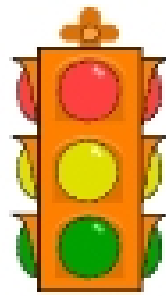


Concurrency Control Part 2

R&G - Chapter 17



The sequel was far better than the original!

-- Nobody



Outline

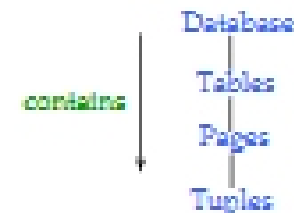
- Last time:
 - Theory: conflict serializability, view serializability
 - Two-phase locking (2PL)
 - Strict 2PL
 - Dealing with deadlocks (prevention, detection)
- Today: "advanced" locking issues...
 - Locking granularity
 - Optimistic Concurrency Control

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 intention 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

	IS	IX	SI	S	X
IS	✓	✓	✓	✓	-
IX	✓	✓	-	-	-
SI	✓	-	-	-	-
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.

Database
|
Tables
|
Pages
|
Tuples



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	SI	S	X
IS	✓	✓	✓	✓	-
IX	✓	✓	-	-	-
SI	✓	-	-	-	-
S	✓	-	-	✓	-
X	-	-	-	-	-



Just So You're Aware: Indexes

- ZPL on B+-tree pages is a rotten idea.
 - Why?
- Instead, do short locks (latches) in a clever way
 - Idea: Upper levels of B+-tree just need to direct traffic correctly. Don't need to be serializably handled!
 - Different tricks to exploit this
- Note: this is pretty complicated!



Just So You're Aware: Phantoms

- Suppose you query for sailors with rating between 10 and 20, using a B+-tree
 - Tuple-level locks in the Heap File
- I insert a Sailor with rating 12
- You do your query again
 - Yikes! A phantom!
 - Problem: Serializability assumed a static DB!
- What we want: lock the *logical* range 10-20
 - Imagine that lock table!
- What is done: set locks in indexes cleverly



Roadmap

- So far:
 - Correctness criterion: serializability
 - Lock-based CC to enforce serializability
 - Strict 2PL
 - Deadlocks
 - Hierarchical Locking
 - Tree latching
 - Phantoms
- Next:
 - Alternative CC mechanism: Optimistic



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.
- What if conflicts are rare?
 - We might get better performance by not locking, and instead checking for conflicts at commit time.



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

- **Idea:** test conditions that are sufficient to ensure that no conflict occurred.
- Each Xact assigned a numeric id.
 - Just use a **timestamp**.
 - Assigned at end of READ phase.
- **ReadSet(T_i):** Set of objects read by Xact T_i.
- **WriteSet(T_i):** Set of objects modified by T_i.



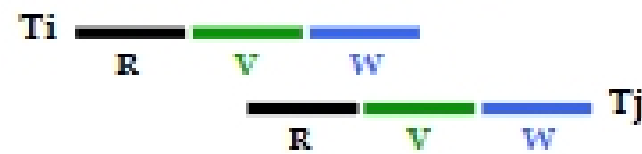
Test 1

- 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 AND
 - WriteSet(T_i) ∩ 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 AND
 - WriteSet(T_i) ∩ ReadSet(T_j) is empty AND
 - WriteSet(T_i) ∩ 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)
// (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

end of critical section