Concurrency Control

Chapter 17

Concurrency Control

- In this chapter we discuss about concurrency control algorithms.

Outline

- 2PL, Serializability and Recoverability
- Lock Management
- Lock Conversions
- Dealing with Deadlocks
- Concurrency control in B+trees
- Concurrency control without locking

A serial schedule

T1:	R(A)W(A)		
T2:		R(A)W(A)W(C)	
T3:			R(B)W(A)

- Assumption: Each Ti is correct when executed individually, i.e., all serial schedules are valid.

- Objective: Accept schedules “equivalent” to a serial schedule (serializable schedules).

- what do we mean by “equivalent”.
- There are two notions of equivalent
  - Conflict equivalent or conflict serializable
  - View equivalent or view serializable

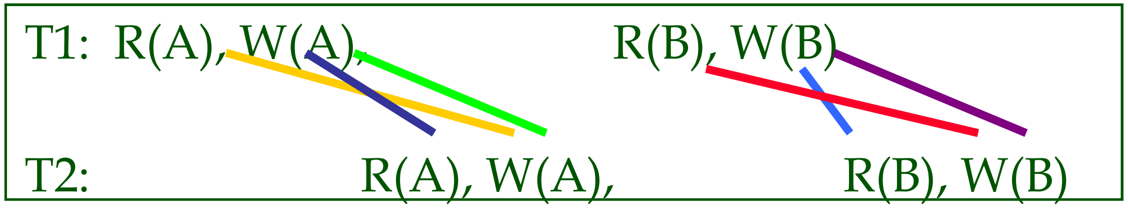
Conflict Serializable Schedules

- Two operations conflict if one is a W(A) and the other is R(A) or W(A).
- Two schedules are conflict equivalent if:
  - They involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered **the same way**
- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

Precedence graph

- A node for each committed transaction
- An arc from Ti to Tj if an action of Ti precedes and conflicts with one of Tj's actions.

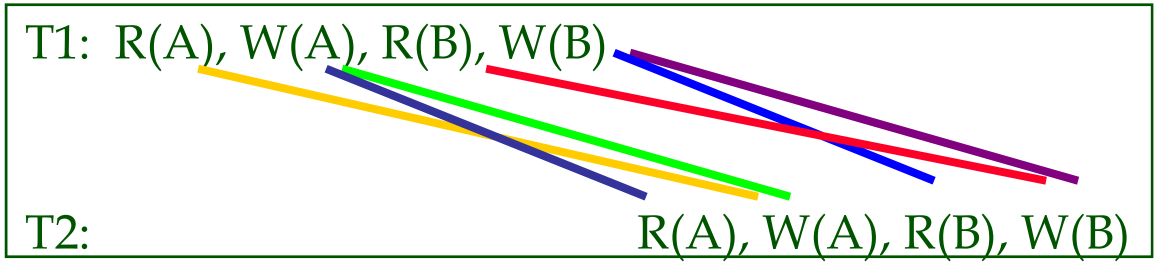
B:



$$\begin{tikzcd} T1 \arrow[r] & T2 \end{tikzcd}$$

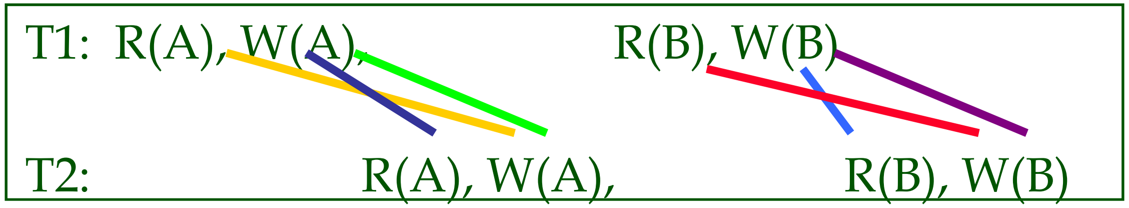
It is conflict equivalent to the following serial schedule

B1:



$$\begin{tikzcd} T1 \arrow[r] & T2 \end{tikzcd}$$

Precedence Graph



$$\begin{tikzcd}T1 \arrow[r] & T2\end{tikzcd}$$

- W(A) and R(A) conflict, W(A) and W(A) conflict
- T = {T_1,….T_n}, a set of transactions.

- The Precedence graph of a schedule S is a directed graph G(S) = (V,E), where
  - V = {T_1, …T_n} is a set of vertices;
  - E consists of edges (T_i,T_j) if

    one of T_i’s operations precedes and conflicts with one of T_j’s operations in S.

Example

- A schedule that is not conflict serializable:

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	

$$\begin{tikzcd} T1 \arrow[r, bend left, "A"] & T2 \arrow[l, bend left, "B"] \end{tikzcd}$$

Precedence graph

- The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

C:

T1:	$R_1(X)$		$W_1(X)$	
T2:		$W_2(X)$		
T3:				$R_3(X)$

$$\begin{tikzcd}
 & T2 \arrow[dl] \arrow[dr, bend left] & \\
T3 & & T1 \arrow[ul, bend left] \arrow[ll]
\end{tikzcd}$$

D:

T1:			W(Y)	
T2:	R(X),			R(Y), W(Z)
T3:		W(X),		

$$\begin{tikzcd}
 & T2 \arrow[dl] & \\
T3 & & T1 \arrow[ul]
\end{tikzcd}$$

It is conflict equivalent to the following serial schedule

D1:

T1:	W(Y)		
T2:		R(X), R(Y), W(Z)	
T3:			W(X),

D:

```
T1:        R(X),        W(X)
T2: W(X),
T3:              R(X)
```

$$\begin{tikzcd}
 & T2 \arrow[dl] \arrow[dr] & \\
T3 \arrow[rr] & & T1
\end{tikzcd}$$

- It is conflict equivalent to the following serial schedule

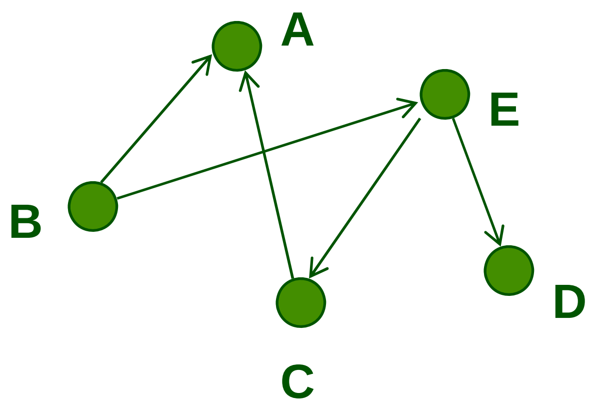
D1:

```
T1:                   R(X), W(X)
T2: W(X),
T3:         R(X)
```

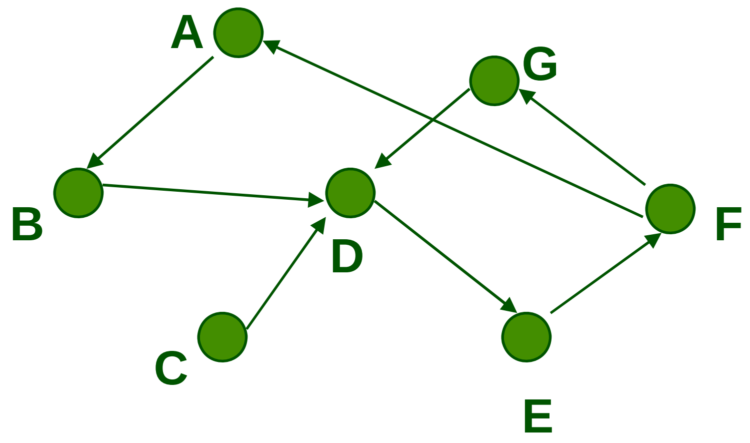
- Interesting question: by looking at the precedence graph, can you figure out how to find an equivalent serial schedule for a serializable schedule?

- Cycle in a graph
  - A cycle is a path that starts and terminates at the same node

- Examples



Does not contain cycle (acyclic)



Contains cycles

- Theorem: Schedule is conflict serializable if and only if its precedence graph is acyclic

- The serialization order can be obtained through topological sorting, which determines a linear order consistent with the partial order of the precedence graph.

D:

```
T1:         R(X),          W(X)
T2: W(X),
T3:               R(X)
```

$$\begin{tikzcd}
 & T2 \arrow[dl] \arrow[dr] & \\
T3 \arrow[rr] & & T1
\end{tikzcd}$$

- It is conflict equivalent to the following serial schedule

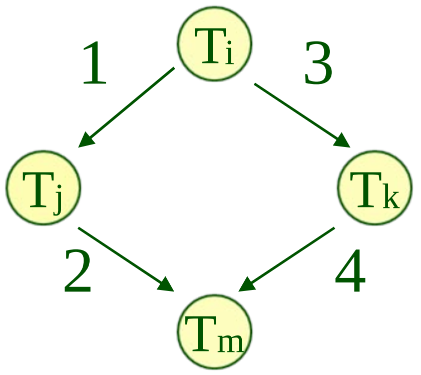
D1:

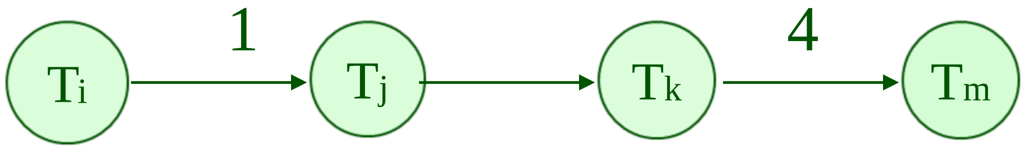
```
T1:                  R(X), W(X)
T2: W(X),
T3:         R(X)
```

$$\begin{tikzcd}
T2 \arrow[r] & T3 \arrow[r] & T1
\end{tikzcd}$$

a linear order consistent with the partial order of the precedence graph.

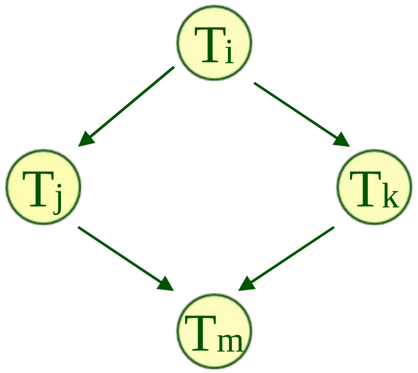
Linear order: Transitive, anti-symmetric and total
Partial order: reflexive, anti-symmetric and transitive





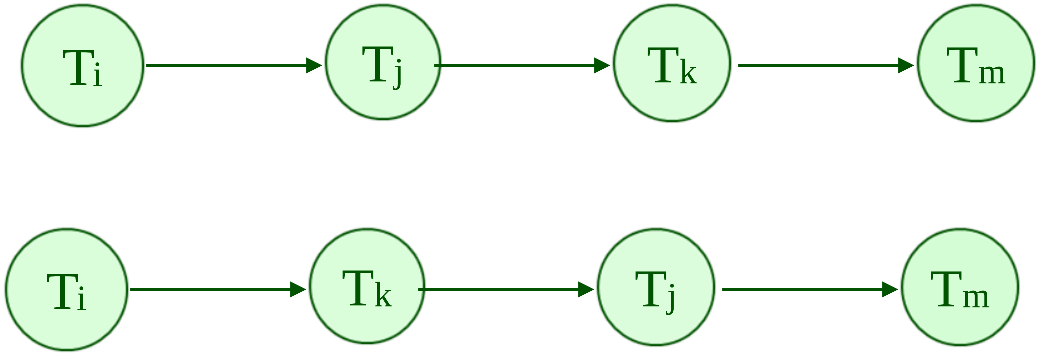
- topological sorting : a linear order consistent with the partial order of the precedence graph.

- $w_i[a]\ r_k[a]\ r_i[b]\ w_j[b]\ w_k[c]\ r_m[c]\ w_j[d]\ r_m[d]$


$$\begin{tikzcd} & T_i \arrow[dl] \arrow[dr] & \\ T_j \arrow[dr] & & T_k \arrow[dl] \\ & T_m & \end{tikzcd}$$

Precedence graph

Two possible serialization orders


$$\begin{tikzcd} T_i \arrow[r] & T_j \arrow[r] & T_k \arrow[r] & T_m \\ T_i \arrow[r] & T_k \arrow[r] & T_j \arrow[r] & T_m \end{tikzcd}$$

Review: Strict 2PL

- *Strict Two-phase Locking (Strict 2PL) Protocol*:
  - Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- Strict 2PL allows only schedules whose precedence graph is acyclic

Two-Phase Locking (2PL)

- Two-Phase Locking Protocol
- Relaxes the second rule of 2PL
  - Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - A transaction can not request additional locks once it releases any locks.
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- Transaction has a growing phase which acquires locks and shrinking phase which releases locks.
- More concurrency than strict 2PL

Proof

- Precedence graph is acyclic.
- Suppose if the graph has a cycle
- It means, T1→T2→…Tn→T1
- T2 gets the locks after T1, T3 gets the lock after T2 Similarly T1 gets the lock after Tn. As a result, T1 gets the lock after T1 which is impossible.

Recoverability and Strict Schedules

- **Strict schedules**: If a value is written by T is not read or overwritten by other transactions until T either commits or aborts.
- Strict schedules are recoverable and avoid cascading aborts.
  - Actions of aborted transactions can be undone.

View Serializability

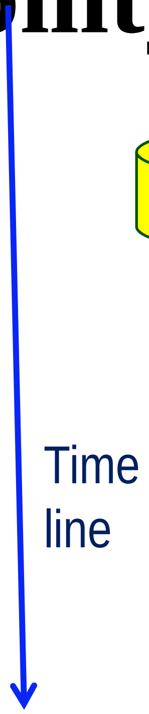
- Equivalent: same effects

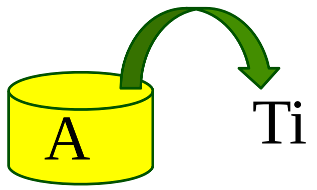
- The effects of a history are the values produced by the Write operations of un-aborted transactions.

- A schedule is view serializable if it is view equivalent to a serial schedule.

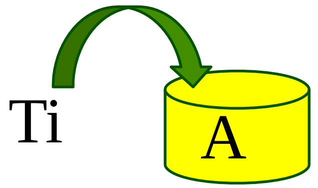
View Serializability

- Schedules S1 and S2 are view equivalent if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2





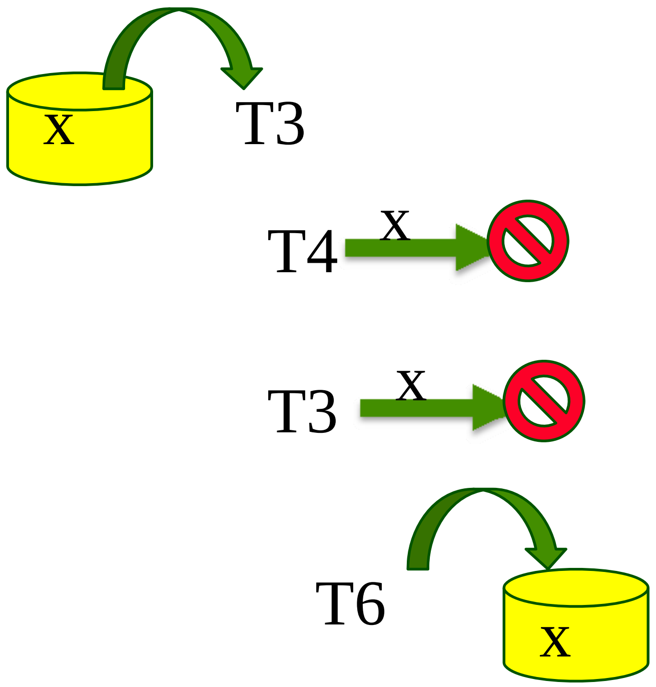
$$Tj \xrightarrow{A} Ti$$

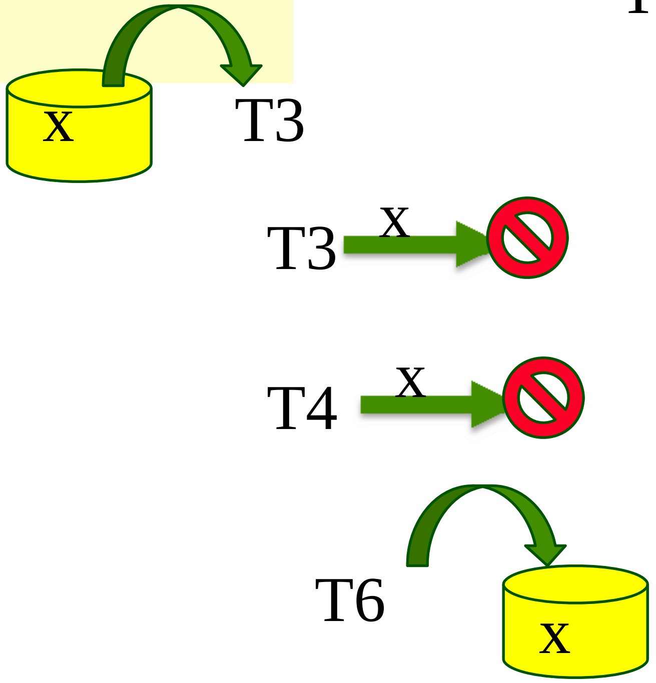


```
T1: R(A)          W(A)
T2:        W(A)
T3:                     W(A)
```

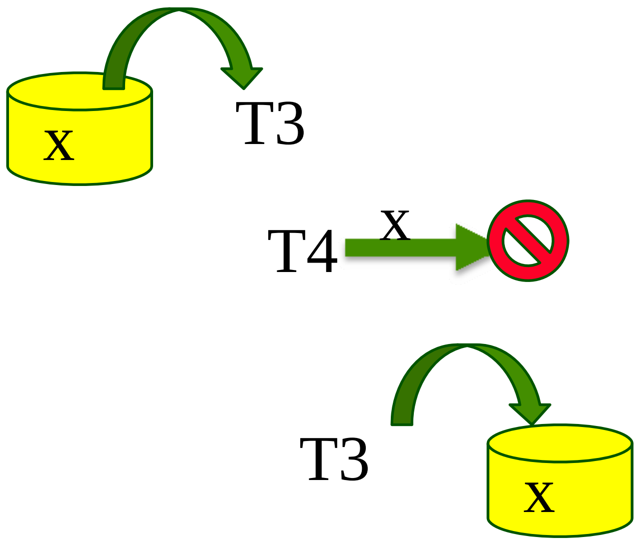
```
T1: R(A),W(A)
T2:                W(A)
T3:                      W(A)
```

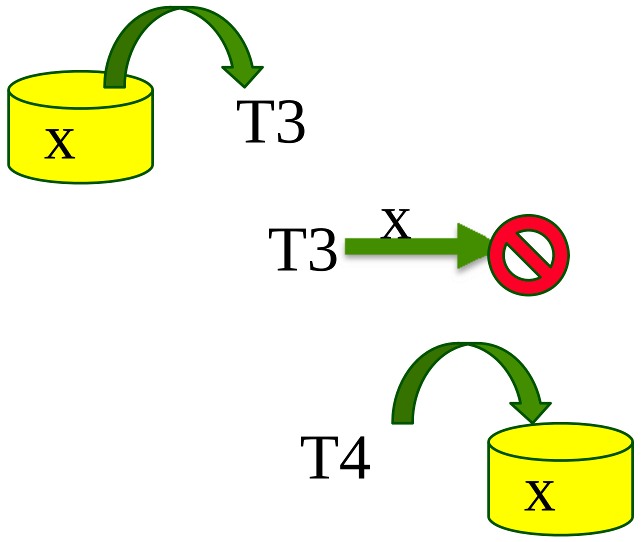
- ❖ $r_3[x]w_4[x]w_3[x]w_6[x]$
- T_3 read-from T_b.
- The final write for *x* is $w_6[x]$.
- View equivalent to $T_3\ T_4\ T_6$:
- r3[x]w3[x]w4[x]w6[x]

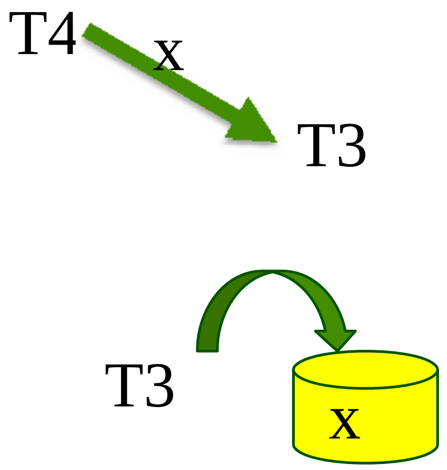




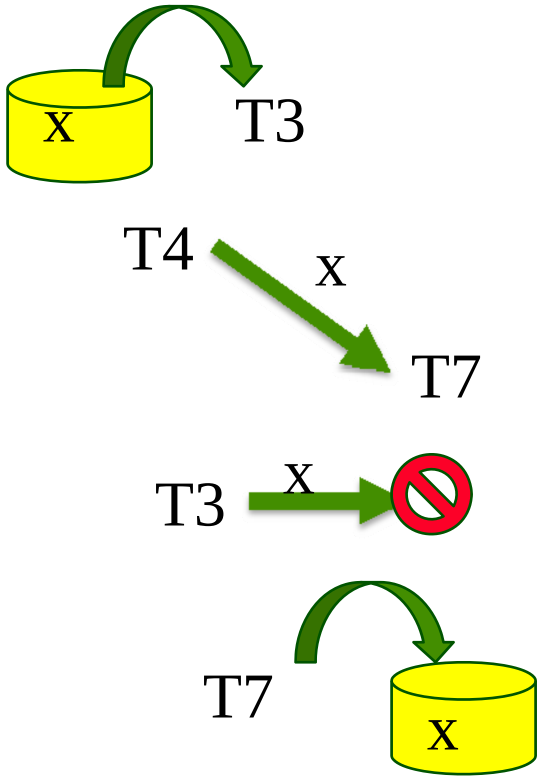
- $r_3[x]\ w_4[x]\ w_3[x]$
  - T_3 read-from T_b.
  - The final write for *x* is $w_3[x]$.
  - Not serializable.
- r3[x] w3[x] w4[x]
- w4[x] r3[x] w3[x]

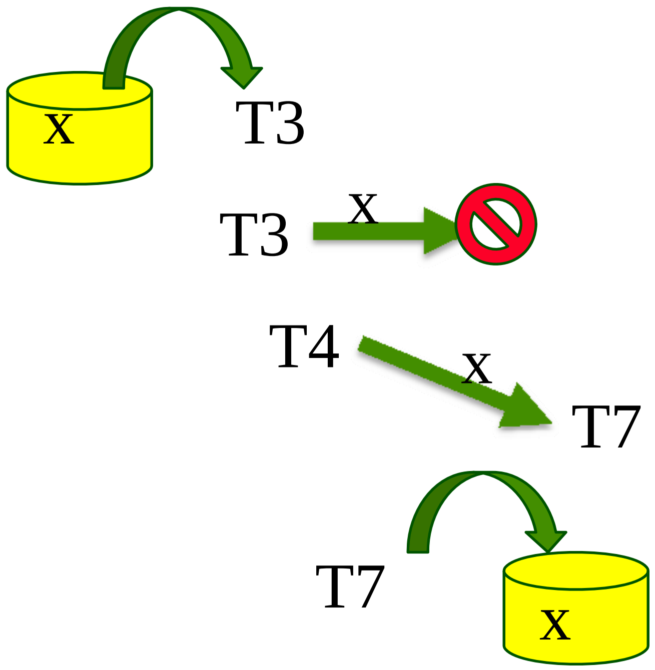






- $r_3[x]\ w_4[x]\ r_7[x]\ w_3[x]\ w_7[x]$
  - T_3 read-from T_b.
  - T_7 read-from T_4.
  - The final write for x is $w_7[x]$.
  - View equivalent to $T_3\ T_4\ T_7$.
- r3[x] w3[x] w4[x]r7[x]w7[x]

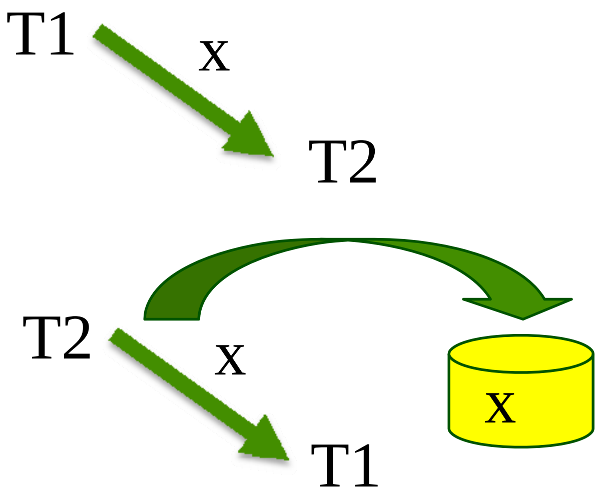


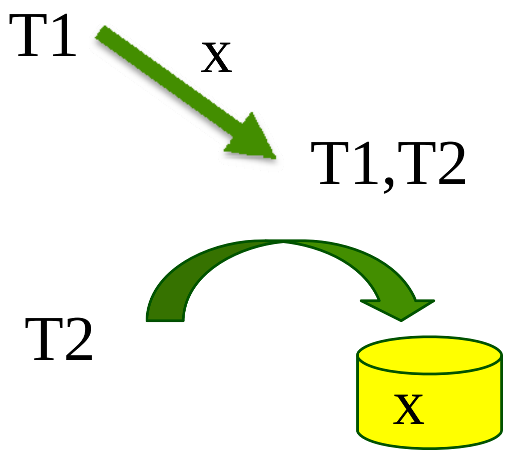


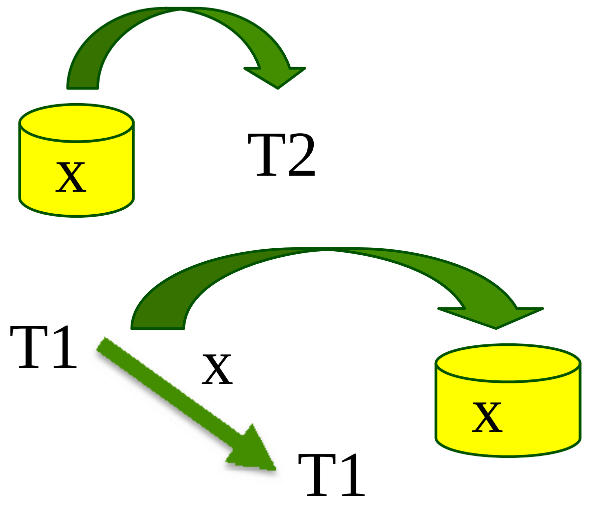
$w_1[x]\ r_2[x]\ w_2[x]\ r_1[x]$

- T_2 read-from T_1.
- T_1 read-from T_2
- The final write for *x* is $w_2[x]$.
- Not serializable.

w1[x] r1[x] r2[x] w2[x]
r2[x] w2[x] w1[x] r1[x]

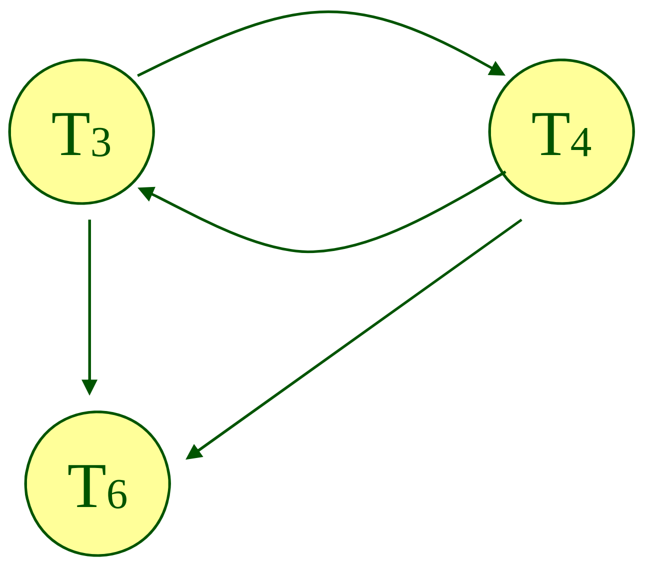






Testing whether a schedule is view serializable is NP-complete
Enforcing is expensive. So, not used.

- $r_3[x]\ w_4[x]\ w_3[x]\ w_6[x]$



- Not conflict serializable, as there is a cycle in the precedence graph.
- But view serializable, equivalent to T3, T4, T6
- $r_3[x]\ w_3[x]\ w_4[x]\ w_6[x]$

Lock Management

- Lock and unlock requests are handled by the **lock manager**
- Lock manager maintains a **lock table**.
- Lock table entry for an object which can be a page or record contains:
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- DBMS also maintains a **transaction table** which maintains a pointer to a list of locks held by a transaction.

Lock and Unlock Requests

- As per 2PL
  - Transaction can only read after obtaining a shared lock
  - Transaction can only write after obtaining a write lock
- When a transaction wants a lock request it issues a request to lock manager.
  - If shared lock is requested if the object is not locked in exclusive mode, it grants a lock.
  - If an exclusive lock is requested and no transcation currenty holds a lock, the lock manager grants a lock.
  - Otherwise the requested lock can not be granted and lock request is added to the queue of lock requests. The transcation which is requesting lock is suspended.
- When transaction commits or aborts, it releases all the locks. The next request is woken-up and granted the lock.

Atomicity of locking and unlocking

- Locking and unlocking have to be atomic operations
- So access to a lock table should be guarded by an OS synchronization mechanism such as a semaphore.
- Example:
  - When a transaction requests a lock, the lock manager may grant a lock as no transaction holds a lock. But in the mean time another transaction may have requested and obtained a lock.
  - So, the sequence of operations in a lock request call (checking to see if the request can be granted, updating a lock table, etc) muat be implemented as an atomic operations.

- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock

Latches, Convoys

- Locks are held for longer duration
- DBMS supports for latches: to read or write a page.
- Latches are unset immediately after the physical read or write operation is completed.
- Convoy: Interleaving interacts with OS scheduling of processes can lead to a situation called convoy.
  - For example OS suspends the transaction which locks heavily used object.
- Convoys tend to be stable and leads to performance problems.

Lock Conversions

- **Lock upgrade**: transaction that holds a shared lock can be upgraded to hold an exclusive lock
  - SQL update initially obtains shared lock on each row, if row satisfies condition, the shared lock is upgraded to exclusive lock.
- Lock upgrade request must be handled by lock manager
  - May lead to deadlock
- Better approach.
  - Obtain exclusive locks initially and then downgrade them.
  - This approach improves throughput by reducing deadlocks and used in commercial systems.

Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection

Deadlock Prevention

- Assign priorities based on timestamps. Assume Ti wants a lock that Tj holds. Two policies are possible:
  - **Wait-Die**: If Ti has higher priority, Ti waits for Tj; otherwise Ti aborts
  - **Wound-wait**: If Ti has higher priority, Tj aborts; otherwise Ti waits
- If a transaction re-starts, make sure it has its original timestamp

Conservative 2PL

- Under conservative 2PL, a transaction obtains all the locks at the beginning or waits for the locks to be come available.
- This scheme ensures that there will be no deadlocks.
- It is difficult to know locks in advance

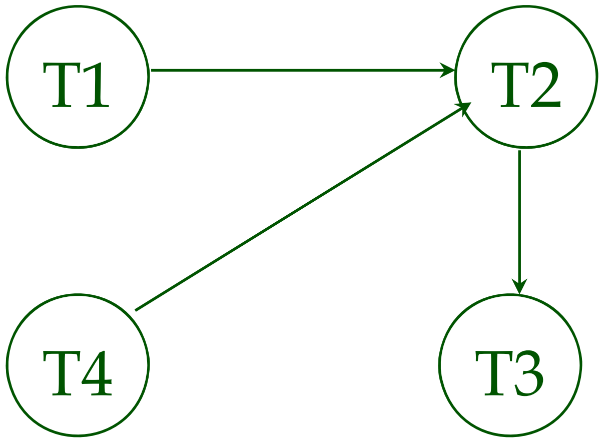
Deadlock Detection

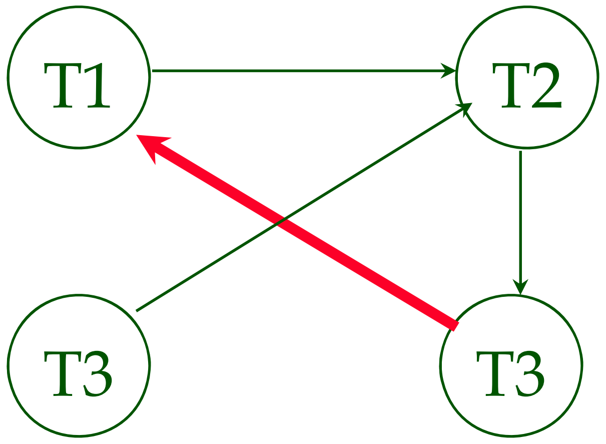
- Create a waits-for graph:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Periodically check for cycles in the waits-for graph

Deadlock Detection (Continued)

Example:

T1:	S(A), R(A),		S(B)				
T2:		X(B),W(B)			X(C)		
T3:				S(C), R(C)			X(A)
T4:						X(B)	





Specialized Locking Protocols

- Dynamic databases and phantom problem
- Multi granularity locking

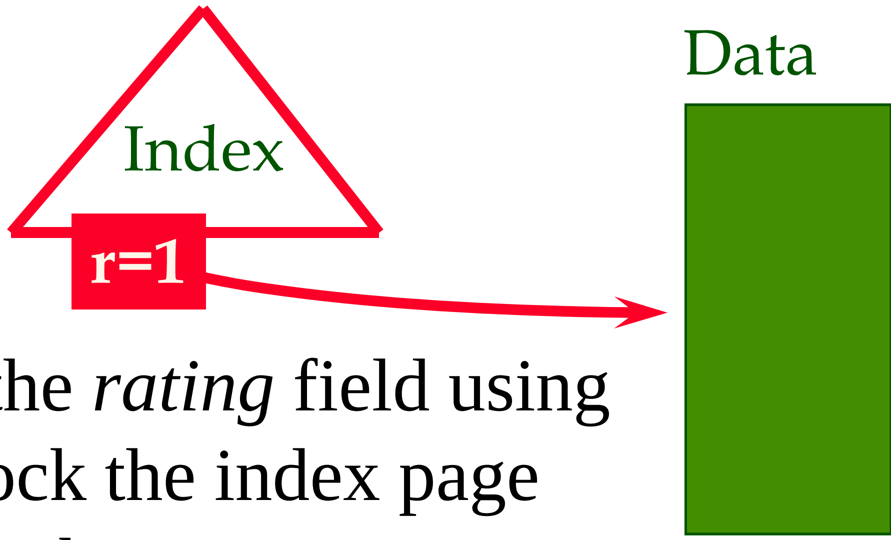
Dynamic Databases

- If we relax the assumption that the DB is a fixed collection of objects, even Strict 2PL will not assure serializability:
  - T1 locks all pages containing sailor records with *rating* = 1, and finds oldest sailor (say, *age* = 71).
  - Next, T2 inserts a new sailor; *rating* = 1, *age* = 96.
  - T2 also deletes oldest sailor with rating = 2 (and, say, *age* = 80), and commits.
  - T1 now locks all pages containing sailor records with *rating* = 2, and finds oldest (say, *age* = 63).
- No consistent DB state where T1 is “correct”!

The Problem

- T1 implicitly assumes that it has locked the set of all sailor records with *rating* = 1.
  - Assumption only holds if no sailor records are added while T1 is executing!
  - Need some mechanism to enforce this assumption. (Index locking and predicate locking.)
- Example shows that conflict serializability guarantees serializability only if the set of objects is fixed!

Index Locking



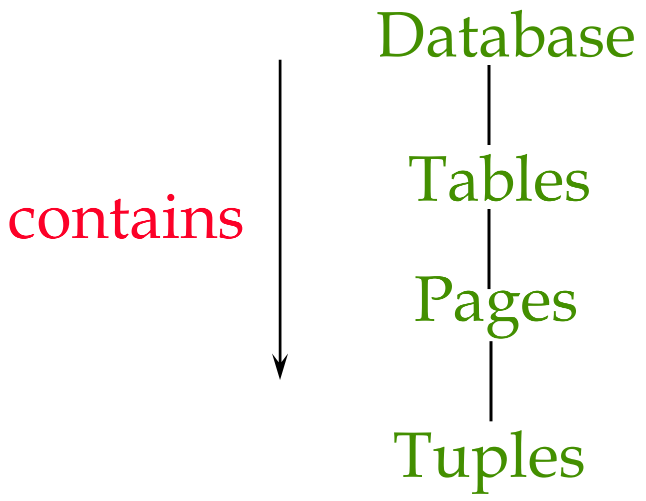
- If there is a dense index on the *rating* field using Alternative (2), T1 should lock the index page containing the data entries with *rating* = 1.
  - If there are no records with *rating* = 1, T1 must lock the index page where such a data entry *would* be, if it existed!
- If there is no suitable index, T1 must lock all pages, and lock the file/table to prevent new pages from being added, to ensure that no new records with *rating* = 1 are added.

Predicate Locking

- Grant lock on all records that satisfy some logical predicate, e.g. *age > 2*salary*.
- Index locking is a special case of predicate locking for which an index supports efficient implementation of the predicate lock.
- In general, predicate locking has a lot of locking overhead.

Multiple-Granularity Locks

- Hard to decide what granularity to lock (tuples vs. pages vs. tables).
- Shouldn't have to decide!
- Data "containers" are nested:



Solution: New Lock Modes, Protocol

- Allow Xacts to lock at each level, but with a special protocol using new "intention" locks:
  - Intention shared lock (IS)
  - Intention exclusive lock (IX)

- Before locking an item, Xact must set "intention locks" on all its ancestors.
- For unlock, go from specific to general (i.e., bottom-up).
- SIX mode: Like S & IX at the same time.

	--	IS	IX	S	X
--	√	√	√	√	√
IS	√	√	√	√	
IX	√	√	√		
S	√	√		√	
X	√				

Multiple Granularity Lock Protocol

- Each Xact starts from the root of the hierarchy.
  - To get S on a node, must hold IS on a parent node.
  - To get X on a node, must hold IX on parent node.
- To get X or IX or SIX on a node, must hold IX or SIX on parent node.
- Must release locks in bottom-up order.

Protocol is correct in that it is equivalent to directly setting locks at the leaf levels of the hierarchy.

Lock escalation

- Locks should be obtained from root to leaf
- Locks should be released from leaf to root
- How to decide the granularity of locks ?
- Follow lock escalation
  - Obtain the fine granularity of locks
  - If number of locks exceed certain level, start obtaining locks at the next higher granularity.

Examples

- T1 scans R, and updates a few tuples:
  - T1 gets an SIX lock on R, then repeatedly gets an S lock on tuples of R, and occasionally upgrades to X on the tuples.
- T2 uses an index to read only part of R:
  - T2 gets an IS lock on R, and repeatedly gets an S lock on tuples of R.
- T3 reads all of R:
  - T3 gets an S lock on R.
  - OR, T3 could behave like T2; can use lock escalation to decide which.

	--	IS	IX	S	X
--	√	√	√	√	√
IS	√	√	√	√	
IX	√	√	√		
S	√	√		√	
X	√				

Locking in B+ Trees

- How can we efficiently lock a particular leaf node?
  - Btw, don’t confuse this with multiple granularity locking!
- One solution: Ignore the tree structure, just lock pages while traversing the tree, following 2PL.
- This has terrible performance!
  - Root node (and many higher level nodes) become bottlenecks because every tree access begins at the root.

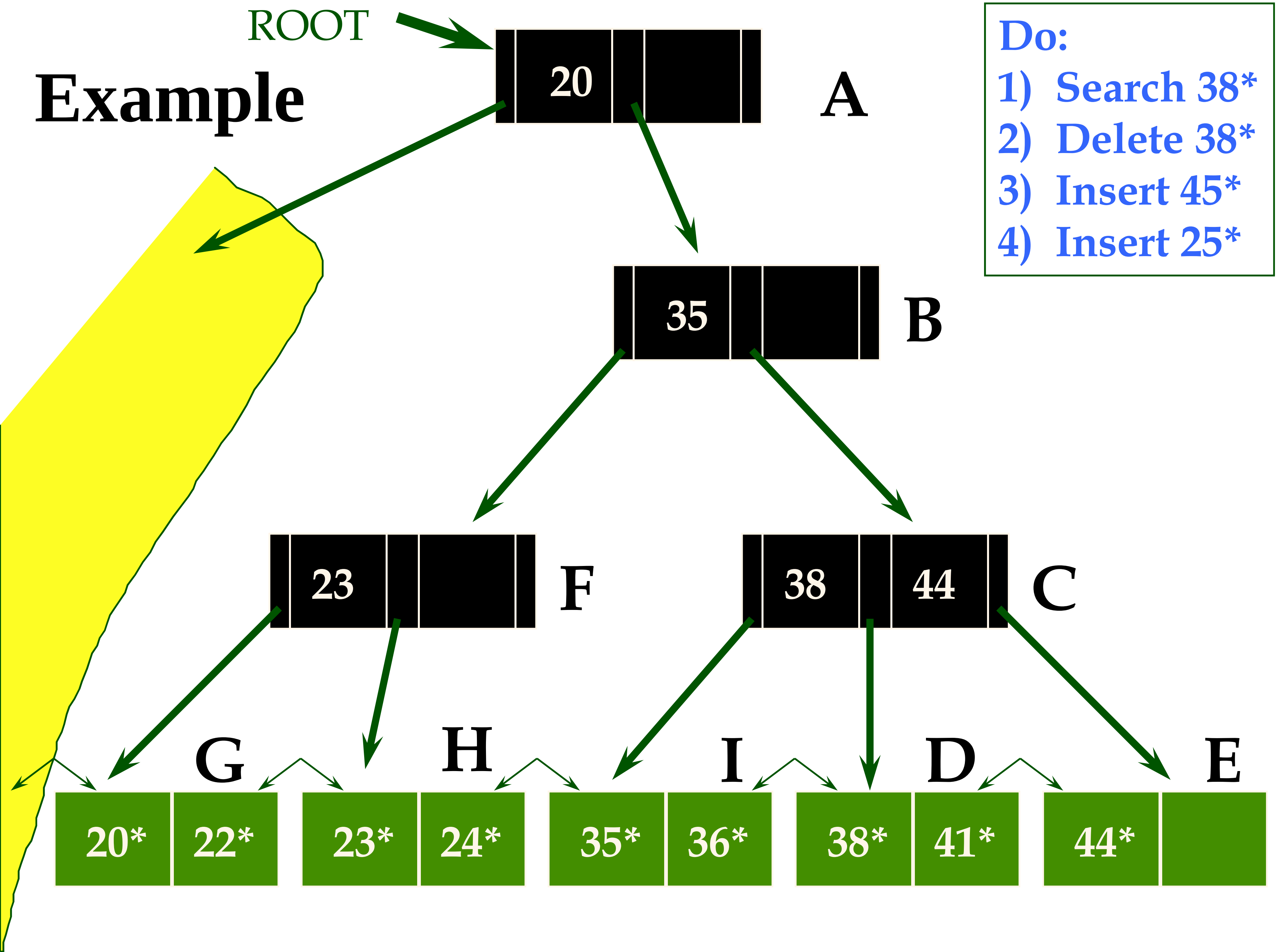
Two Useful Observations

- Higher levels of the tree only direct searches for leaf pages.
- For inserts, a node on a path from root to modified leaf must be locked (in X mode, of course), only if a split can propagate up to it from the modified leaf. (Similar point holds w.r.t. deletes.)
- We can exploit these observations to design efficient locking protocols that guarantee serializability *even though they violate 2PL.*

A Simple Tree Locking Algorithm (Lock coupling or Crabbing)

- Search: Start at root and go down; repeatedly, S lock child then unlock parent.
- Insert/Delete: Start at root and go down, obtaining X locks as needed. Once child is locked, check if it is <u>safe</u>:
  - If child is safe, release all locks on ancestors.
- Safe node: Node such that changes will not propagate up beyond this node.
  - Inserts: Node is not full.
  - Deletes: Node is not half-empty.

Example



Do:
1) Search 38*
2) Delete 38*
3) Insert 45*
4) Insert 25*

Example

- Search 38: obtain S lock on node A, determine node B, obtain S lock on B, release S lock on node A, obtain S lock on C and release the S lock on node B, obtain S lock on node D and release the S lock node C.
- If some other transaction wants to delete 38, it will wait till the first transaction is done.
- Insert 45: Since D has the space, insertion would be no problem.
- Insert 25: Node H is full. It must be split also F has to be modified. It must request lock upgrade.
  - If another transaction holds lock, the first transaction must be suspended.
  - Deadlock might occur.

Concurrency Control Without Locking

- Optimistic Concurrency Control
- Timestamp-based Concurrency control
- Multi-version Concurrency Control

Optimistic CC (Kung-Robinson)

- Locking is a conservative approach in which conflicts are prevented. Disadvantages:
  - Lock management overhead.
  - Deadlock detection/resolution.
  - Lock contention for heavily used objects.
- If conflicts are rare, we might be able to gain concurrency by not locking, and instead checking for conflicts before Xacts commit.

Optimistic CC

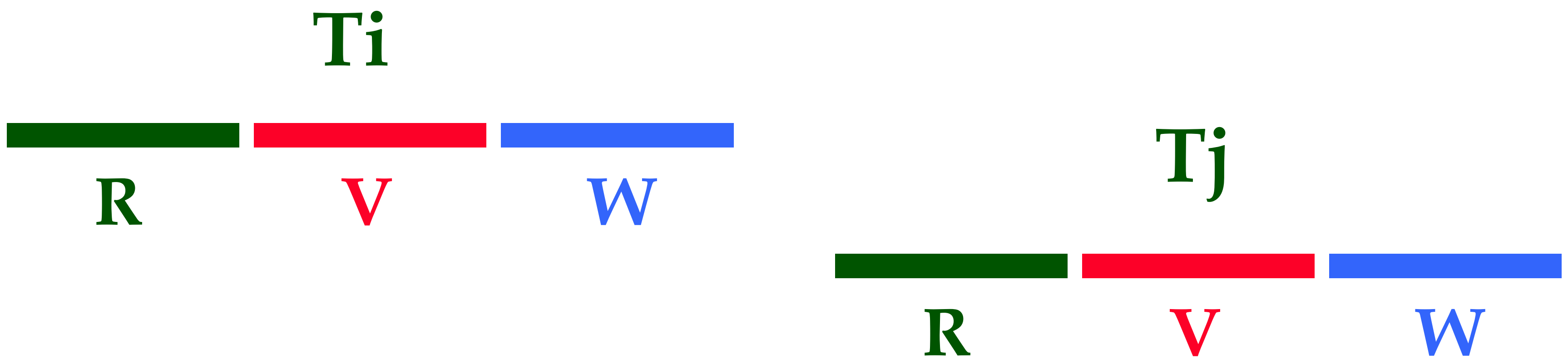
- Xacts have three phases:
  - READ: Xacts read from the database, but make changes to private copies of objects.
  - VALIDATE:
    - If the transaction decides to commit, it checks for conflicts.
    - If there is a conflict, the transaction is aborted.
  - WRITE:
    - If there are no conflicts during validation phase, make local copies of changes public.
- Better for few conflicts
- Bad for more conflicts.

Validation

- Test conditions that are sufficient to ensure that no conflict occurred.
- Each Xact is assigned a numeric id.
  - Just use a **timestamp**.
- Xact ids assigned at end of READ phase, just before validation begins.
- ReadSet(Ti): Set of objects read by Xact Ti.
- WriteSet(Ti): Set of objects modified by Ti.
- For all i and j such that Ti < Tj, **check if one of the three conditions hold.**

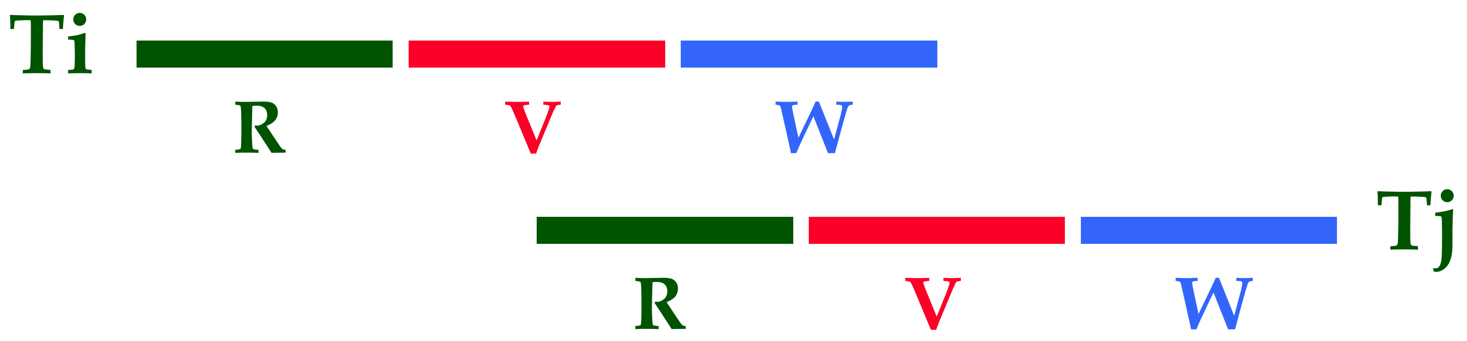
Test 1

- For all i and j such that Ti < Tj, check that Ti completes before Tj begins.



Test 2

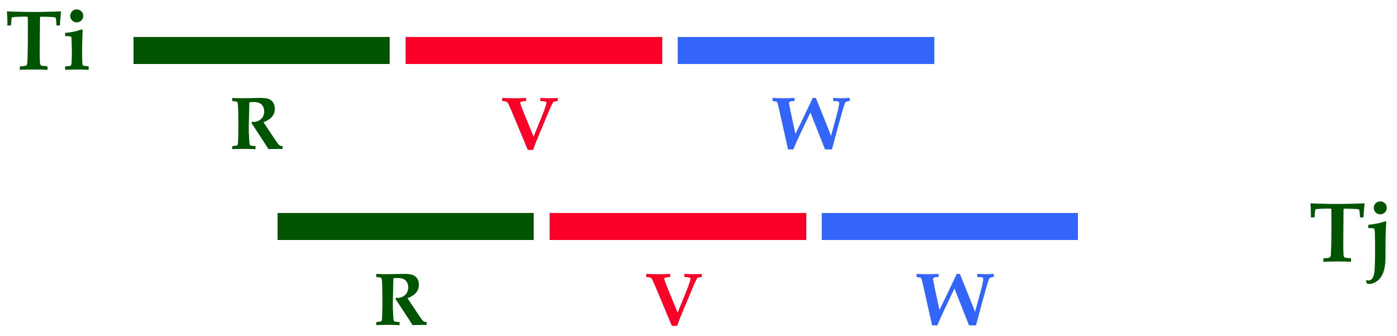
- For all i and j such that Ti < Tj, check that:
  - Ti completes before Tj begins its Write phase +
  - WriteSet(Ti) $\cap$ ReadSet(Tj) is empty.



•Tj reads whicle Ti is writing objects, but there is no conflict as there is no intersection
•Tj overwrites some of the objects of Ti. But it is OK as All of Ti's writes precede all of Tj's writes.

Test 3

- For all i and j such that Ti < Tj, check that:
  - Ti completes Read phase before Tj does +
  - WriteSet(Ti) $\cap$ ReadSet(Tj) is empty +
  - WriteSet(Ti) $\cap$ WriteSet(Tj) is empty.



• Even though both can write at same time, the data objects can not overlap.

Applying Tests 1 & 2: Serial Validation

- To validate Xact T:

```
valid = true;
// S = set of Xacts that committed after Begin(T)
< foreach  Ts in S do {
   if ReadSet(Ts)  intersect WriteSet(T)
         then valid = false;
   }
   if valid then { install updates; // Write phase
                    Commit T } >
               else Restart T
```

end of critical section

Comments on Serial Validation

- Applies Test 2, with T playing the role of Tj and each Xact in Ts (in turn) being Ti.
- Assignment of Xact id, validation, and the Write phase are inside a **critical section**!
  - I.e., Nothing else goes on concurrently.
  - If Write phase is long, major drawback.
- Optimization for Read-only Xacts:
  - Don’t need critical section (because there is no Write phase).

Overheads in Optimistic CC

- Must record read/write activity in ReadSet and WriteSet per Xact.
  - Must create and destroy these sets as needed.
- Must check for conflicts during validation, and must make validated writes ``global''.
  - Critical section can reduce concurrency.
  - Scheme for making writes global can reduce clustering of objects.
- Optimistic CC restarts Xacts that fail validation.
  - Work done so far is wasted; requires clean-up.

``Optimistic'' 2PL

- If desired, we can do the following:
  - Set S locks as usual.
  - Make changes to private copies of objects.
  - Obtain all X locks at end of Xact, make writes global, then release all locks.
- In contrast to Optimistic CC as in Kung-Robinson, this scheme results in Xacts being blocked, waiting for locks.
  - However, no validation phase, no restarts (modulo deadlocks).

Timestamp CC

- **Idea:** Give each object a read-timestamp (RTS) and a write-timestamp (WTS), give each Xact a timestamp (TS) when it begins:
  - If action ai of Xact Ti conflicts with action aj of Xact Tj, and TS(Ti) < TS(Tj), then ai must occur before aj. Otherwise, restart violating Xact.

When Xact T wants to read Object O

- If TS(T) < WTS(O), this violates timestamp order of T w.r.t. writer of O.
  - So, abort T and restart it with a new, larger TS. (If restarted with same TS, T will fail again! Contrast use of timestamps in 2PL for ddlk prevention.)
- If TS(T) > WTS(O):
  - Allow T to read O.
  - Reset RTS(O) to max(RTS(O), TS(T))
- Change to RTS(O) on reads must be written to disk! This and restarts represent overheads.

When Xact T wants to Write Object O

- If TS(T) < RTS(O), this violates timestamp order of T w.r.t. writer of O; abort and restart T.
- If TS(T) < WTS(O), violates timestamp order of T w.r.t. writer of O.
  - **Thomas Write Rule: We can safely ignore such outdated writes; need not restart T! (T's write is effectively followed by another write, with no intervening reads.) Allows some serializable but non conflict serializable schedules:**
- Else, allow T to write O.

T1	T2
R(A)	
	W(A)
	Commit
W(A)	
Commit	

Timestamp CC and Recoverability

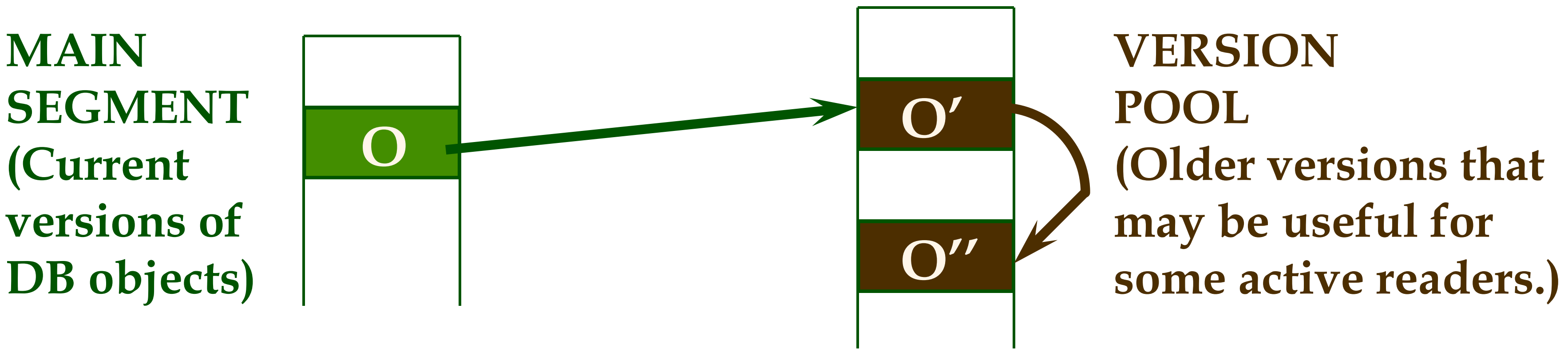
- Unfortunately, unrecoverable schedules are allowed:

T1	T2
W(A)	
	R(A)
	W(B)
	Commit

- Timestamp CC can be modified to allow only recoverable schedules:
  - Buffer all writes until writer commits (but update WTS(O) when the write is allowed.)
  - Block readers T (where TS(T) > WTS(O)) until writer of O commits.
- Similar to writers holding X locks until commit, but still not quite 2PL.

Multiversion Timestamp CC

- **Idea:** Let writers make a "new" copy while readers use an appropriate "old" copy:

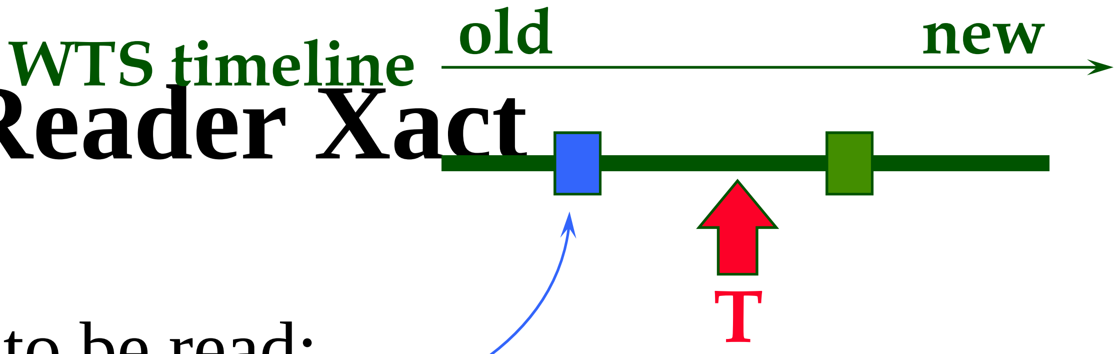

MAIN SEGMENT (Current versions of DB objects)
O
O′
O″
VERSION POOL (Older versions that may be useful for some active readers.)

- Readers are always allowed to proceed.
  - But may be blocked until writer commits.

Multiversion CC (Contd.)

- Each version of an object has its writer’s TS as its WTS, and the TS of the Xact that most recently read this version as its RTS.
- Versions are chained backward; we can discard versions that are “too old to be of interest”.
- Each Xact is classified as Reader or Writer.
  - Writer *may* write some object; Reader never will.
  - Xact declares whether it is a Reader when it begins.

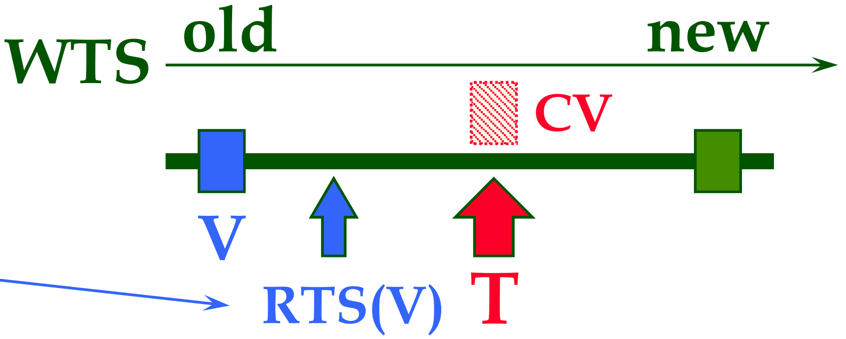
Reader Xact



- For each object to be read:
  - Finds **newest version** with WTS < TS(T). (Starts with current version in the main segment and chains backward through earlier versions.)
- Assuming that some version of every object exists from the beginning of time, Reader Xacts are never restarted.
  - However, might block until writer of the appropriate version commits.

Writer Xact

- To read an object, follows reader protocol.
- To write an object:
  - Finds **newest version V** s.t. WTS < TS(T).
  - If RTS(V) < TS(T), T makes a copy CV of V, with a pointer to V, with WTS(CV) = TS(T), RTS(CV) = TS(T). (Write is buffered until T commits; other Xacts can see TS values but can't read version CV.)
  - Else, reject write.



Transaction Support in SQL-92

- Each transaction has an access mode, a diagnostics size, and an isolation level.

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Problem
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeatable Reads	No	No	Maybe
Serializable	No	No	No

Summary

- There are several lock-based concurrency control schemes (Strict 2PL, 2PL). Conflicts between transactions can be detected in the dependency graph
- The lock manager keeps track of the locks issued. Deadlocks can either be prevented or detected.
- Naïve locking strategies may have the phantom problem

Summary (Contd.)

- Index locking is common, and affects performance significantly.
  - Needed when accessing records via index.
  - Needed for locking logical sets of records (index locking/predicate locking).
- Tree-structured indexes:
  - Straightforward use of 2PL very inefficient.
  - Bayer-Schkolnick illustrates potential for improvement.
- In practice, better techniques now known; do record-level, rather than page-level locking.

Summary (Contd.)

- Multiple granularity locking reduces the overhead involved in setting locks for nested collections of objects (e.g., a file of pages); should not be confused with tree index locking!
- Optimistic CC aims to minimize CC overheads in an ``optimistic'' environment where reads are common and writes are rare.
- Optimistic CC has its own overheads however; most real systems use locking.
- SQL-92 provides different isolation levels that control the degree of concurrency

Summary (Contd.)

- Timestamp CC is another alternative to 2PL; allows some serializable schedules that 2PL does not (although converse is also true).
- Ensuring recoverability with Timestamp CC requires ability to block Xacts, which is similar to locking.
- Multiversion Timestamp CC is a variant which ensures that read-only Xacts are never restarted; they can always read a suitable older version. Additional overhead of version maintenance.