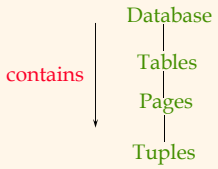
Transaction Management

Index Locking

Instructor: Sharma Chakravarthy
sharma@cse.uta.edu
The University of Texas @ Arlington

Multiple-Granularity Locks

- Hard to decide what granularity to lock (tuples vs. pages vs. tables vs. database).
- 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. (top-down)
- 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.
  - 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.

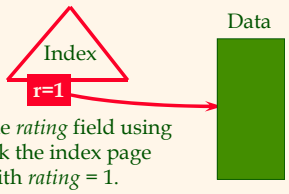
Protocol is correct in that it is equivalent to directly setting locks at the leaf levels of the hierarchy.

Examples

- 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	√				

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 (equivalent to locking the end of file or eof), 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. *title = supervisor*.
- 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 (or index)

- Why do we have to worry about locking when using B+ trees?
- We lock the data pages (or records) anyway!
- If we are only reading, should we worry about locking the index
- If we are writing (insert/update/delete), should we worry about locking the index

Locking in B+ Trees (or index)

- 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

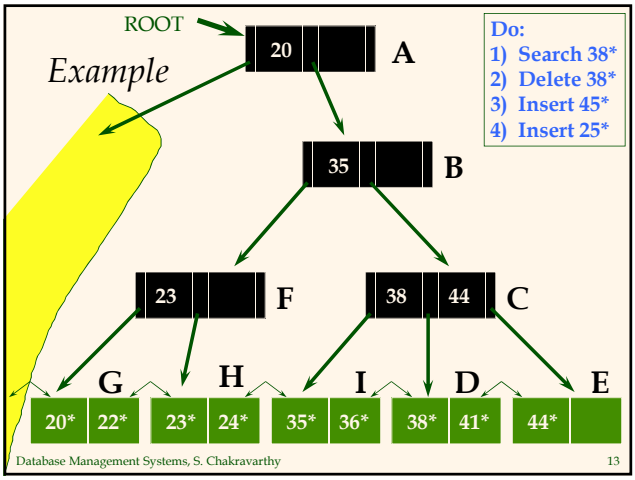
- Higher levels of the tree only direct searches for leaf pages.
- For inserts, a node on a path from root to modified leaf must be locked (in X mode, of course), only if a split can propagate up to it from the modified leaf. (Similar point holds w.r.t. deletes.)
- We can exploit these observations to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

Tree Locking

- If we are only doing read operations on the database, should we lock the index tree nodes and how?
- If we are doing updates to the database (insert/delete/modify), should we lock the index nodes and how?
- Where does the 2PL come into picture? We have not discussed it for indexes; we discussed it only for data

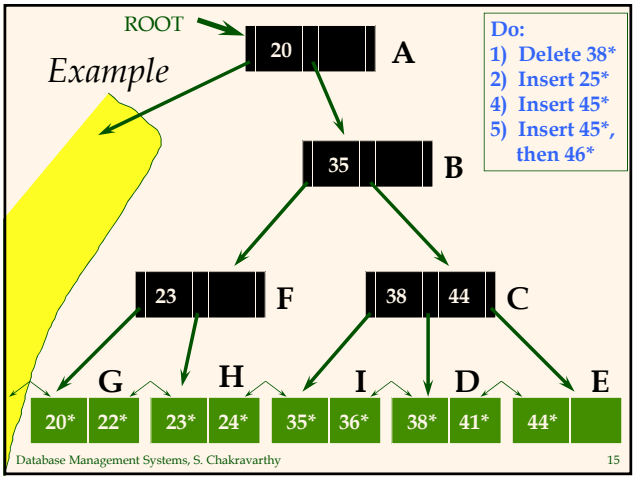
A Simple Tree Locking Algorithm

- Search: Start at root and go down; repeatedly, S lock child then unlock parent.
  - Why?
- 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.
  - Why will this work?
- Safe node: Node such that changes will not propagate up beyond this node.
  - Inserts: Node is not full.
  - Deletes: Node is not half-empty.
- Why does this violate 2PL?



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.

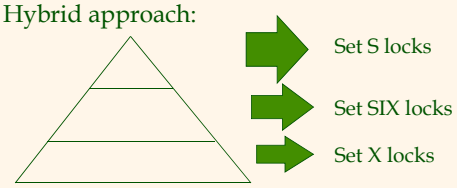


Even Better Algorithm

- Search: As before.
- Insert/Delete:
  - Use original Insert/Delete protocol, but set IX locks instead of X locks at all nodes.
  - Once leaf is locked, convert all IX locks to X locks top-down: i.e., starting from node nearest to root. (Top-down reduces chances of deadlock.)
- Contrast use of IX locks here with their use in multiple-granularity locking.
  - Not containment
  - Need for dealing with a path whose nodes may get updated

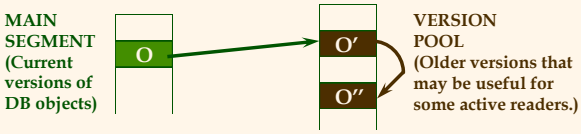
Hybrid Algorithm

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



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.

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.

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

CC comparison

- Locks
  - used by lock-based approaches
  - space in the lock table is proportional to the number of database elements locked
- Timestamps
  - Used by timestamp-based algos
  - space is needed for read- and write-times to be recorded
  - Can use something similar to a lock table to record timestamps of only those database elements that have been accessed recently
- Validation
  - used by optimistic CC.
  - Space is needed for timestamps and read/write sets for each currently active Tx.

Summary

- There are several lock-based concurrency control schemes (Strict 2PL, 2PL, conservative 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.

Thank You !