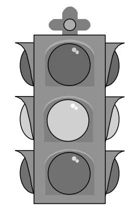
## Concurrency Control

15-415 Spring 2003, Lecture 22
R &G - Chapter 17



Smile, it is the key that fits the lock of everybody's heart.

Anthony J. D'Angelo,
The College Blue Book

## Review

- DBMSs support ACID Transaction semantics.
- Concurrency control and Crash Recovery are key compontents here.
- For Isolation property, a serial execution of transactions is safe but slow
  - Try to find schedules equivalent to serial execution
- One solution for "conflict serializable" schedules is Two Phase Locking (2PL)

## Conflict Serializable Schedules

- We need a formal notion of equivalence that can be implemented efficiently…
- Two operations conflict if they are by different transactions, they are on the same object, and at least one of them is a write.
- Two schedules are conflict equivalent iff:
  They involve the same actions of the same transactions, and
  every pair of conflicting actions is ordered the same way
- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule.
  - Note, some "serializable" schedules are NOT conflict serializable.
  - This is the price we pay for efficiency.

## Conflict Equivalence - Intuition

- If you can transform an interleaved schedule by swapping consecutive *non-conflicting* operations of different transactions into a serial schedule, then the original schedule is *conflict serializable.*
- *Example:*

R(A) W(A) R(B) W(B)
R(A) W(A) R(B) W(B)

≡

R(A) W(A) R(B) W(B)
R(A) W(A) R(B) W(B)

## Conflict Equivalence (Continued)

- Here's another example:

R(A) W(A)
R(A) W(A)

- Serializable or not????

NOT!

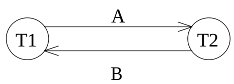
## Dependency Graph

- *Dependency graph*: One node per Xact; edge from *Ti* to *Tj* if an operation of Ti conflicts with an operation of Tj and Ti's operation appears earlier in the schedule than the conflicting operation of Tj.
- Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic

## 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.

## View Serializability – an Aside

- Alternative (weaker) notion of 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
- Basically, allows all conflict serializable schedules + "blind writes"

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

view ≡

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

## Notes on Serializability Definitions

- View Serializability allows (slightly) more schedules than Conflict Serializability does.
  - Problem is that it is difficult to implement efficiently.
- Neither definition allows all schedules that you would consider "serializable".
  - This is because they don't understand the meanings of the operations or the data.
- In practice, Conflict Serializability is what gets used, because it can be done efficiently.
  - In order to allow more concurrency, some special cases do get implemented, such as for travel reservations, etc.

## Two-Phase Locking (2PL)

Lock Compatibility Matrix

| | S | X |
|---|---|---|
| S | √ | – |
| X | – | – |

- 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.
  - Thus, there is a "growing phase" followed by a "shrinking phase".
- 2PL on its own is sufficient to guarantee conflict serializability, but, it is subject to Cascading Aborts.

## Strict 2PL

- Problem: Cascading Aborts
- Example: rollback of T1 requires rollback of T2!

| | |
|---|---|
| T1: | R(A), W(A), R(B), W(B), Abort |
| T2: | R(A), W(A) |

- To avoid Cascading aborts, use Strict 2PL
- Strict Two-phase Locking (Strict 2PL) Protocol:
  - Same as 2PL, except:
  - <u>All locks held by a transaction are released only when the transaction completes</u>

## Strict 2PL (continued)

<u>All locks held by a transaction are released only when the transaction completes</u>

- Strict 2PL allows only schedules whose precedence graph is acyclic, but it is actually stronger than needed for that purpose.
- In effect, "shrinking phase" is delayed until
  a) Transaction has committed (commit log record on disk), or
  b) Decision has been made to abort the xact (then locks can be released after rollback).

## Non-2PL, A= 1000, B=2000, Output =?

| | |
|---|---|
| Lock_X(A) | |
| Read(A) | Lock_S(A) |
| A: = A-50 | |
| Write(A) | |
| Unlock(A) | |
| | Read(A) |
| | Unlock(A) |
| | Lock_S(B) |
| Lock_X(B) | |
| | Read(B) |
| | Unlock(B) |
| | PRINT(A+B) |
| Read(B) | |
| B := B +50 | |
| Write(B) | |
| Unlock(B) | |

## 2PL, A= 1000, B=2000, Output =?

| | |
|---|---|
| Lock_X(A) | |
| Read(A) | Lock_S(A) |
| A: = A-50 | |
| Write(A) | |
| Lock_X(B) | |
| Unlock(A) | |
| | Read(A) |
| | Lock_S(B) |
| | |
| Read(B) | |
| B := B +50 | |
| Write(B) | |
| Unlock(B) | Unlock(A) |
| | Read(B) |
| | Unlock(B) |
| | PRINT(A+B) |

## Strict 2PL, A= 1000, B=2000, Output =?

| | |
|---|---|
| Lock_X(A) | |
| Read(A) | Lock_S(A) |
| A: = A-50 | |
| Write(A) | |
| Lock_X(B) | |
| Read(B) | |
| B := B +50 | |
| Write(B) | |
| Unlock(A) | |
| Unlock(B) | |
| | Read(A) |
| | Lock_S(B) |
| | Read(B) |
| | PRINT(A+B) |
| | Unlock(A) |
| | Unlock(B) |

## Lock Management

- Lock and unlock requests are handled by the Lock Manager.
- LM contains an entry for each currently held lock.
- Lock table entry:
  - Ptr. to list of transactions currently holding the lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- When lock request arrives see if anyone else holding a conflicting lock.
  - If not, create an entry and grant the lock.
  - Else, put the requestor on the wait queue
- Locking and unlocking have to be atomic operations
- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock
  - Can cause deadlock problems

## Example: Output = ?

| | |
|---|---|
| Lock_X(A) | |
| | Lock_S(B) |
| | Read(B) |
| | Lock_S(A) |
| Read(A) | |
| A: = A-50 | |
| Write(A) | |
| Lock_X(B) | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |

## Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection
- Many systems just punt and use Timeouts
  - What are the dangers with this approach?

## 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 gets its original timestamp
  - Why?

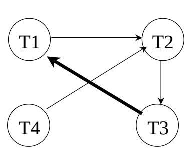
## Deadlock Detection

- 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)

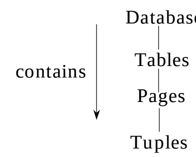
Example:

| | | | | | | |
|---|---|---|---|---|---|---|
| T1: | S(A), S(D), | | S(B) | | | |
| T2: | | X(B) | | | X(C) | |
| T3: | | | | S(D), S(C), | | X(A) |
| T4: | | | | | X(B) | |

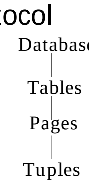



## Multiple-Granularity Locks

- Hard to decide what granularity to lock (tuples vs. pages vs. tables).
- Shouldn't have to make same decision for all transactions!
- 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:
- Still need S and X locks, but before locking an item, Xact must have proper intension locks on all its ancestors in the granularity hierarchy.

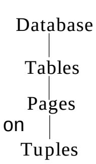
❖ IS – Intent to get S lock(s) at finer granularity.

❖ IX – Intent to get X lock(s) at finer granularity.

❖ SIX mode: Like S & IX at the same time. Why useful?

| | IS | IX | SIX | S | X |
|---|---|---|---|---|---|
| IS | √ | √ | √ | √ | - |
| IX | √ | √ | - | - | - |
| SIX | √ | - | - | - | - |
| 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.
  - What if Xact holds SIX on parent? S on parent?
- 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 – 2 level hierarchy

Tables
|
Tuples

- T1 scans R, and updates a few tuples:
  - T1 gets an SIX lock on R, then get X lock on tuples that are updated.
- 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.
  - Lock escalation dynamically asks for coarser-grained locks when too many low level locks acquired

| | IS | IX | SIX | S | X |
|---|---|---|---|---|---|
| IS | √ | √ | √ | √ | |
| IX | √ | √ | | | |
| SIX | √ | | | | |
| S | √ | | | √ | |
| X | | | | | |

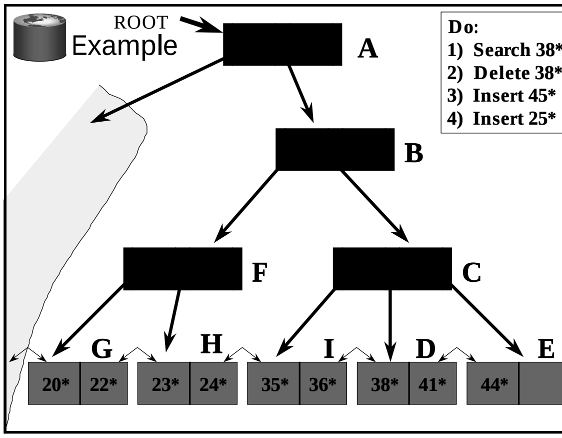
## A Simple Tree Locking Algorithm: "crabbing"

- 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 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.

## Locking in B+ Trees

- What about locking indexes --- why is it needed?
- Tree-based indexes present a potential concurrency bottleneck:
- If you ignore the tree structure & just lock pages while traversing the tree, following 2PL.
  - Root node (and many higher level nodes) become bottlenecks because every tree access begins at the root.
- How can we efficiently lock a particular leaf node?
  - Btw, don't confuse this with multiple granularity locking!

## Example




## Two Useful Observations

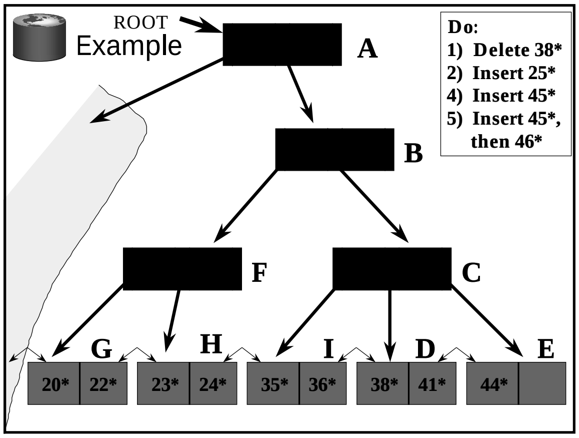
- 1) In a B+Tree, higher levels of the tree only direct searches for leaf pages.
- 2) For inserts, a node on a path from root to modified leaf must be locked (in X mode, of course), only if a split can propagate up to it from the modified leaf. (Similar point holds w.r.t. deletes.)

- We can exploit these observations to design efficient locking protocols that guarantee serializability *even though they violate 2PL.*

## A Better Tree Locking Algorithm (From 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.

## Example




## The Problem

- T1 and T3 implicitly assumed that they had locked the set of all sailor records satisfying a predicate.
  - Assumption only holds if no sailor records are added while they are executing!
  - Need some mechanism to enforce this assumption. (Index locking and predicate locking.)
- Examples show that conflict serializability on reads and writes of individual items guarantees serializability only if the set of objects is fixed!

## Dynamic Databases – The "Phantom" Problem

- If we relax the assumption that the DB is a fixed collection of objects, even Strict 2PL (on individual items) will not assure serializability:
- Consider T1 – "Find oldest sailor"
  - T1 locks all records, and finds oldest sailor (say, *age* = 71).
  - Next, T2 inserts a new sailor; *age* = 96 and commits.
  - T1 (within the same transaction) checks for the oldest sailor again and finds sailor aged 96!!
- The sailor with age 96 is a "phantom tuple" from T1's point of view --- first it's not there then it is.
- No serial execution where T1's result could happen!

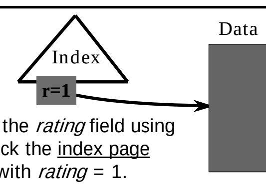
## Predicate Locking

- 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.

## The "Phantom" Problem – example 2

- Consider T3 – "Find oldest sailor for each rating"
  - T3 locks all pages containing sailor records with *rating* = 1, and finds oldest sailor (say, *age* = 71).
  - Next, T4 inserts a new sailor; *rating* = 1, *age* = 96.
  - T4 also deletes oldest sailor with rating = 2 (and, say, *age* = 80), and commits.
  - T3 now locks all pages containing sailor records with *rating* = 2, and finds oldest (say, *age* = 63).
- T3 saw only part of T4's effects!
- No serial execution where T3's result could happen!

## Index Locking




- If there is a dense index on the *rating* field using Alternative (2), T3 should lock the index page containing the data entries with *rating* = 1.
  - If there are no records with *rating* = 1, T3 must lock the index page where such a data entry *would* be, if it existed!
- If there is no suitable index, T3 must lock all pages, and lock the file/table to prevent new pages from being added, to ensure that no records with *rating* = 1 are added or deleted.

## Transaction Support in SQL-92

- SERIALIZABLE – No phantoms, all reads repeatable, no “dirty” (uncommited) reads.
- REPEATABLE READS – phantoms may happen.
- READ COMMITTED – phantoms and unrepeatable reads may happen
- READ UNCOMMITTED – all of them may happen.

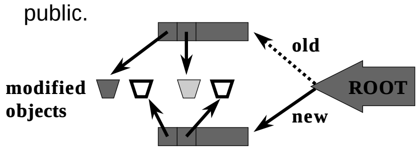
## Optimistic CC (Kung-Robinson)

Locking is a conservative approach in which conflicts are prevented. Disadvantages:

- Lock management overhead.
- Deadlock detection/resolution.
- Lock contention for heavily used objects.
- Locking is “pessimistic” because it assumes that conflicts will happen.
- If conflicts are rare, we might get better performance by not locking, and instead checking for conflicts at 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.

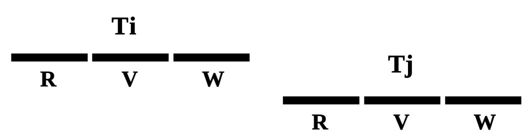



## Validation

- 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.

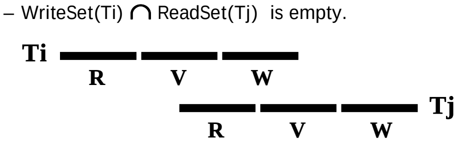
## Test 1

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




## Test 2

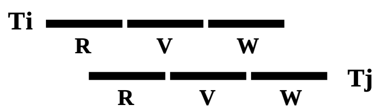
- For all i and j such that Ti < Tj, check that:
  - Ti completes before Tj begins its Write phase **AND**
  - WriteSet(Ti) $\cap$ ReadSet(Tj) is empty.




Does Tj read dirty data? Does Ti overwrite Tj’s writes?

## Test 3

- For all i and j such that Ti < Tj, check that:
  - Ti completes Read phase before Tj does **AND**
  - WriteSet(Ti) ∩ ReadSet(Tj) is empty **AND**
  - WriteSet(Ti) ∩ WriteSet(Tj) is empty.




Does Tj read dirty data? Does Ti overwrite Tj's writes?

## 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.

## Applying Tests 1 & 2: Serial Validation

- To validate Xact T:

```
valid = true;
// S = set of Xacts that committed after Begin(T)
//    (above defn implements Test 1)
//The following is done in critical section
< foreach  Ts in S do {
  if ReadSet(T) intersects WriteSet(Ts)
      then valid = false;
  }
  if valid then { install updates; // Write phase
                Commit T } >
          else Restart T
```

start of critical section (points to "<")

end of critical section (points to ">")

## `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).

## 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**!
  - Nothing else goes on concurrently.
  - So, no need to check for Test 3 --- can't happen.
  - If Write phase is long, major drawback.
- Optimization for Read-only Xacts:
  - Don't need critical section (because there is no Write phase).

## Other Techniques

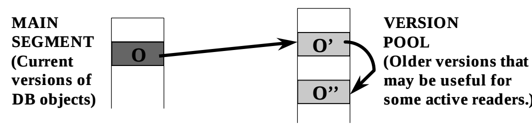
- **Timestamp CC:** Give each object a read-timestamp (RTS) and a write-timestamp (WTS), give each Xact a timestamp (TS) when it begins:
  - If action ai of Xact Ti conflicts with action aj of Xact Tj, and TS(Ti) < TS(Tj), then ai must occur before aj. Otherwise, restart violating Xact.
- **Multiversion CC:** Let writers make a "new" copy while readers use an appropriate "old" copy.
  - Advantage is that readers don't need to get locks
  - Oracle uses a simple form of this.

## 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.

## Multiversion Timestamp CC

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




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

## 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:**

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

- Else, allow T to write O.

## Multiversion CC (Contd.)

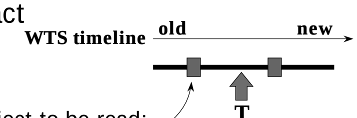
- Each version of an object has its writer's TS as its WTS, and the TS of the Xact that most recently read this version as its RTS.
- Versions are chained backward; we can discard versions that are "too old to be of interest".
- Each Xact is classified as Reader or Writer.
  - Writer *may* write some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

## Timestamp CC and Recoverability

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

- ❖ Unfortunately, unrecoverable schedules are allowed:
- 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.

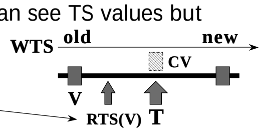
## Reader Xact




- 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.




## Summary (Contd.)

- 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.
- There are many other approaches to CC that we don't cover here. These include:
  - timestamp-based approaches
  - multiple-version approaches
  - semantic approaches

## Summary

- Correctness criterion for isolation is "serializability".
  - In practice, we use "conflict serializability", which is somewhat more restrictive but easy to enforce.
- Two Phase Locking, and Strict 2PL: Locks directly implement the notions of conflict.
  - The lock manager keeps track of the locks issued. Deadlocks can either be prevented or detected.
- Must be careful if objects can be added to or removed from the database ("phantom problem").
- Index locking common, affects performance significantly.
  - Needed when accessing records via index.
  - Needed for locking logical sets of records (index locking/predicate locking).

## 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.

## 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!
- Tree-structured indexes:
  - Straightforward use of 2PL very inefficient.
  - Idea is to use 2PL on data to ensure serializability and use other protocols on tree to ensure structural integrity.
  - Bayer-Schkolnick illustrates potential for improvement.