Transaction Management: Concurrency Control, part 2

CS634
Class 16

Slides based on "Database Management Systems" 3rd ed, Ramakrishnan and Gehrke

Locking for B+ Trees

- Naïve solution
  - Ignore tree structure, just lock its pages following 2PL
- Very poor performance!
  - Root node (and many higher level nodes) become bottlenecks
  - Every tree access begins at the root!
- Not needed anyway!
  - Only row data needs 2PL (contents of tree)
  - Tree structure also needs protection from concurrent access
  - But only like other shared data of the server program
  - Note this modern view is not covered in book
  - See Graefe, A Survey of B-tree locking techniques (2010)
  - B-tree locking is a huge challenge!

Locking vs. Latching

- To protect shared data in memory, multithreaded programs use mutex (semaphores) AKA latches, sometimes "locks" (confusing!)
  - API: enter_section/leave_section, or lock/unlock
  - Every Java object contains a mutex, for convenience of Java programming: underlies synchronized methods
  - Database people call mutexes and related mechanisms "latches"
  - Need background in multi-threaded programming to understand this topic fully
- The tree *structure* needs mutex/latch protection
- Example: split node. No row data is changed, just the details in pages in the buffer pool. No i/o is needed (can't hold a latch across disk i/o without ruining performance.)
- Latches can be provided by the same lock manager as does 2PL locking, and can have share and exclusive types like locks.
- In these slides, will use "lock" in quotes to mean non-2PL lock/latch to make it look somewhat like the book's discussion…

Locking for B+ Trees (contd.)

- **Searches**
  - Higher levels only direct searches for leaf pages
- **Insertions**
  - Node on a path from root to modified leaf must be "locked" in X mode only if a split can propagate up to it
  - Similar point holds for deletions
- There are efficient locking protocols that keep the B-tree healthy under concurrent access, and support 2PL on rows

A Simple Tree Locking Algorithm:

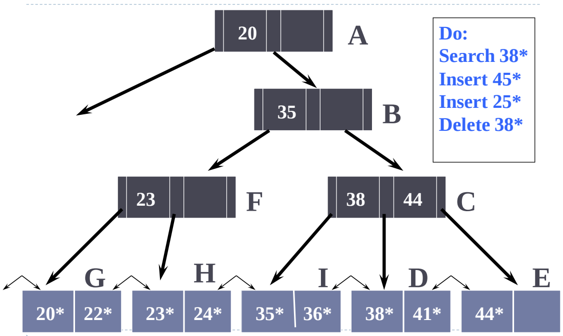
("lock" here is really a latch on tree structure)

- **Search**
  - Start at root and descend: "crabbing down the tree"
  - repeatedly, get S "lock" for child then "unlock" parent, end up with S "lock" on leaf page
  - Get 2PL S lock on row, provide row pointer to caller
  - Later, caller is done with reading row, arranges release of S "lock"
- **Insert/Delete**
  - Start at root and descend, crabbing, obtaining X "locks" as needed
  - Once child is "locked", check if it is **safe**
  - If child is safe, release "lock" on parent, leaving X "lock" on child
  - Get 2PL X lock on place for new row/old row, insert/delete row, release "lock"
- **Safe node**: not about to split or coalesce
  - Inserts: Node is not full
  - Deletes: Node is not half-empty
- When control gets back to QP, transaction only has 2PL locks on rows. Only 2PL locks are long-term across multiple DB actions.

Difference from text

- The algorithm actions described in the text are valid, for example, crabbing down the tree, worrying about full nodes, etc.
- What's different is that the locks for index nodes are shorter lived than described in the text: only 2PL locks on rows are kept until end of transaction, not any locks on index nodes.
- Note that text uses locks and releases them before commit, a sign that they are not actually Strict 2PL locks.
- Note the admission on pg. 564 that the text's coverage on this topic is "not state of the art". Graefe's paper is.

An Example from pg. 563



Insert 45 case (corrected 4/12)

Crab down tree getting X "locks" (really latches)
"Xlock" A
"Xlock" B
B is safe, so "unXlock" A
"Xlock" C
C is unsafe, so can't "unXlock" B now
"Xlock" E (its page of rows is in buffer,)
E is safe, so "unXlock" C, and B too
Xlock E (real 2PL page lock)
"UnXLock" E
Return to QP with 2PL Xlock on page, and pointer to it in pinned buffer.
QP will unpin when done with edits to page

A Variation on Algorithms

- **Search**
  - As before
- **Insert/Delete**
  - Set "locks" as if for search, get to leaf, and set 2PL X lock on leaf
  - If leaf is not safe, release all "locks", and restart using previous Insert/Delete protocol
- This is what happens if the search down the tree happens on a page that is not in buffer—don't want to hold a latch across a disk i/o (takes too long)

Multiple-Granularity Locks

- Hard to decide what granularity to lock
  - tuples vs. pages vs. files
- Shouldn't have to decide!
- Data containers are nested:

contains ↓

Database
Files
Pages
Tuples

New Lock Modes, Protocol

- Allow transactions to lock at each level, but with a special protocol using new **intention locks**

- Before locking an item, must set intention locks on ancestors
- 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 transaction starts from the root of the hierarchy
- To get S or IS lock on a node, must hold IS or IX on parent node
- 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

Snapshot Isolation (SI)

- Multiversion Concurrency Control Mechanism (MVCC)
- This means the database holds more than one value for a data item at the same time
- Used in PostgreSQL (open source), Oracle, others
- Readers never conflict with writers unlike traditional DBMS (e.g., IBM DB2)! Read-only transactions run fast.
- Does not guarantee "real" serializability
- But: ANSI "serializability" fulfilled, i.e., avoids anomalies in the ANSI table
- Found in use at Microsoft in 1993, published as example of MVCC

Snapshot Isolation - Basic Idea:

- Every transaction reads from its own snapshot (copy) of the database (will be created when the transaction starts, or reconstructed from the undo log).
- Writes are collected into a writeset (WS), not visible to concurrent transactions.
- Two transactions are considered to be concurrent if one starts (takes a snapshot) while the other is in progress.

First Committer Wins Rule of SI

- At the commit time of a transaction its WS is compared to those of concurrent committed transactions.
- If there is no conflict (overlapping), then the WS can be applied to stable storage and is visible to transactions that begin afterwards.
- However, if there is a conflict with the WS of a concurrent, already committed transaction, then the transaction must be aborted.
- That's the "First Committer Wins Rule"
- Actually Oracle uses first updater wins, basically same idea, but doesn't require separate WS

Write Skew Anomaly of SI

- In MVCC, data items need subscripts to say which version is being considered
  - Zero version: original database value
  - T1 writes new value of X, X_1
  - T2 writes new value of Y, Y_2
- Write skew anomaly schedule:

$R_1(X_0)\ R_2(X_0)\ R_1(Y_0)\ R_2(Y_0)\ W_1(X_1)\ C_1\ W_2(Y_2)\ C_2$

- Writesets WS(T1) = {X}, WS(T2) = {Y}, do not overlap, so both commit.
- So what's wrong—where's the anomaly?

Write Skew Anomaly of SI

$R_1(X_0)\ R_2(X_0)\ R_1(Y_0)\ R_2(Y_0)\ W_1(X_1)\ C_1\ W_2(Y_2)\ C_2$

- Scenario:
  - X = husband's balance, orig 100,
  - Y = wife's balance, orig 100.
  - Bank allows withdrawals up to combined balance
  - Rule: X + Y >= 0
  - Both withdraw 150, thinking OK, end up with -50 and -50.
- Easy to make this happen in Oracle at "Serializable" isolation.
- See conflicts, cycle in PG, can't happen with full 2PL
- Can happen with RC/locking

How can an Oracle app handle this?

- If X+Y >= 0 is needed as a constraint, it can be "materialized" as sum in another column value.
- Old program: R(X)R(X-spouse)W(X)C
- New program: R(X)R(X-spouse) W(sum) W(X)C
- So schedule will have W(sum) in both transactions, and sum will be in both Writesets, so second committer aborts.
- Or, after the W(X), run a query for the sum and abort if it's negative.

Oracle, Postgres: new failure to handle

- Recall deadlock-abort handling: retry the aborted transaction
- With SI, get "can't serialize access“
  - ORA-08177: can't serialize access for this transaction
  - Means another transaction won for a contended write
- App handles this error like deadlock-abort: just retry transaction, up to a few times
- This only happens when you set serializable isolation level

Other anomalies under SI

- Oldest sailors example
  - Both concurrent transactions see original sailor data in snapshots, plus own updates
  - Updates are on different rows, so both commit
  - Neither sees the other's update
  - So not serializable: one should see one update, other should see two updates.
- Task Registry example:
  - Both concurrent transactions see original state with 6 hours available for Joe
  - Both insert new task for Joe
  - Inserts involve different rows, so both commit

Fixing the task registry

- Following the idea of the simple write skew, we can materialize the constraint "workhours <= 8"
- Add a workhours column to worker table
- Old program:
- if sum(hours-for-x)+newhours<=8
- insert new task
- New program:
- if workhours-for-x + newhours <=8
- { update worker set workhours = workhours + newhours…
- insert new task
- }

Fixing the Oldest sailor example

- If the oldest sailor is important to the app, materialize it!

Create table oldestsailor (rating int primary key, sid int)

Oracle Read Committed Isolation

- READ COMMITTED is the default isolation level for both Oracle and PostgreSQL.
- A new snapshot is taken for every issued SQL statement (every statement sees the latest committed values).
- If a transaction T2 running in READ COMMITTED mode tries to update a row which was already updated by a concurrent transaction T1, then T2 gets blocked until T1 has either committed or aborted
- Nearly same as 2PL/RC, though all reads occur effectively at the same time for the statement.

Transaction Management: Crash Recovery

CS634

Slides based on "Database Management Systems" 3rd ed, Ramakrishnan and Gehrke

ACID Properties

Transaction Management must fulfill four requirements:

1. Atomicity: either all actions within a transaction are carried out, or none is
   - Only actions of committed transactions must be visible
2. Consistency: concurrent execution must leave DBMS in consistent state
3. Isolation: each transaction is protected from effects of other concurrent transactions
   - Net effect is that of **some sequential execution**
4. Durability: once a transaction commits, DBMS changes will persist
   - Conversely, if a transaction aborts/is aborted, there are no effects

Recovery Manager

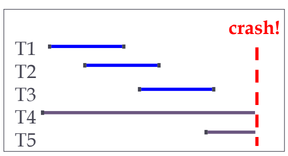
- Crash recovery
  - Ensure that atomicity is preserved if the system crashes while one or more transactions are still incomplete
  - Main idea is to keep a log of operations; every action is logged before its page updates reach disk (Write-Ahead Log or WAL)
- The **Recovery Manager** guarantees Atomicity & Durability

Motivation

- Atomicity:
  - Transactions may abort – must rollback their actions
- Durability:
  - What if DBMS stops running – e.g., power failure?

Desired Behavior after system restarts:

- T1, T2 & T3 should be durable
- T4 & T5 should be aborted (effects not seen)



Assumptions

- Concurrency control is in effect
  - Strict 2PL
- Updates are happening "in place"
  - Data overwritten on (deleted from) the disk
- A simple scheme is needed
  - A protocol that is too complex is difficult to implement
  - Performance is also an important issue