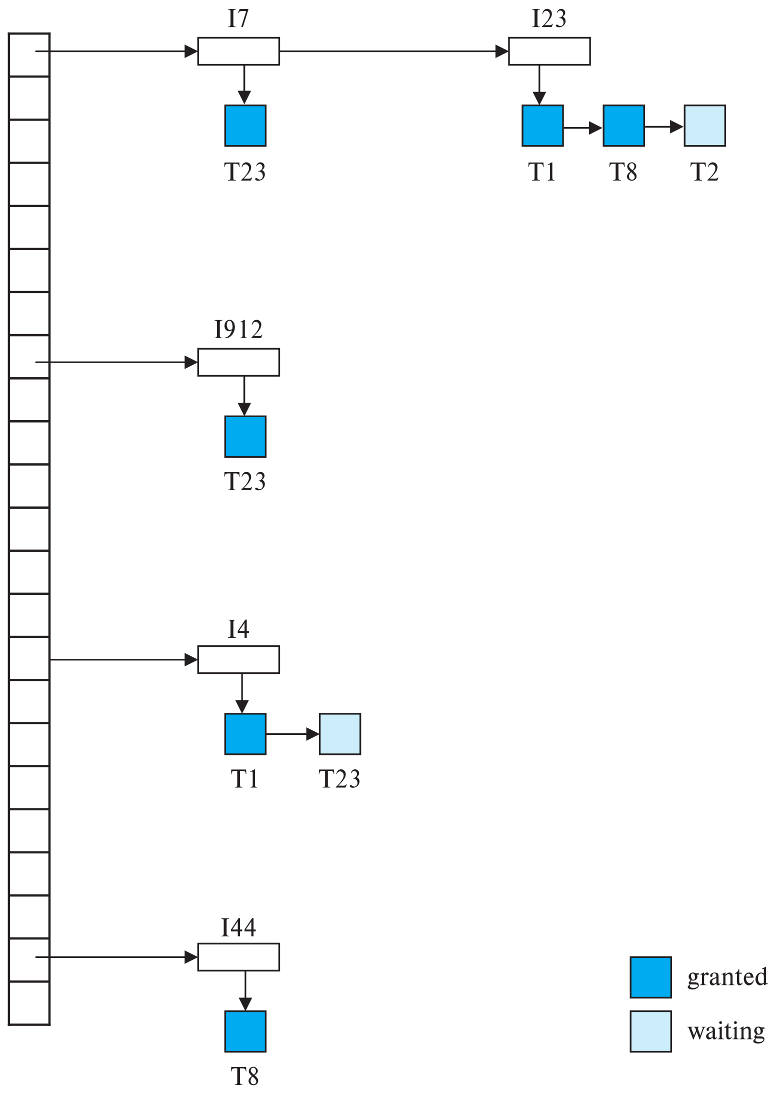
```
if $T_i$ has a lock-X on $D$
  then
    write($D$)
  else begin
      if necessary wait until no other trans. has any lock on $D$,
      if $T_i$ has a lock-S on $D$
          then
            upgrade lock on $D$ to lock-X
          else
            grant $T_i$ a lock-X on $D$
          write($D$)
    end;
```

- **All locks are released after commit or abort**

## Implementation of Locking

- A **lock manager** can be implemented as a separate process
- Transactions can send lock and unlock requests as messages
- 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 an in-memory data-structure called a **lock table** to record granted locks and pending requests

# Lock Table



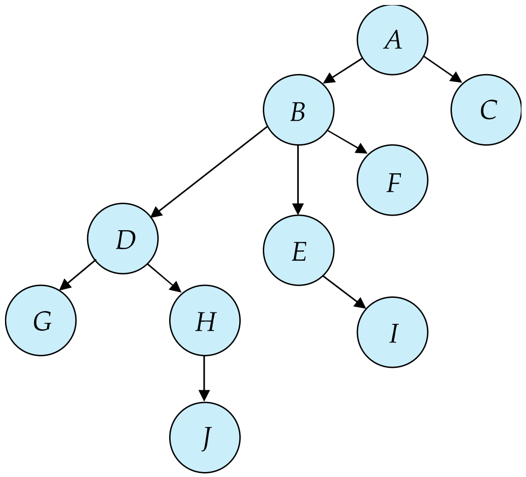
- Dark rectangles indicate granted locks, light colored ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests 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** = $\{d_1, d_2, \ldots, d_h\}$ 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
    - 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 the tree protocol, and vice versa.

## 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.

| $T_3$ | $T_4$ |
|---|---|
| lock-X($B$) | |
| read($B$) | |
| $B := B - 50$ | |
| write($B$) | |
| | lock-S($A$) |
| | read($A$) |
| | lock-S($B$) |
| lock-X($A$) | |

## Deadlock Handling

- ***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 (pre-declaration).
  - 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

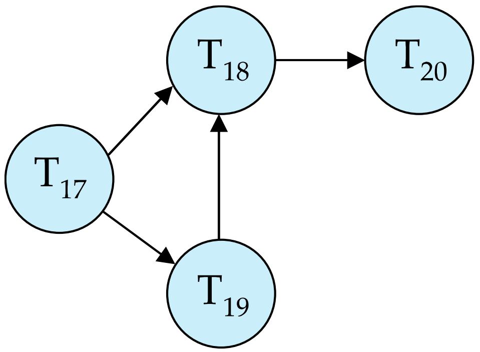
- **wait-die** scheme — non-preemptive
  - Older transaction may wait for younger one to release data item.
  - Younger transactions never wait for older ones; they are rolled back instead.
  - A transaction may die several times before acquiring a lock
- **wound-wait** scheme — preemptive
  - Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it.
  - Younger transactions may wait for older ones.
  - Fewer rollbacks than *wait-die* scheme.
- In both schemes, a rolled back transactions is restarted with its original timestamp.
  - Ensures that older transactions have precedence over newer ones, and starvation is thus avoided.

## Deadlock prevention (Cont.)

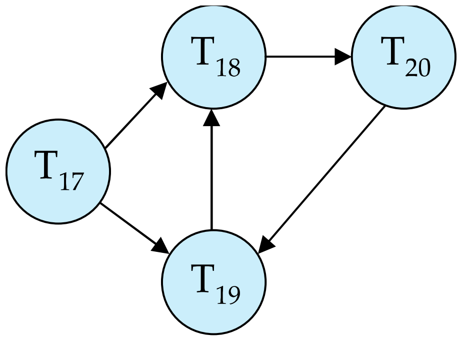
- **Timeout-Based Schemes**:
  - A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - Ensures that deadlocks get resolved by timeout if they occur
  - Simple to implement
  - But may roll back transaction unnecessarily in absence of deadlock
    - Difficult to determine good value of the timeout interval.
  - Starvation is also possible

# Deadlock Detection

- **Wait-for graph**
  - *Vertices:* transactions
  - *Edge from $T_i \rightarrow T_j$*. : if $T_i$ is waiting for a lock held in conflicting mode by $T_j$
- The system is in a deadlock state if and only if the wait-for graph has a cycle.
- Invoke a deadlock-detection algorithm periodically to look for cycles.



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 cycle.
    - 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.
    - **Partial rollback**: Roll back victim transaction only as far as necessary to release locks that another transaction in cycle is waiting for
- Starvation can happen (why?)
  - One solution: oldest transaction in the deadlock set is never chosen as victim

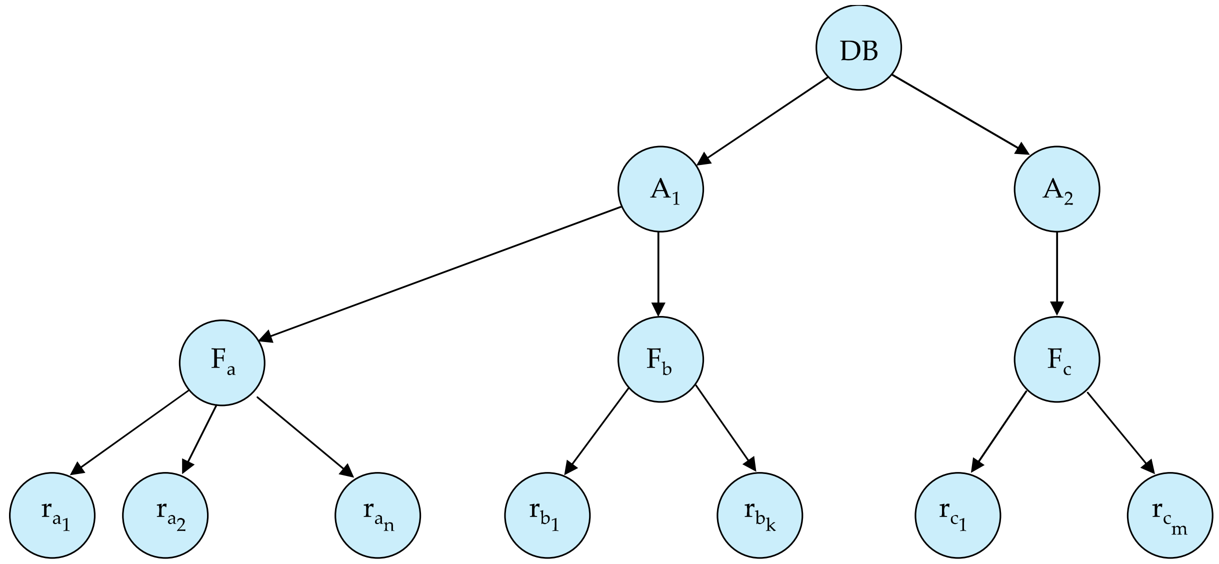
## Multiple Granularity

- 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 descendants 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

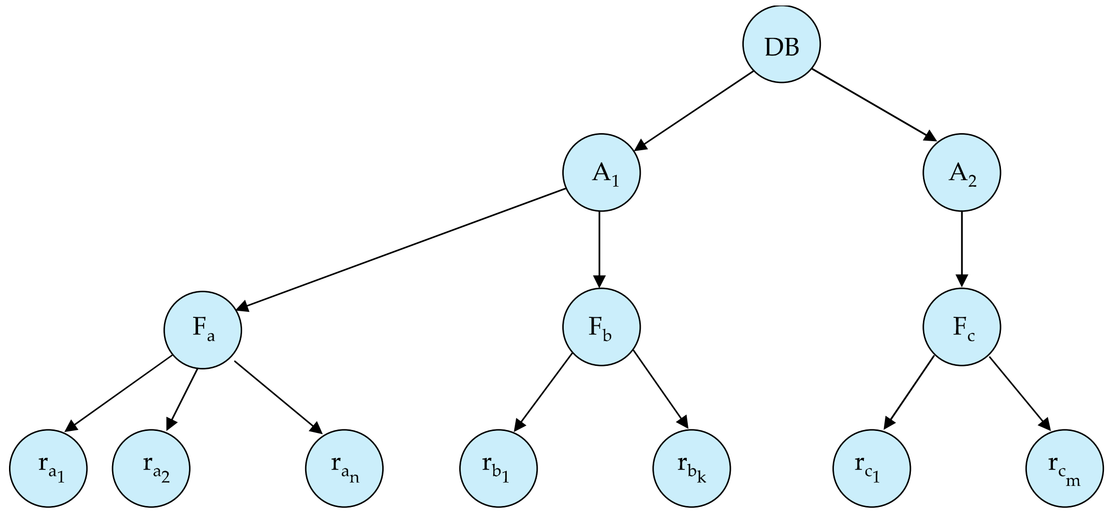
The levels, starting from the coarsest (top) level are

- *database*
- *area*
- *file*
- *record*



# Example of Granularity Hierarchy

- The levels, starting from the coarsest (top) level are
  - *database*
  - *area*
  - *file*
  - *record*
- The corresponding tree



## 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.
- **Lock granularity escalation**: in case there are too many locks at a particular level, switch to higher granularity S or X lock

## Insert/Delete Operations and Predicate Reads

- Locking rules for insert/delete operations
  - An exclusive lock must be obtained on an item before it is deleted
  - A transaction that inserts a new tuple into the database I automatically given an X-mode lock on the tuple
- Ensures that
  - reads/writes conflict with deletes
  - Inserted tuple is not accessible by other transactions until the transaction that inserts the tuple commits

# Phantom Phenomenon

- Example of **phantom phenomenon**.
  - A transaction T1 that performs **predicate read** (or scan) of a relation
    - **select count**(*)
      **from** *instructor*
      **where** *dept_name* = 'Physics'
  - and a transaction T2 that inserts a tuple while T1 is active but after predicate read
    - **insert into** *instructor* **values** ('11111', 'Feynman', 'Physics', 94000)

    (conceptually) conflict in spite of not accessing any tuple in common.
- If only tuple locks are used, non-serializable schedules can result
  - E.g. the scan transaction does not see the new instructor, but may read some other tuple written by the update transaction
- Can also occur with updates
  - E.g. update Wu's department from Finance to Physics

## Insert/Delete Operations and Predicate Reads

- **Another Example**: T1 and T2 both find maximum instructor ID in parallel, and create new instructors with ID = maximum ID + 1
  - Both instructors get same ID, not possible in serializable schedule
- Schedule

| T1 | T2 |
|---|---|
| Read(instructor where dept_name='Physics') | |
| | Insert Instructor in Physics |
| | Insert Instructor in Comp. Sci. |
| | Commit |
| Read(instructor where dept_name='Comp. Sci.') | |

## Handling Phantoms

- There is a conflict at the data level
  - The transaction performing predicate read or scanning the relation is reading information that indicates what tuples the relation contains
  - The transaction inserting/deleting/updating a tuple updates the same information.
  - The conflict should be detected, e.g. by locking the information.
- One solution:
  - Associate a data item with the relation, to represent the information about what tuples the relation contains.
  - Transactions scanning the relation acquire a shared lock in the data item,
  - Transactions inserting or deleting a tuple acquire an exclusive lock on the data item. (Note: locks on the data item do not conflict with locks on individual tuples.)
- Above protocol provides very low concurrency for insertions/deletions.

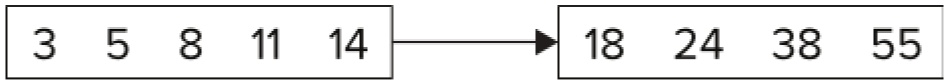
## Index Locking To Prevent Phantoms

- **Index locking protocol** to prevent phantoms
  - Every relation must have at least one index.
  - A transaction can access tuples only after finding them through one or more indices on the relation
  - A transaction $T_i$ that performs a lookup must lock all the index leaf nodes that it accesses, in S-mode
    - Even if the leaf node does not contain any tuple satisfying the index lookup (e.g. for a range query, no tuple in a leaf is in the range)
  - A transaction $T_i$ that inserts, updates or deletes a tuple $t_i$ in a relation $r$
    - Must update all indices to $r$
    - Must obtain exclusive locks on all index leaf nodes affected by the insert/update/delete
  - The rules of the two-phase locking protocol must be observed
- Guarantees that phantom phenomenon won't occur

## Next-Key Locking to Prevent Phantoms

- Index-locking protocol to prevent phantoms locks entire leaf node
  - Can result in poor concurrency if there are many inserts
- **Next-key locking protocol**: provides higher concurrency
  - Lock all values that satisfy index lookup (match lookup value, or fall in lookup range)
  - Also lock next key value in index
    - even for inserts/deletes
  - Lock mode: S for lookups, X for insert/delete/update
- Ensures detection of query conflicts with inserts, deletes and updates

Consider B+-tree leaf nodes as below, with query predicate $7 \leq X \leq 16$.
Check what happens with next-key locking when inserting: (i) 15 and (ii) 7



## Timestamp Based Concurrency Control

## Timestamp-Based Protocols

- Each transaction $T_i$ is issued a timestamp TS($T_i$) when it enters the system.
  - Each transaction has a *unique* timestamp
  - Newer transactions have timestamps strictly greater than earlier ones
  - Timestamp could be based on a logical counter
    - Real time may not be unique
    - Can use (wall-clock time, logical counter) to ensure
- Timestamp-based protocols manage concurrent execution such that
  **time-stamp order = serializability order**
- Several alternative protocols based on timestamps

## Timestamp-Ordering Protocol

The **timestamp ordering (TSO) 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.
- Imposes rules on read and write operations to ensure that
  - Any conflicting operations are executed in timestamp order
  - Out of order operations cause transaction rollback

## Timestamp-Based Protocols (Cont.)

- 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($T_i$).

# Example of Schedule Under TSO

- Is this schedule valid under TSO?

  Assume that initially:
  R-TS(A) = W-TS(A) = 0
  R-TS(B) = W-TS(B) = 0
  Assume TS($T_{25}$) = 25 and
  TS($T_{26}$) = 26

  We represent an item by a triple
  <R-TS,W-TS,Value>

A=<0,0,100> B=<0,0,100>

| $T_{25}$ | $T_{26}$ | |
|---|---|---|
| read($B$) | | B=<25,0,100> |
| | read($B$) | B=<26,0,100> |
| | $B := B - 50$ | |
| | write($B$) | B=<26,26,50> |
| read($A$) | | A=<25,0,100> |
| | read($A$) | A=<26,0,100> |
| display($A + B$) | | |
| 200 | $A := A + 50$ | |
| | write($A$) | A=<26,26,150> |
| | display($A + B$) | |
| | 200 | |

- How about this one, where initially
  R-TS(Q)=W-TS(Q)=0

| $T_{27}$ | $T_{28}$ | Q=<0,0,Q0> |
|---|---|---|
| read($Q$) | | Q=<27,0,Q0> |
| | write($Q$) | Q=<27,28,Q1> |
| write($Q$) | | abort(T27) |

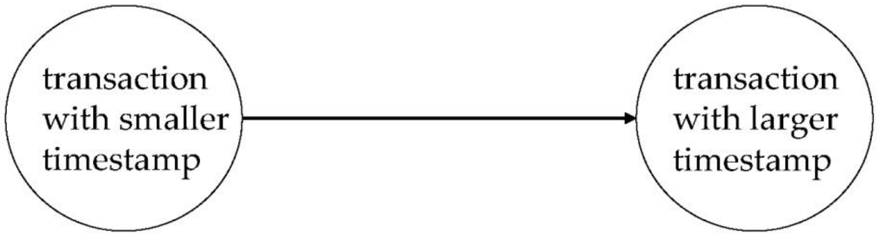
# Another Example Under TSO

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5, with all R-TS and W-TS = 0 initially

| $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | |
|---|---|---|---|---|---|
| | | | | read (*X*) | X=<5,0,X0> |
| | read (*Y*) | | | | Y=<2,0,Y0> |
| read (*Y*) | | | | | Y=<2,0,Y0> |
| | | write (*Y*) | | | Y=<2,3,Y3> |
| | | write (*Z*) | | | Z=<0,3,Z3> |
| | | | | read (*Z*) | Z=<5,3,Z3> |
| | read (*Z*) | | | | abort(Z2) |
| | abort | | | | |
| read (*X*) | | | | | X=<5,0,X0> |
| | | | read (*W*) | | W=<4,0,W0> |
| | | write (*W*) | | | abort(T3) |
| | | abort | | | |
| | | | | write (*Y*) | Y=<2,5,Y5> |
| | | | | write (*Z*) | Z=<5,5,Z5> |

## Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:


transaction with smaller timestamp
transaction with larger timestamp

  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

- 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-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.

| $T_{27}$ | $T_{28}$ |
|---|---|
| read($Q$) | |
| | write($Q$) |
| write($Q$) | |

Q=<0,0,Q0>

Q=<27,28,Q1>

IGNORED!

## Validation-Based Protocol

- Idea: can we use commit time as serialization order?
- To do so:
  - Postpone writes to end of transaction
  - Keep track of data items read/written by transaction
  - **Validation** performed at commit time, detect any out-of-serialization order reads/writes
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation

# 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.
  - We 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.

## Validation-Based Protocol (Cont.)

- Each transaction $T_i$ has 3 timestamps
  - **StartTS**($T_i$) : the time when $T_i$ started its execution
  - **ValidationTS**($T_i$): the time when $T_i$ entered its validation phase
  - **FinishTS**($T_i$) : the time when $T_i$ finished its write phase
- Validation tests use above timestamps and read/write sets to ensure that serializability order is determined by validation time
  - Thus, TS($T_i$) = ValidationTS($T_i$)
- Validation-based protocol has been found to give greater degree of concurrency than locking/TSO if probability of conflicts is low.

## 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:
  - **finishTS**($T_i$) < **startTS**($T_j$)
  - **startTS**($T_j$) < **finishTS**($T_i$) < **validationTS**($T_j$) **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:
  - First condition applies when execution is not concurrent
    - The writes of $T_j$ do not affect reads of $T_i$ since they occur after $T_i$ has finished its reads.
  - If the second condition holds, execution is concurrent, $T_j$ does not read any item written by $T_i$.

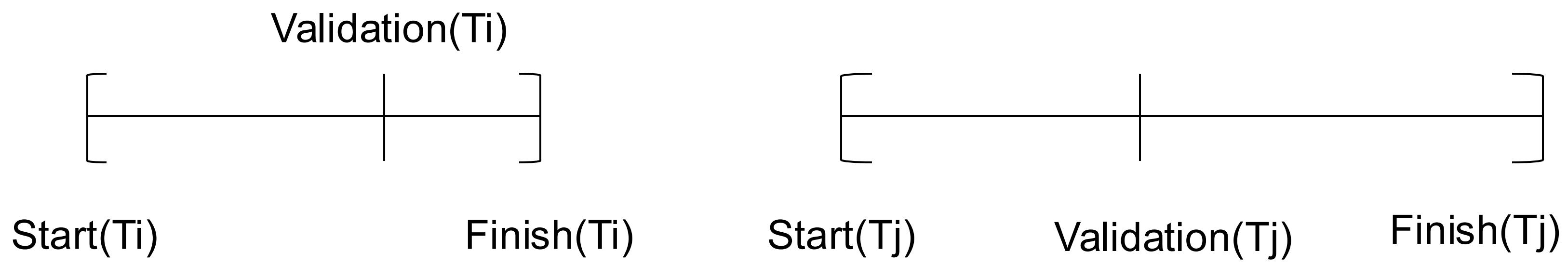
# Schedule Produced by Validation

- Example of schedule produced using validation

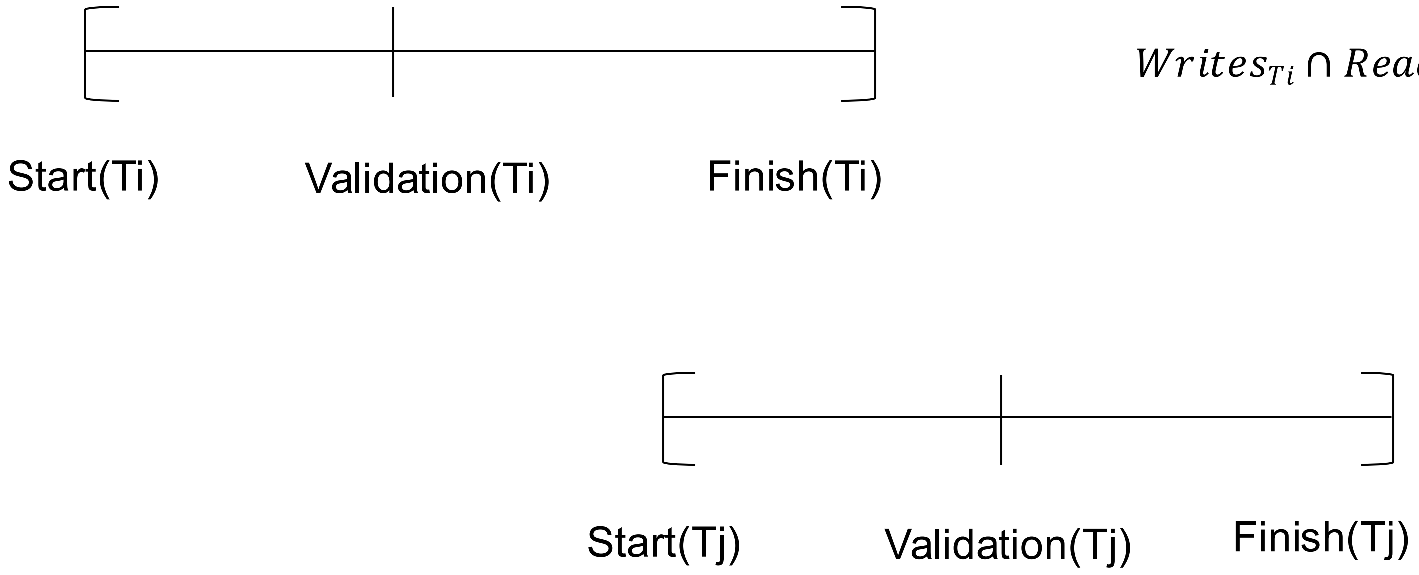
| $T_{25}$ | $T_{26}$ |
|---|---|
| read($B$) | |
| | read($B$)<br>$B := B - 50$<br>read($A$)<br>$A := A + 50$ |
| read($A$)<br><validate><br>display($A + B$) | |
| | <validate><br>write($B$)<br>write($A$) |

# Allowed Schedules

a)



b)



$Writes_{Ti} \cap Reads_{Tj} = \{\}$

## Multiversion Concurrency Control

## Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency. Several variants:
  - **Multiversion Timestamp Ordering**
  - **Multiversion Two-Phase Locking**
  - **Snapshot isolation**
- Key ideas:
  - Each successful **write** results in the creation of a new version of the data item written.
  - Use timestamps to label versions.
  - When a **read**(*Q*) operation is issued, select an appropriate version of *Q* based on the timestamp of the transaction issuing the read request, 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 $\mathrm{Q}_\mathrm{k}$
  - **R-timestamp**($Q_k$) -- largest timestamp of a transaction that successfully read version $\mathrm{Q}_\mathrm{k}$

## Multiversion Timestamp Ordering (Cont)

- 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($T_i$).
  1. If transaction $T_i$ issues a **read**($Q$), then
     - the value returned is the content of version $Q_k$
     - If R-timestamp($Q_k$) < TS($T_i$), set R-timestamp($Q_k$) = TS($T_i$),
  2. If transaction $T_i$ issues a **write**($Q$)
     1. if TS($T_i$) < R-timestamp($Q_k$), then transaction $T_i$ is rolled back.
     2. if TS($T_i$) = W-timestamp($Q_k$), the contents of $Q_k$ are overwritten
     3. Otherwise, a new version $Q_i$ of $Q$ is created
        - W-timestamp($Q_i$) and R-timestamp($Q_i$) are initialized to TS($T_i$).

## Multiversion Timestamp Ordering (Cont)

- Observations
  - 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.
  - Read of a data item returns the latest version of the item
  - The first **write** of Q by $T_i$ results in the creation of a new version $Q_i$ of the data item Q written
    - W-timestamp($Q_i$) set to ∞ initially
  - When update transaction $T_i$ completes, commit processing occurs:
    - Value **ts-counter** stored in the database is used to assign timestamps
      - **ts-counter** is locked in two-phase manner
    - Set TS($T_i$) = **ts-counter** + 1
    - Set W-timestamp($Q_i$) = TS($T_i$) for all versions $Q_i$ that it creates
    - **ts-counter** = **ts-counter +** 1

## Multiversion Two-Phase Locking (Cont.)

- **Read-only transactions**
  - are assigned a timestamp = **ts-counter** when they start execution
  - follow the multiversion timestamp-ordering protocol for performing reads
    - Do not obtain any locks
- 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
- Issues with
  - primary key and foreign key constraint checking
  - Indexing of records with multiple versions

  See textbook for details

## Snapshot Isolation

- Motivation: Decision support queries that read large amounts of data have concurrency conflicts with OLTP transactions that update a few rows
  - Poor performance results
- Solution 1:  Use multiversion 2-phase locking
  - Give logical "snapshot" of database state to read only transaction
    - Reads performed on snapshot
  - Update (read-write) transactions use normal locking
  - Works well, but how does system know a transaction is read only?
- Solution 2 (partial): Give snapshot of database state to every transaction
  - Reads performed on snapshot
  - Use 2-phase locking on updated data items
  - Problem: variety of anomalies such as lost update can result
  - Better solution: snapshot isolation level (next slide)

# Snapshot Isolation

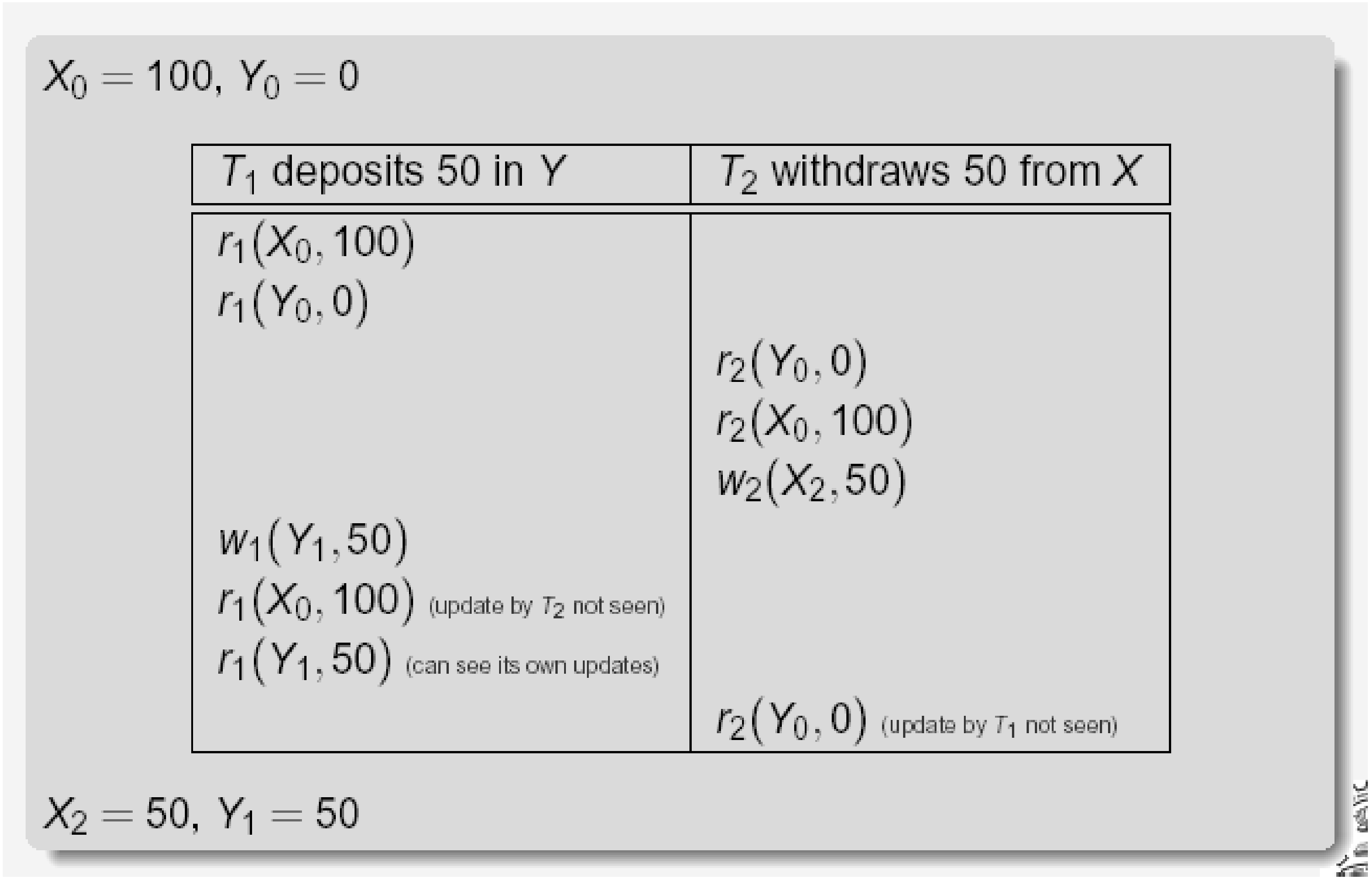
- A transaction T1 executing with Snapshot Isolation
  - Takes snapshot of committed data at start
  - Always reads/modifies data in its own snapshot
  - Updates of concurrent transactions are not visible to T1
  - Writes of T1 complete when it commits
  - **First-committer-wins rule**:
    - Commits only if no other concurrent transaction has already written data that T1 intends to write.

| T1 | T2 | T3 |
|---|---|---|
| W(Y := 1)<br>Commit | | |
| | Start<br>R(X) → 0<br>R(Y)→ 1 | |
| | | W(X:=2)<br>W(Z:=3)<br>Commit |
| | R(Z) → 0<br>R(Y) → 1<br>W(X:=3)<br>Commit-Req<br>Abort | |

Concurrent updates not visible
Own updates are visible
Not first-committer of X
Serialization error, T2 is rolled back

# Snapshot Read

- Concurrent updates invisible to snapshot read

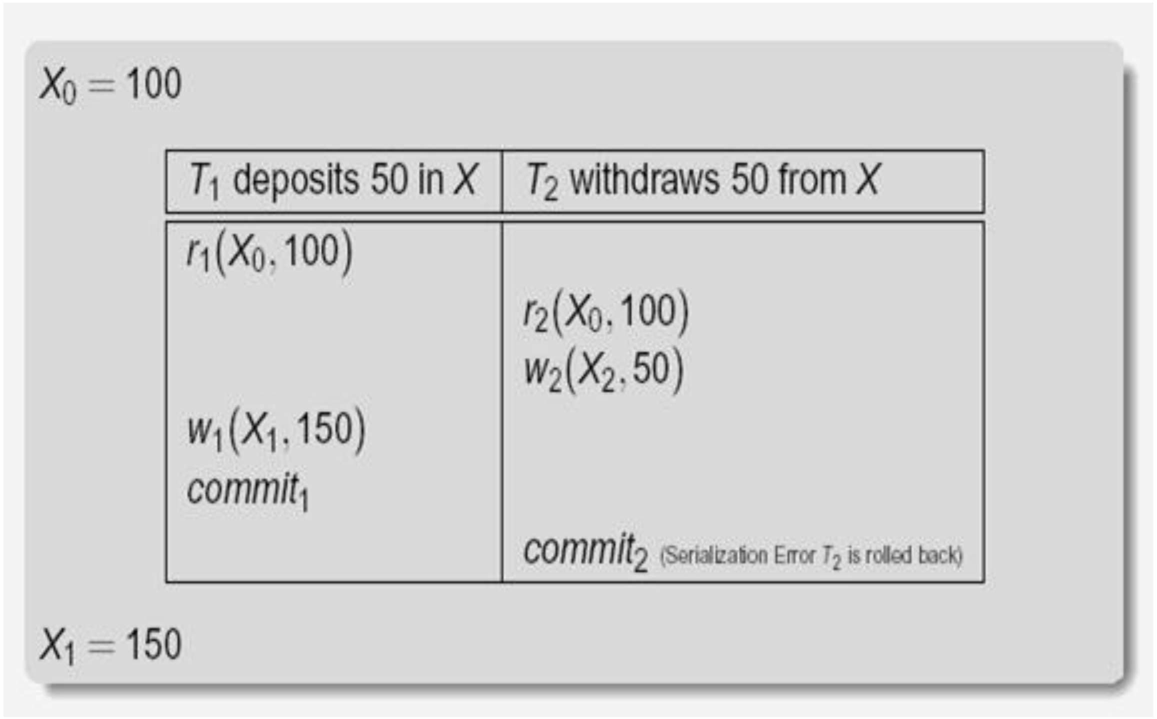


$X_0 = 100,\ Y_0 = 0$

| $T_1$ deposits 50 in $Y$ | $T_2$ withdraws 50 from $X$ |
|---|---|
| $r_1(X_0, 100)$ | |
| $r_1(Y_0, 0)$ | |
| | $r_2(Y_0, 0)$ |
| | $r_2(X_0, 100)$ |
| | $w_2(X_2, 50)$ |
| $w_1(Y_1, 50)$ | |
| $r_1(X_0, 100)$ (update by $T_2$ not seen) | |
| $r_1(Y_1, 50)$ (can see its own updates) | |
| | $r_2(Y_0, 0)$ (update by $T_1$ not seen) |

$X_2 = 50,\ Y_1 = 50$

# **Snapshot Write:** First Committer Wins



$X_0 = 100$

| $T_1$ deposits 50 in $X$ | $T_2$ withdraws 50 from $X$ |
|---|---|
| $r_1(X_0, 100)$ | |
| | $r_2(X_0, 100)$ |
| | $w_2(X_2, 50)$ |
| $w_1(X_1, 150)$ | |
| $commit_1$ | |
| | $commit_2$ (Serialization Error $T_2$ is rolled back) |

$X_1 = 150$

- Variant: "**First-updater-wins**"
  - Check for concurrent updates when write occurs by locking item
    - But lock should be held till all concurrent transactions have finished
  - (Oracle uses this plus some extra features)
  - Differs only in when abort occurs, otherwise equivalent

## Benefits of SI

- Reads are *never* blocked,
  - and also don't block other txns activities
- Performance similar to Read Committed
- Avoids several anomalies
  - No dirty read, i.e. no read of uncommitted data
  - No lost update
    - I.e., update made by a transaction is overwritten by another transaction that did not see the update)
  - No non-repeatable read
    - I.e., if read is executed again, it will see the same value
- Problems with SI
  - SI does not always give serializable executions
    - Serializable: among two concurrent txns, one sees the effects of the other
    - In SI: neither sees the effects of the other
  - Result: Integrity constraints can be violated

# Snapshot Isolation

- Example of problem with SI
  - Initially A = 3 and B = 17
    - Serial execution: A = ??, B = ??
    - if both transactions start at the same time, with snapshot isolation: A = ?? , B = ??
- Called **skew write**
- Skew also occurs with inserts
  - E.g:
    - Find max order number among all orders
    - Create a new order with order number = previous max + 1
    - Two transaction can both create order with same number
      - Is an example of phantom phenomenon

| $T_i$ | $T_j$ |
|---|---|
| read($A$) | |
| read($B$) | |
| | read($A$) |
| | read($B$) |
| A=B | |
| | B=A |
| write($A$) | |
| | write($B$) |

## Snapshot Isolation Anomalies

- SI breaks serializability when transactions modify *different* items, each based on a previous state of the item the other modified
  - Not very common in practice
    - E.g., the TPC-C benchmark runs correctly under SI
    - when txns conflict due to modifying different data, there is usually also a shared item they both modify, so SI will abort one of them
  - But problems do occur
    - Application developers should be careful about write skew
- SI can also cause a read-only transaction anomaly, where read-only transaction may see an inconsistent state even if updaters are serializable
  - We omit details
- Using snapshots to verify primary/foreign key integrity can lead to inconsistency
  - Integrity constraint checking usually done outside of snapshot

## Serializable Snapshot Isolation

- **Serializable snapshot isolation (SSI)**: extension of snapshot isolation that ensures serializability
- Snapshot isolation tracks write-write conflicts, but does not track read-write conflicts
  - Where $T_i$ writes a data a data item Q, $T_j$ reads an earlier version of Q, but $T_j$ is serialized after $T_i$
- Idea: track read-write dependencies separately, and roll-back transactions where cycles can occur
  - Ensures serializability
  - Details in book
- Implemented in PostgreSQL from version 9.1 onwards
  - PostgreSQL implementation of SSI also uses index locking to detect phantom conflicts, thus ensuring **true serializability**

## SI Implementations

- Snapshot isolation supported by many databases
  - Including Oracle, PostgreSQL, SQL Server, IBM DB2, etc
  - Isolation level can be set to snapshot isolation
- Oracle implements "first updater wins" rule (variant of "first committer wins")
  - Concurrent writer check is done at time of write, not at commit time
  - Allows transactions to be rolled back earlier
- **Warning**: *even if isolation level is set to serializable, Oracle actually uses snapshot isolation*
  - Old versions of PostgreSQL prior to 9.1 did this too
  - Oracle and PostgreSQL < 9.1 do not support true serializable execution

# Working Around SI Anomalies

- Can work around SI anomalies for specific queries by using **select .. for update** (supported e.g. in Oracle)
  - Example
    - **select max**(orderno) **from** orders **for update**
    - read value into local variable maxorder
    - insert into orders (maxorder+1, …)
- **select for update (SFU) clause** treats all data read by the query as if it were also updated, preventing concurrent updates
- Can be added to queries to ensure serializability in many applications
  - Does not handle phantom phenomenon/predicate reads though

## Weak Levels of Concurrency

## Weak Levels of Consistency

- **Degree-two consistency**: differs from two-phase locking in that S-locks may be released at any time, and locks may be acquired at any time
  - X-locks must be held till end of transaction
  - Serializability is not guaranteed, programmer must ensure that no erroneous database state will occur]
- **Cursor stability**:
  - For reads, each tuple is locked, read, and lock is immediately released
  - X-locks are held till end of transaction
  - Special case of degree-two consistency

# Weak Levels of Consistency in SQL

- SQL allows non-serializable executions
  - **Serializable**: is the default
  - **Repeatable read**: allows only committed records to be read, and repeating a read should return the same value (so read locks should be retained)
    - However, the phantom phenomenon need not be prevented
      - T1 may see some records inserted by T2, but may not see others inserted by T2
  - **Read committed**: same as degree two consistency, but most systems implement it as cursor-stability
  - **Read uncommitted**: allows even uncommitted data to be read
- In most database systems, read committed is the default consistency level
  - Can be changed as database configuration parameter, or per transaction
    - **set isolation level serializable**

## Concurrency Control across User Interactions

- Many applications need transaction support across user interactions
  - Can’t use locking for long durations
- Application level concurrency control
  - Each tuple has a version number
  - Transaction notes version number when reading tuple
    - **select** r.balance, r.version **into** :A, :version
      **from** r **where** acctId =23
  - When writing tuple, check that current version number is same as the version when tuple was read
    - **update** r **set** r.balance = r.balance + :deposit, r.version = r.version+1
      **where** acctId = 23 **and** r.version = :version

## Concurrency Control across User Interactions

- Equivalent to **optimistic concurrency control without validating read set**
  - Unlike SI, reads are not guaranteed to be from a single snapshot.
  - Does not guarantee serializability
  - But avoids some anomalies such as “lost update anomaly”
- Used internally in Hibernate ORM system
- Implemented manually in many applications
- Version numbers stored in tuples can also be used to support first committer wins check of snapshot isolation

## Advanced topics in Concurrency Control

## Online Index Creation

- Problem: how to create an index on a large relation without affecting concurrent updates
  - Index construction may take a long time
  - Two-phase locking will block all concurrent updates
- Key ideas:
  - Build index on a snapshot of the relation, but keep track of all updates that occur after snapshot
    - Updates are not applied on the index at this point
  - Then apply subsequent updates to catch up
  - Acquire relation lock towards end of catchup phase to block concurrent updates
  - Catch up with remaining updates, and add index to system catalog
  - Subsequent transactions will find the index in catalog and update it

## Concurrency in Index Structures

- Indices are unlike other database items in that their only job is to help in accessing data.
- Index-structures are typically accessed very often, much more than other database items.
  - Treating index-structures like other database items, e.g. by 2-phase locking of index nodes can lead to low concurrency.
- There are several index concurrency protocols where locks on internal nodes are released early, and not in a two-phase fashion.
  - It is acceptable to have nonserializable concurrent access to an index as long as the accuracy of the index is maintained.
    - In particular, the exact values read in an internal node of a $B^+$-tree are irrelevant so long as we land up in the correct leaf node.

## Concurrency in Index Structures (Cont.)

- **Crabbing protocol** used instead of two-phase locking on the nodes of the $B^+$-tree during search/insertion/deletion:
  - First lock the root node in shared mode.
  - After locking all required children of a node in shared mode, release the lock on the node
  - During insertion/deletion, upgrade leaf node locks to exclusive mode.
  - When splitting or coalescing requires changes to a parent, lock the parent in exclusive mode.
- Above protocol can cause excessive deadlocks
  - Searches coming down the tree deadlock with updates going up the tree
  - Can abort and restart search, without affecting transaction
- The **B-link tree locking protocol** improves concurrency
  - Intuition: release lock on parent before acquiring lock on child
    - And deal with changes that may have happened between lock release and acquire

# Concurrency Control in Main-Memory Databases

- Index locking protocols can be simplified with main-memory databases
  - Short term lock can be obtained on entire index for duration of an operation, serializing updates on the index
    - Avoids overheads of multiple lock acquire/release
    - No major penalty since operations finish fast, since there is no disk wait
- Latch-free techniques for data-structure update can speed up operations further

# Latch-Free Data-structure Updates

- This code is not safe without latches if executed concurrently:

```
insert(value, head) {
    node = new node
    node−>value = value
    node−>next = head
    head = node
}
```

- This code is safe

```
insert latchfree(head, value) {
    node = new node
    node−>value = value
    repeat
        oldhead = head
        node−>next = oldhead
        result = CAS(head, oldhead, node)
    until (result == success)
}
```

## Latch-Free Data-structure Updates

- This code is not safe without latches if executed concurrently:

```
insert(value, head) {
    node = new node
    node−>value = value
    node−>next = head
    head = node
}
```

- This code is safe

```
insert latchfree(head, value) {
    node = new node
    node−>value = value
    repeat
        oldhead = head
        node−>next = oldhead
        result = CAS(head, oldhead, node)
    until (result == success)
}
```

# Latch-Free Data-structures (Cont.)

- *Consider:*

```
delete latchfree(head) {
    /* This function is not quite safe; see explanation in text. */
    repeat
        oldhead = head
        newhead = oldhead−>next
        result = CAS(head, oldhead, newhead)
    until (result == success)
}
```

- Above code is almost correct, but has a concurrency bug
  - P1 initiates delete with N1 as head; concurrently P2 deletes N1 and next node N2, and then reinserts N1 as head, with N3 as next
  - P1 may set head as N2 instead of N3.
- Known as ABA problem
- See book for details of how to avoid this problem

## Concurrency Control with Operations

- Consider this non-two phase schedule, which preserves database integrity constraints
- Can be understood as transaction performing increment operation
  - E.g., increment(A, -50), increment (B, 50)
  - As long as increment operation does not return actual value, increments can be reordered
    - ***Increments commute***
  - New increment-mode lock to support reordering
  - Conflict matrix with increment lock mode
    - *Two increment operations do not conflict with each other*

| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| $A := A - 50$ | |
| write($A$) | |
| | read($B$) |
| | $B := B - 10$ |
| | write($B$) |
| read($B$) | |
| $B := B + 50$ | |
| write($B$) | |
| | read($A$) |
| | $A := A + 10$ |
| | write($A$) |

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

## Concurrency Control with Operations (Cont.)

- Undo of increment(v, n) is performed by increment (v, -n)
- Increment_conditional(v, n):
  - Updates v by adding n to it, as long as final v > 0, fails otherwise
  - Can be used to model, e.g. number of available tickets, *avail_tickets,* for a concert
  - Increment_conditional is NOT commutative
    - E.g., last few tickets for a concert
  - But reordering may still be acceptable

## Real-Time Transaction Systems

- Transactions in a system may have deadlines within which they must be completed.
  - Hard deadline: missing deadline is an error
  - Firm deadline: value of transaction is 0 in case deadline is missed
  - Soft deadline: transaction still has some value if done after deadline
- Locking can cause blocking
- Optimistic concurrency control (validation protocol) has been shown to do well in a real-time setting

# End of Chapter 18