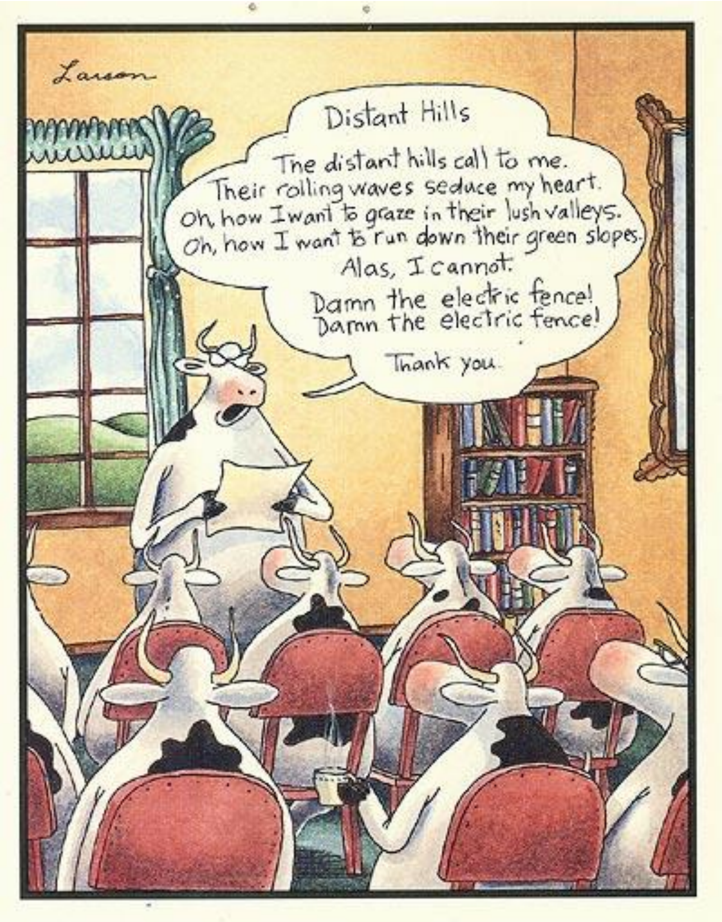
Concurrency Control

Chapter 17



Cow poetry

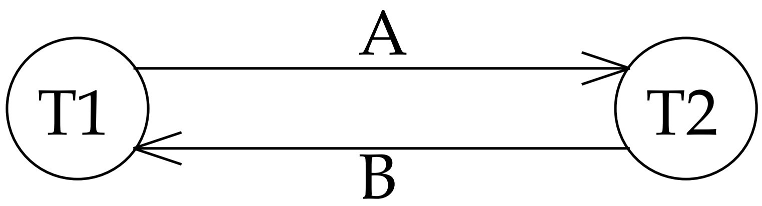
Conflict Serializable Schedules

- Recall *conflicts (WR, RW, WW)* were the cause of sequential inconsistency
- Two schedules are conflict equivalent if:
  - Involve the same actions over the same transactions
  - Every pair of conflicting actions is ordered the same way
- A schedule is conflict serializable if it is *conflict equivalent* to some serializable schedule

Example 1

- A non-serializable schedule that is also not *conflict serializable*:

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



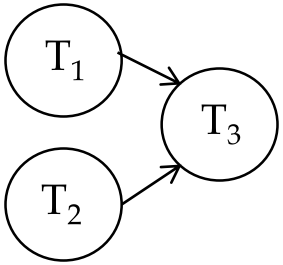
Precedence graph

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

Example 2

- A serializable schedule that is not conflict serializable:

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



- Serializable because it is equiv to
  T1, T2, T3, or T2, T1, T3
- Not *conflict serializable*, because the ordering:
  $R_1(A), W_2(A), W_1(A), W_3(A)$
  is not consistent with any ordering, but *conflict equivalent*
- Importance of this distinction is that it can be proven that *Strict 2PL* permits only conflict serializable schedules

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 (a DAG)

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 release its locks once it has performed its desired operation (R or W). A transaction cannot 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.
- Note: locks can be released before Xact completes (commit/abort), thus relaxing Strict 2PL. 2PL starts with a "growing" phase, where locks are requested followed by a "shrinking" phase, where locks are released

Lock Management

- Lock and unlock requests are handled by the database's *lock manager*
- Lock table entry (per table, record, or index):
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- Locking and unlocking must be atomic
- *Lock upgrades*: transaction that holds a shared lock can be upgraded to hold an exclusive lock

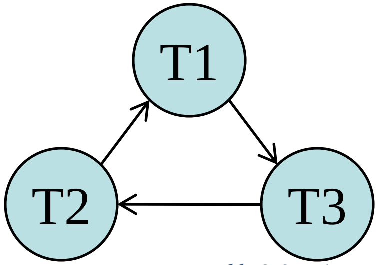
Deadlocks

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

Deadlock Detection

- Create a waits-for graph:
  - Nodes are transactions
  - Edge from Ti to Tj indicates Ti is waiting for Tj to release a lock
- DBMS periodically checks for cycles in the waits-for graph
- ex: T1: A = f(B), T2: B = g(C) , T3: C = h(A), arriving T1,T3,T2

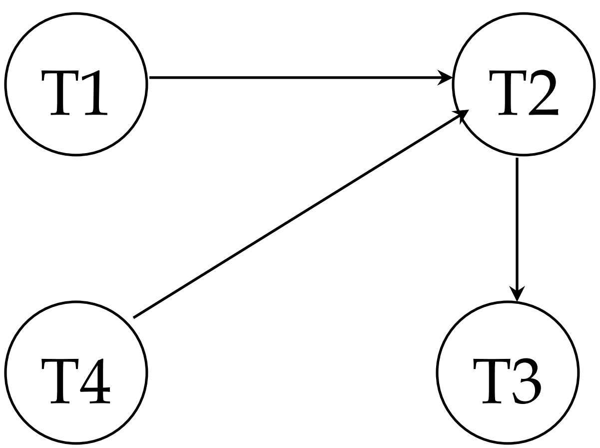
T1:	S(B),R(B),		X(A),…		
T2:				S(C),R(C),X(B),…	
T3:		S(A),R(A),			X(C),…

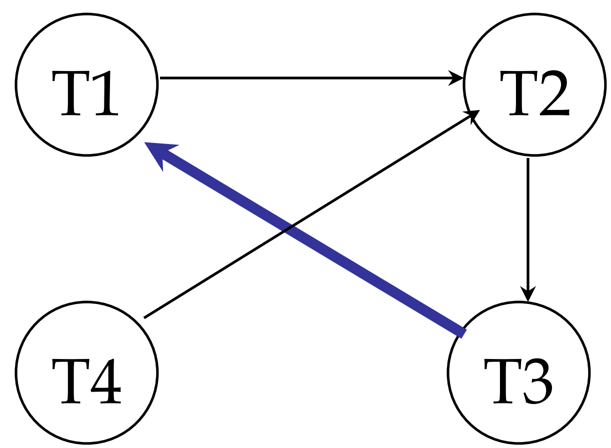


Deadlock Detection (Continued)

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)…





Deadlock Prevention

- When there is high contention for locks, detection and aborting can hurt performance
- Assign priorities (eg. based on a Xact's duration using 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 abort Ti (wait only if higher priority)
  - *Wound-wait*: If Ti has higher priority, abort Tj; otherwise Ti waits (preempt lower priorities)
- When Ti re-starts, it retains its original timestamp, thus moves up the priority list

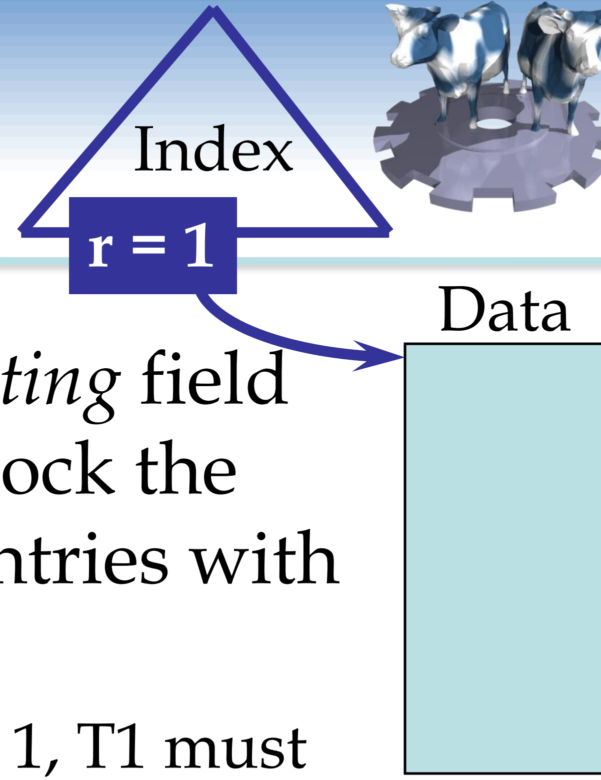
Dynamic Databases

- With fine-grain locks, even Strict 2PL will not assure serializability:
  - T1 locks all pages that currently contain sailors records with *rating* = 1, and finds oldest sailor (say, *age* = 71).
  - Next, T2 inserts a new sailor; *rating* = 1, *age* = 96. (added to a page that previously had no sailor with rating 1, such pages are not locked)
  - T2 also deletes oldest sailor with rating = 2 (and, say, *age* = 80), and commits. (these aren't locked, and T2 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"!
- Locking pages based on a selection is called a "predicate" lock

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.
  - What is the predicate in the sailor example?
- In general, predicate locking has a lot of overhead, and is seldom implemented.

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.
- In practice, better techniques now known; do record-level, rather than page-level locking.