Milestone Three

Nick Abcarius
Andrew Do
Travis Garcia
Nicole Pavlovich
Steven Tan

UC Davis, ECS 165A
Prof Mohammad Sadoghi

Milestone Goals

- Implement a lock manager to protect records during transactions and latches to protect shared data structures
- Implement concurrency through multithreaded transactions
- Be able to roll back aborted transactions to ensure data integrity

Locks & Latches

Locks

Lock Policy

Two-Phase Locking (2PL) with no wait: if a transaction can't obtain a lock, it immediately aborts

Locking occurs at the record level

Implementation

The lock manager is implemented at a global level

- Contains a dictionary mapping record keys to lock objects
- Transactions have an ID which determines if multiple writes or reads can happen on the same record by a single transaction

```
class RecordLock():
    def __init__(self):
        self.sLocks = 0
        self.xLocks =  0
        self.isShrinking = False
        self.inUseBy = []

class LockManager():
    def __init__(self):
        self.latch = threading.Lock()
        self.KeytoLocks = {}
        self.transactionID = -1
    def getTransactionID(self)
    def obtainSLock(self, Key, transactionID)
    def obtainXLock(self, Key, transactionID)
    def giveUpSLock(self, Key, transactionID)
    def giveUpXLock(self, Key, transactionID)
```

Locks & Latches

Rules for Locking

	SHARED	XCLUSIVE
SHARED	allowed	same transaction no multiple transactions
XCLUSIVE	same transaction no multiple transactions	not allowed

Latching

- Shared data structures are latched so that data integrity is ensured with concurrent access
- Used to prevent race conditions from non-atomic operations such as accessing the bufferpool or index updates
- Implemented using the Lock object from threading module (`self.latch = threading.Lock()`)

Threading

Threading

Implementing threading must be done carefully without breaking the promises of ACID:

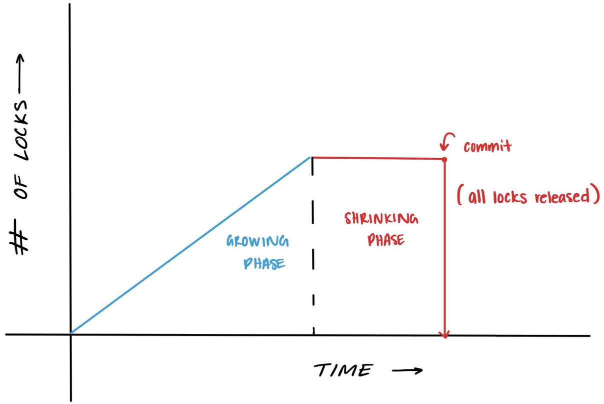
Atomicity: A transaction fails or finishes, but never partially

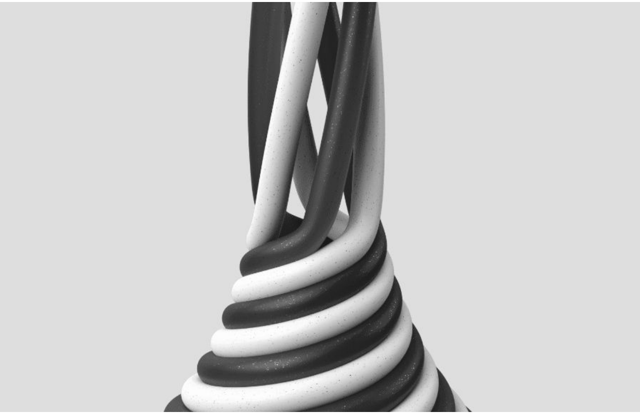
Consistency: Only valid data is written to the database

Integrity: Concurrent transactions execute in an order than can be sequentialized

Durability: Changes are saved in non-volatile memory

Threads are represented by the `transaction_worker` class. We instantiated 8 threads for testing





Commits

Commits

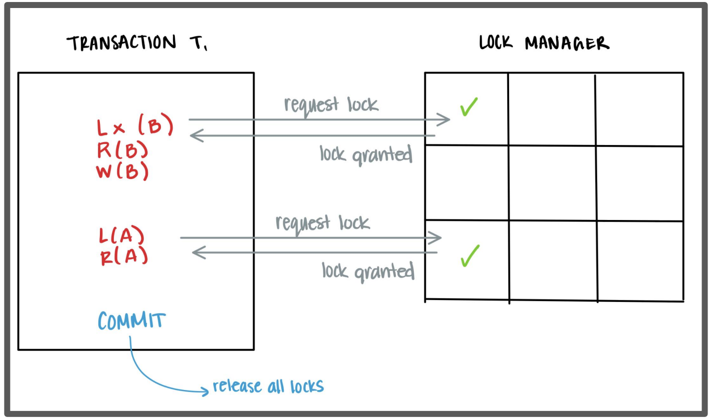
Once a transaction successfully completes all of its queries, the changes are committed to the database

Step 1: Commit Records

➔ Acquire a latch, then update the key mapping in our table to point to the committed base RID so that the record is now visible
➔ Update the base indirection value so the tail record is now visible

Step 2: Release all locks

➔ For each query in the transaction, depending on its type, X or S locks are released and the latch is released
➔ Committed transactions then return `True`



Aborts

Aborts

In order to maintain atomicity, a transaction that fails to completely execute must be aborted

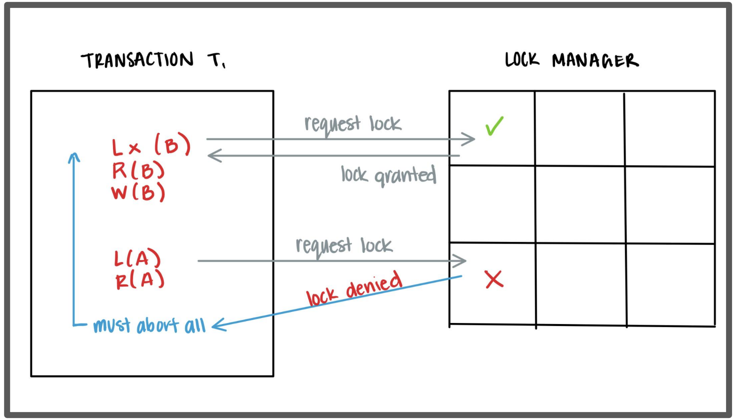
Step 1: Rollback Changes

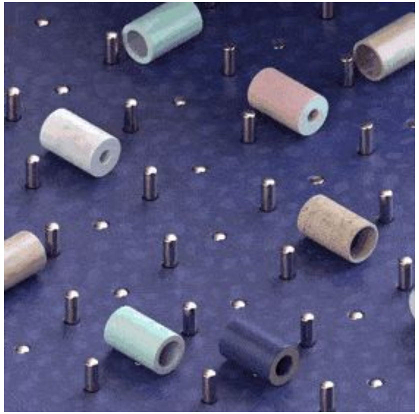
➔ Delete any inserted tail or base records

Step 2: Release all locks

➔ For each query in the transaction, depending on its type, X or S locks are released

➔ Aborted transactions then return `False`





Final Thoughts

- Durability could be increased as we currently can roll back aborts but have no formal log for crash protection
- Implementation could switch from 2PL to 2VCC to avoid aborts
- Could do further optimization and testing to improve overall performance

Milestone Three

Nick Abcarius
Andrew Do
Travis Garcia
Nicole Pavlovich
Steven Tan

UC Davis, ECS 165A
Prof Mohammad Sadoghi