

Concurrency Control

CHAPTER 17

SINA MERAJI

Conflict Serializable Schedules

Two schedules are **conflict equivalent** if:

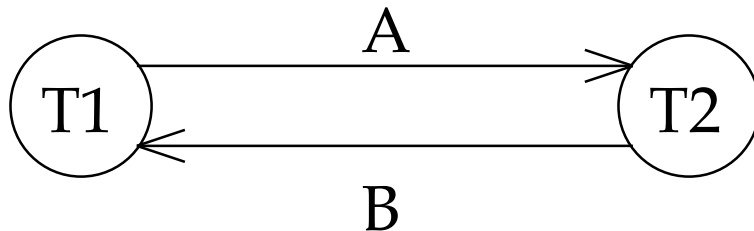
- Involve the same actions of the same transactions
- Every pair of conflicting actions is ordered the same way

Schedule S is **conflict serializable** if S is conflict equivalent to some serial schedule

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)	



Dependency graph

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

Dependency Graph

Dependency graph: One node per X_{act} ; edge from T_i to T_j if *an action of T_i precedes and conflicts with of T_j 's actions*

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

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

- 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.

View Serializability

Schedules S1 and S2 are **view equivalent** if:

- If T_i reads initial value of A in S1, then T_i also reads initial value of A in S2
- If T_i reads value of A written by T_j in S1, then T_i also reads value of A written by T_j in S2
- If T_i writes final value of A in S1, then T_i also writes final value of A in S2

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

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

Lock Management

Lock and unlock requests are handled by the lock manager

Lock table entry:

- Number of transactions currently holding a lock
- Type of lock held (shared or exclusive)
- Pointer to queue of lock requests

Locking and unlocking have to be atomic operations

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

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 T_i wants a lock that T_j holds. Two policies are possible:

- Wait-Die: If T_i has higher priority, T_i waits for T_j ; otherwise T_i aborts
- Wound-wait: If T_i has higher priority, T_j aborts; otherwise T_i waits

If a transaction re-starts, make sure it has its original timestamp

Deadlock Detection

Create a **waits-for graph**:

- Nodes are transactions
- There is an edge from T_i to T_j if T_i is waiting for T_j to release a lock

Periodically check for cycles in the waits-for graph

Deadlock Detection (Continued)

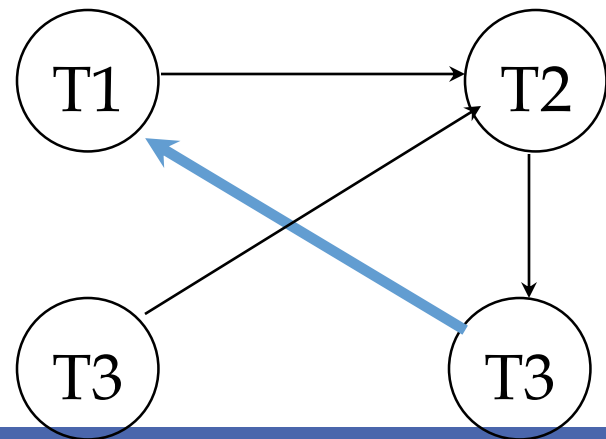
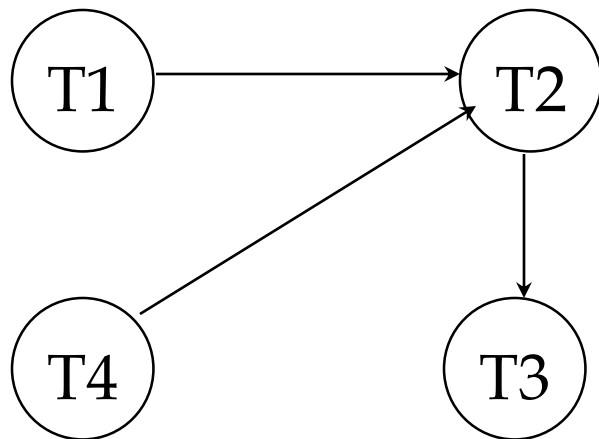
Example:

T1: S(A), R(A), S(B)

T2: X(B), W(B)

T3: S(C), R(C) X(C) X(A)

T4: X(B)

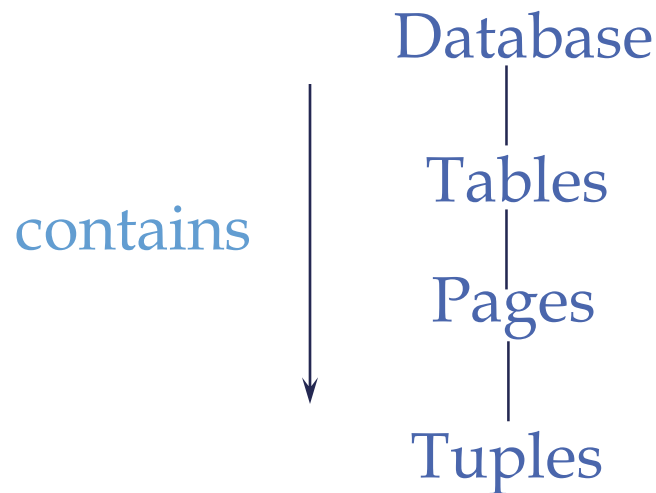


Multiple-Granularity Locks

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:

- 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 or IS lock on a node, must hold IS or 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.

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 [lock escalation](#) to decide which.

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”!

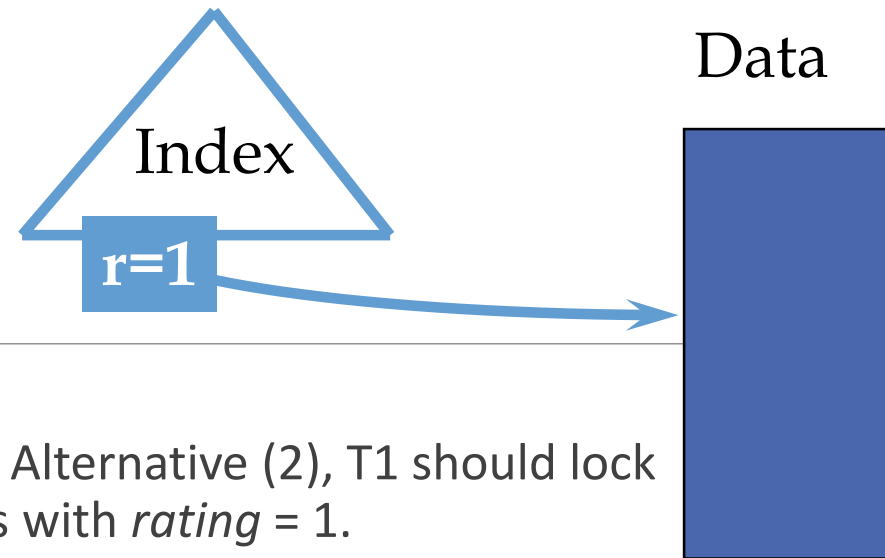
The Problem

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 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.

- What is the predicate in the sailor example?

In general, predicate locking has a lot of locking overhead.

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

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 safe:

- 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

ROOT



A

Do:

- 1) Search 38*
- 2) Insert 45*
- 3) Insert 25*



B



F



C

G

H

I

D

E



20*

22*

23*

24*

35*

36*

38*

41*

44*

A Better Tree Locking Algorithm (See Bayer-Schkolnick paper)

Search: As before.

Insert/Delete:

- Set locks as if for search, get to leaf, and set X lock on leaf.
- If leaf is not **safe**, release all locks, and restart Xact using previous Insert/Delete protocol.

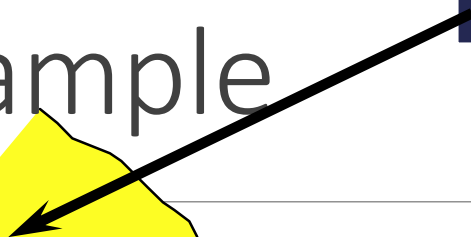
Gambles that only leaf node will be modified; if not, S locks set on the first pass to leaf are wasteful. In practice, better than previous alg.

ROOT



A

Example



B



F



C

G

H

I

D

E



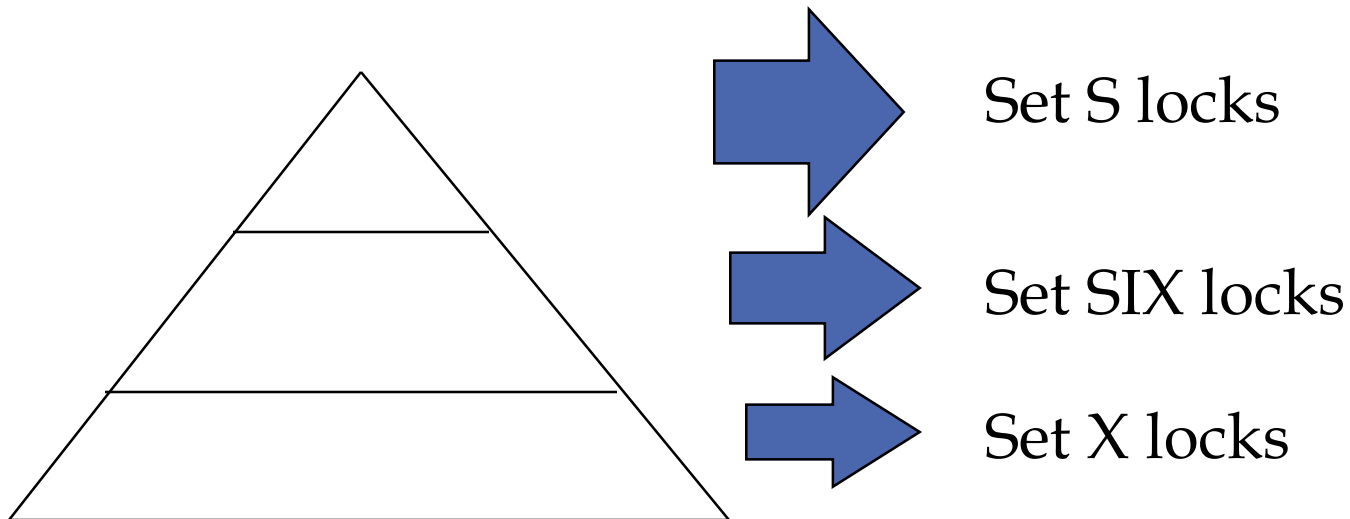
Do:

- 1) Delete 38*
- 2) Insert 25*
- 4) Insert 45*

Hybrid Algorithm

The likelihood that we really need an X lock decreases as we move up the tree.

Hybrid approach:



Optimistic CC (Kung-Robinson)

Locking is a conservative approach in which conflicts are prevented.

Disadvantages:

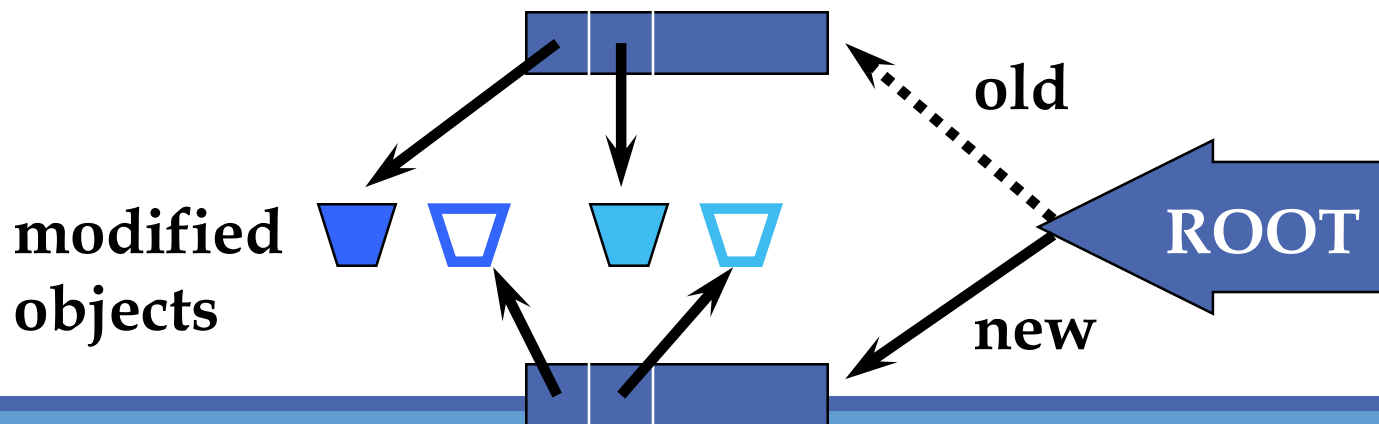
- Lock management overhead.
- Deadlock detection/resolution.
- Lock contention for heavily used objects.

If conflicts are rare, we might be able to gain concurrency by not locking, and instead checking for conflicts before Xacts commit.

Kung-Robinson Model

Xacts have three phases:

- **READ:** Xacts read from the database, but make changes to private copies of objects.
- **VALIDATE:** Check for conflicts.
- **WRITE:** Make local copies of changes public.



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.

Test 1

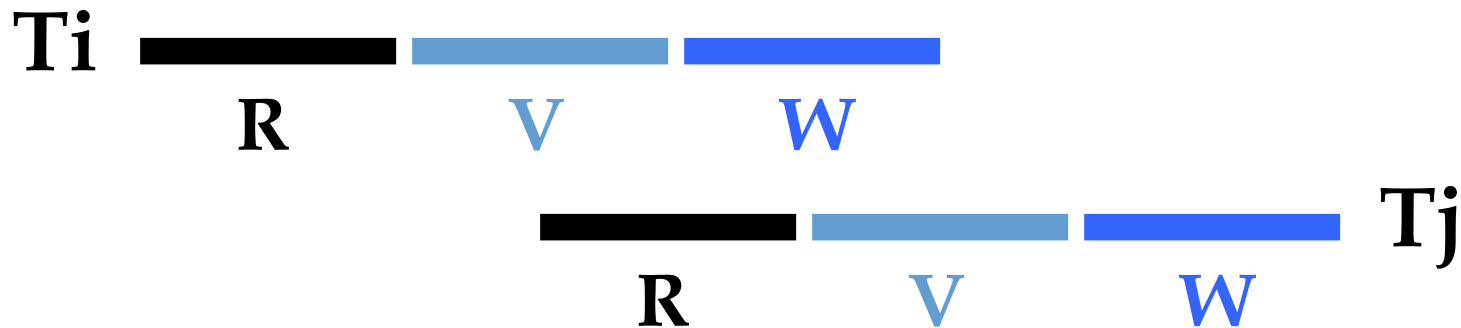
For all i and j such that $T_i < T_j$, check that T_i completes before T_j begins.



Test 2

For all i and j such that $T_i < T_j$, check that:

- T_i completes before T_j begins its Write phase +
- $\text{WriteSet}(T_i) \cap \text{ReadSet}(T_j)$ is empty.

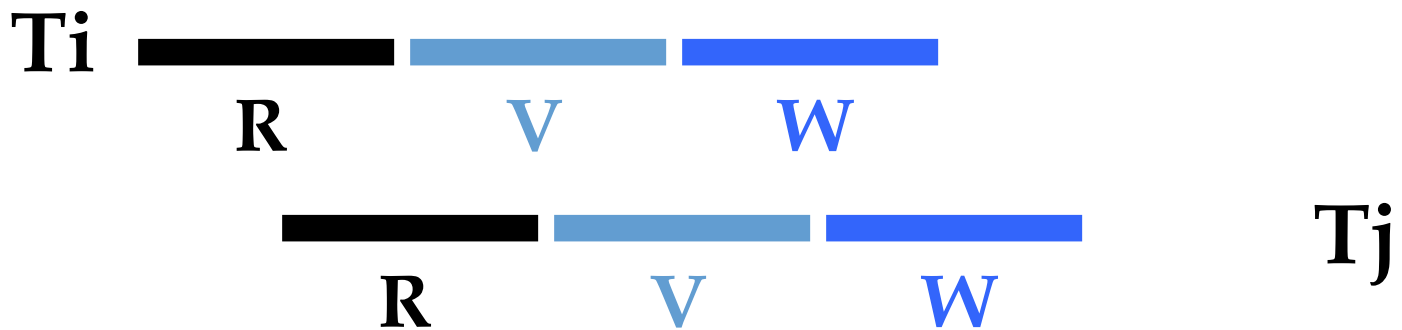


Does T_j read dirty data? Does T_i overwrite T_j 's writes?

Test 3

For all i and j such that $T_i < T_j$, check that:

- T_i completes Read phase before T_j does +
- $WriteSet(T_i) \cap ReadSet(T_j)$ is empty +
- $WriteSet(T_i) \cap WriteSet(T_j)$ is empty.



Does T_j read dirty data? Does T_i overwrite T_j 's writes?

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(T) intersect WriteSet(Ts) ≠ ∅  
    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 T_j and each Xact in T_s (in turn) being T_i .

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 a_i of Xact T_i conflicts with action a_j of Xact T_j , and $TS(T_i) < TS(T_j)$, then a_i must occur before a_j . 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 ddk 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. reader 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 R(A)	T2
W(A) Commit	W(A) Commit

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.

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.