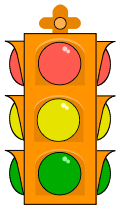
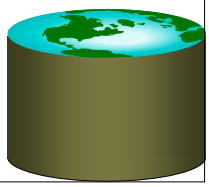
Concurrency Control Part 2

R&G - Chapter 17



The sequel was far better than the original!

-- Nobody



Outline

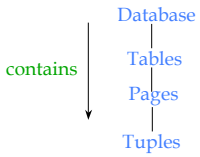
- Last time:
  - Theory: conflict serializability, view serializability
  - Two-phase locking (2PL)
  - Strict 2PL
  - Dealing with deadlocks (prevention, detection)
- Today: "advanced" locking issues…
  - Locking granularity
  - Tree locking protocols
  - Phantoms & predicate locking

Locking Granularity

- Hard to decide what granularity to lock (tuples vs. pages vs. tables).
- ***why?***

Multiple-Granularity Locks

- 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?

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 S on parent? SIX 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.

Lock Compatibility Matrix

Database
Tables
Pages
Tuples

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

- 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.

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					

Outline

- Today: "advanced" locking issues…
  - Locking granularity
  - Tree locking protocols
  - Phantoms & predicate locking

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.
- Special protocol for tree locking?
  - BTW, don't confuse this with multiple granularity locking!

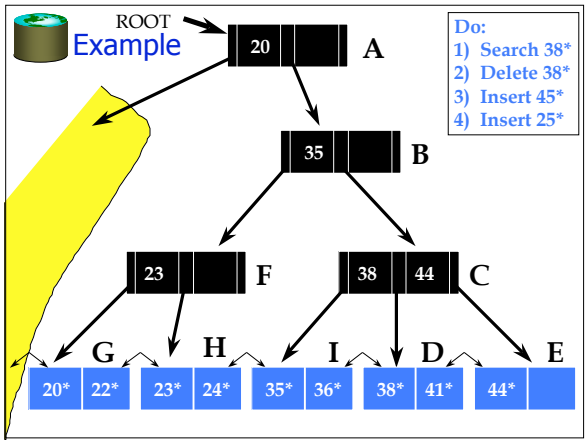
Two Useful Observations

- 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 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.
  - Insertions: Node is not full.
  - Deletions: Node is not half-empty.

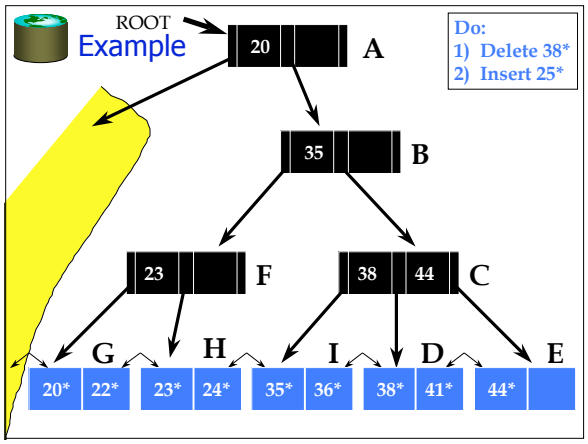
Example



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, usually better than previous alg.

Example



Outline

- Today: "advanced" locking issues…
  - Locking granularity
  - Tree locking protocols
  - Phantoms & predicate locking

Dynamic Databases – The "Phantom" Problem

- Relax assumption that DB = **fixed** collection of objects
- Even Strict 2PL (on individual items) will not ensure 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!

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!

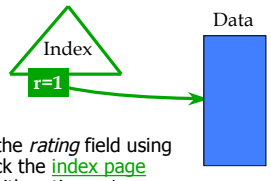
The Problem

- 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!

Predicate Locking

- Grant lock on all records that satisfy some logical predicate, e.g. *age > 2*salary*.
- In general, predicate locking has a lot of locking overhead.
- 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?

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 obtain:
  1. A lock on every page in the table file
     - To prevent a record's rating from being changed to 1

  AND
  2. The lock for the file itself
     - To prevent records with *rating* = 1 from being 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.

Summary

- Multiple granularity locking – flexibility for each xact to choose locking granularity independently
- Tree-structured indexes:
  - Straightforward use of 2PL very inefficient.
  - Instead, design specialized locking protocols for trees
    - Other work in this (important) area, e.g., Lehman-Yao
- If database objects can be added/removed, need to guard against Phantom Problem
  - Must lock logical sets of records.
  - Efficient solution: index locking.