6.5830 / 6.5831 – Lab 3 Bootcamp

Transactions & Concurrency Control

Code Overview

- Part 1: The Lock Manager
- Part 2: Transaction Context
- Part 3: Execution Engine Integration
- Part 4: Transaction Manager

Part 1: Lock Manager

- Who can access what resource, and when?
  - Isolation
  - Mutual exclusion
  - Deadlock handling
- Pessimistic locking
  - Locking data upon access to prevent concurrent modification.
  - Conflicts prevented not detected later.

Lock Granularities

- When a txn wants to acquire a “lock”, the DBMS can decide the granularity (i.e., scope) of that lock.
  - Attribute? Tuple? Page? Table?
  - **DBLockTag**
    - **NewTableLockTag**
    - **NewTupleLockTag**

- The DBMS should ideally obtain fewest number of locks that a txn needs.

Intention Locks

- An intention lock allows a higher-level node to be locked in shared or exclusive mode without having to check all descendent nodes.

- If a node is locked in an intention mode, then some txn is doing explicit locking at a lower level in the tree.

Intention Locks

- Intention-Shared (IS)
  - Indicates explicit locking at lower level with S locks.
  - Intent to get S lock(s) at finer granularity.
- Intention-Exclusive (IX)
  - Indicates explicit locking at lower level with X locks.
  - Intent to get X lock(s) at finer granularity.
- Shared+Intention-Exclusive (SIX)
  - The subtree rooted by that node is locked explicitly in S mode and explicit locking is being done at a lower level with X locks.

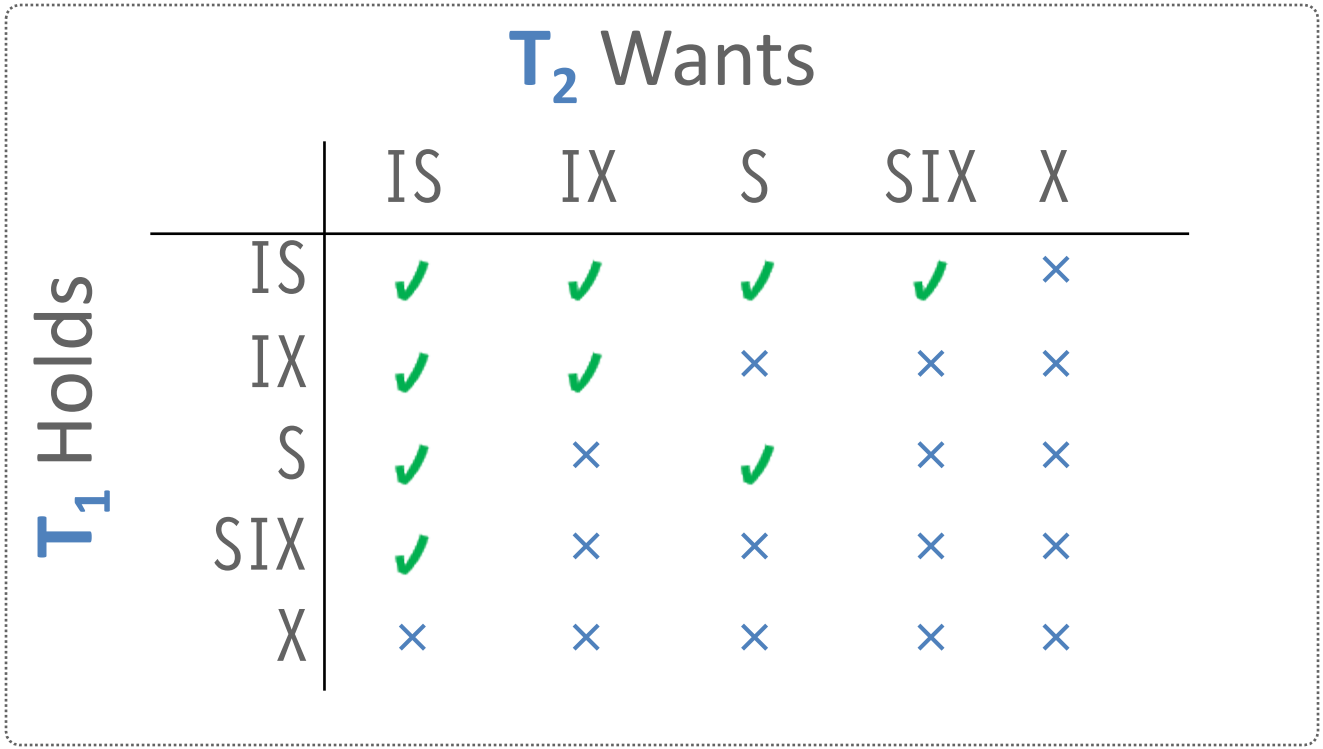
Intention Locks

- **DBLockMode**

```
// DBLockMode represents the type of access a transaction is requesting.
// GoDB supports a standard Multi-Granularity Locking hierarchy.
type DBLockMode int

const (
    // LockModeS (Shared) allows reading a resource. Multiple transactio
    LockModeS DBLockMode = iota
    // LockModeX (Exclusive) allows modification. It is incompatible wit
    LockModeX
    // LockModeIS (Intent Shared) indicates the intention to read resour
    LockModeIS
    // LockModeIX (Intent Exclusive) indicates the intention to modify r
    LockModeIX
    // LockModeSIX (Shared Intent Exclusive) allows reading the resource
    LockModeSIX
)
```

Compatibility Matrix



Compatibility vs Coverage

- Compatibility is between different transactions
  - Determines whether a **new lock request conflicts with locks held by OTHER transactions**
  - T1 holds S, T2 wants S, can I grant it?
- Coverage is between same transactions
  - Determines whether a transaction's **existing lock is already strong enough**
  - T1 holds X, T1 requests S, do I already have permission?

Compatibility vs Coverage

- Compatibility is between different transactions
  - Determines whether a **new lock request conflicts with locks held by OTHER transactions**
  - T1 holds S, T2 wants S, can I grant it?
- Coverage is between same transactions
  - Determines whether a transaction's **existing lock is already strong enough**
  - T1 holds X, T1 requests S, do I already have permission?

Locking Protocol

- Each txn obtains the appropriate lock at highest level of the database hierarchy.
  - Acquire locks top to bottom
- To get S or IS lock on a node, the txn must hold at least IS on parent node.
- To get X, IX, or SIX on a node, must hold at least IX on parent node.

Data Structures & Strategy

- LockManager
  - Represents all locks
- You need to create data structures to represent one singular lock
- Lock(tid, tag, mode)
- Unlock(tid, tag)

Lock Requests and Entries

- What can we track that would help us?
  - Transaction id: who is requesting a resource
  - Tag: resource identifier
  - Mode: locking mode
  - Holders: who is holding the resource
  - Waiters: who is waiting for the resource

Lock Pipeline

- Find/Create your logical ‘lock’
- Check if Transaction already holds the lock
  - If yes, check if it is strong enough, upgrade
- Check conflicts with other transactions
  - If compatible, grant lock
  - If conflict, block until you can proceed

Unlock Pipeline

- Find the lock
- ‘Remove’ transaction holding the lock
- Check who is waiting for the lock
- Grant the lock to new transaction

Concurrency Helpers

- Sync.Cond
  - Synchronization primitive used to suspend (wait) one or more goroutins until a specific condition becomes true, and then notify (wake) them to continue execution
  - Wait()
  - Signal()
  - Broadcast()

Concurrency Helpers

- Sync.pool
  - Used for recycling objects
  - If there is an object in the pool, grab it from there, instead of making a new one from scratch
- Sync.mutex
- Xsync.MapOf

Deadlocks

- Deadlock detection:
  - Waits-for graphs to keep track of what locks each txn is waiting to acquire
  - Periodically check for cycles in wait-for graphs and decide how to break it

Deadlock Prevention

- Assign each txn a timestamp when they start and use them to determine priorities.
  - For example, Older Timestamp = Higher Priority (e.g., T1 > T2)
- Wait-Die ("Old Waits for Young")
  - If requesting txn has higher priority than holding txn, then requesting txn waits for holding txn.
  - Otherwise requesting txn aborts.
- Wound-Wait ("Young Waits for Old")
  - If requesting txn has higher priority than holding txn, then holding txn aborts and releases lock.
  - Otherwise requesting txn waits.

Common Mistake: Reentrancy

- Transaction requests same lock again
- Without check → self-deadlock
- Make sure your ‘upgrade’ logic is correct.
- Who should get priority when acquiring locks?

Common Mistake: Fairness

- You need to be fair to transactions waiting.
- One way is to somehow keep track of who requested what and when. This could also help with deadlocks.

Common Mistake: Wrong Wait-Die

- Letting younger txn wait → deadlocks
- Killing wrong txn → starvation

Common Mistake: Mutex Bugs

- Accessing shared state without lock
- Leads to race conditions

Big Picture

- Lock(): decide grant vs wait
- Unlock(): decide who wakes up
- Correctness = ordering

Part 2: Transaction Context

- Everything this txn knows and did
- State of ONE transaction

```
// TransactionContext holds the runtime state of a single transaction.
type TransactionContext struct {
    field id common.TransactionID // size=8, offset=0
    id         common.TransactionID
    lm         *LockManager
    logRecords *logRecordBuffer
    heldLocks  map[DBLockTag]DBLockMode

    // These holds in-memory actions (e.g., indexing rollbacks) deferred until transactions end. This is used because in
    // GoDB, indexes are memory-only for simplicity, and do not need to participate in the WAL-driven recovery process.
    // In a real DBMS, indexes are often also on disk and must be protected by a WAL; in fact, indexing recovery is
    // often much more complicated than just the table storage itself, due to multi-page structural modifications
    // (e.g., B-tree merge/split). YOU SHOULD NOT NEED TO MANIPULATE THIS FIELD
    abortActions, commitActions []IndexTask
}
```

Transaction context

- Track locks held
- Maintain undo history
- Provide reentrancy
- Be recyclable

LogRecordBuffer

- Each txn maintains its own local bytes slice, that acts as a private Undo Log
- Methods to implement
  - Allocate()
  - Len()
  - Get()
  - Pop()
  - Reset()

Transaction context

- Transaction context methods
  - AcquireLock()
  - HeldLock()
- Connect to LockManager you already implemented

Common Mistakes

- Should be pretty straightforward to implement
- Make sure you understand reentrancy and recycle behavior.

Part 3: Integration

- Buffer Pool FlushAllPages() from Lab 1
  - WAL in wal.go
  - LogManager interface is already defined for you
  - You must not flush dirty page until its log records are flushed
  - Page.LSN

Part 3: Integration

- TableHeap
  - TransactionContext is passed to Insert/Delete/ Update methods, you need to make sure that you are acquiring appropriate locks
- Pay attention to forUpdate argument in ReadTuple
  - If true, should acquire exclusive lock instead of shared

Part 3: Integration

- Index Updates
  - In memory indexes are not separately locked – only tuples carry locks
- **Inserts are applied immediately.** When a transaction inserts a row, the index entry is added right away. This is required for own-write visibility — a transaction must be able to find its own inserts via index scans.
- **Deletes are deferred to commit time.** When a transaction deletes an index entry, the underlying tuple is changed immediately (under the X-lock), but the index entry is only removed when the transaction *commits*.

Part 3: Integration

- Call things in the right order and handle stale reads.
- abortActions and commitActions
- IndexOpUndoInsert - commit
- IndexopDelete - abort

Part 4: Transaction Manager

- Strong Strict 2 Phase Locking
  - Before every operation acquire a lock.
  - Accumulate all locks
  - Release locks only when commit or abort
- Transaction cannot be considered committed until its log record is flushed to disk
- Review Lecture 11 if you are having problems