Milestone 3

Group JeLLY-B

Jessica LaCourse
Lingxiao Meng
Lisa Malins
Yinuo Zhang
Benjamin Rausch

Record Locks

Strict two-phase lock implementation

- **Shared locks** for reads
- **Exclusive locks** for writes
- Write locks only released at end of transaction

Two **custom lock classes**

- **XSLock** provides S and X locks
- **IntentXSLock** supports multiple granularity locking with IS, S, IX, and X modes

Record Locks - 2 phases

Expanding Phase:

- What's the greatest lock we need?
- Collect all locks before starting transaction
- Read locks upgradable to write lock for the same query

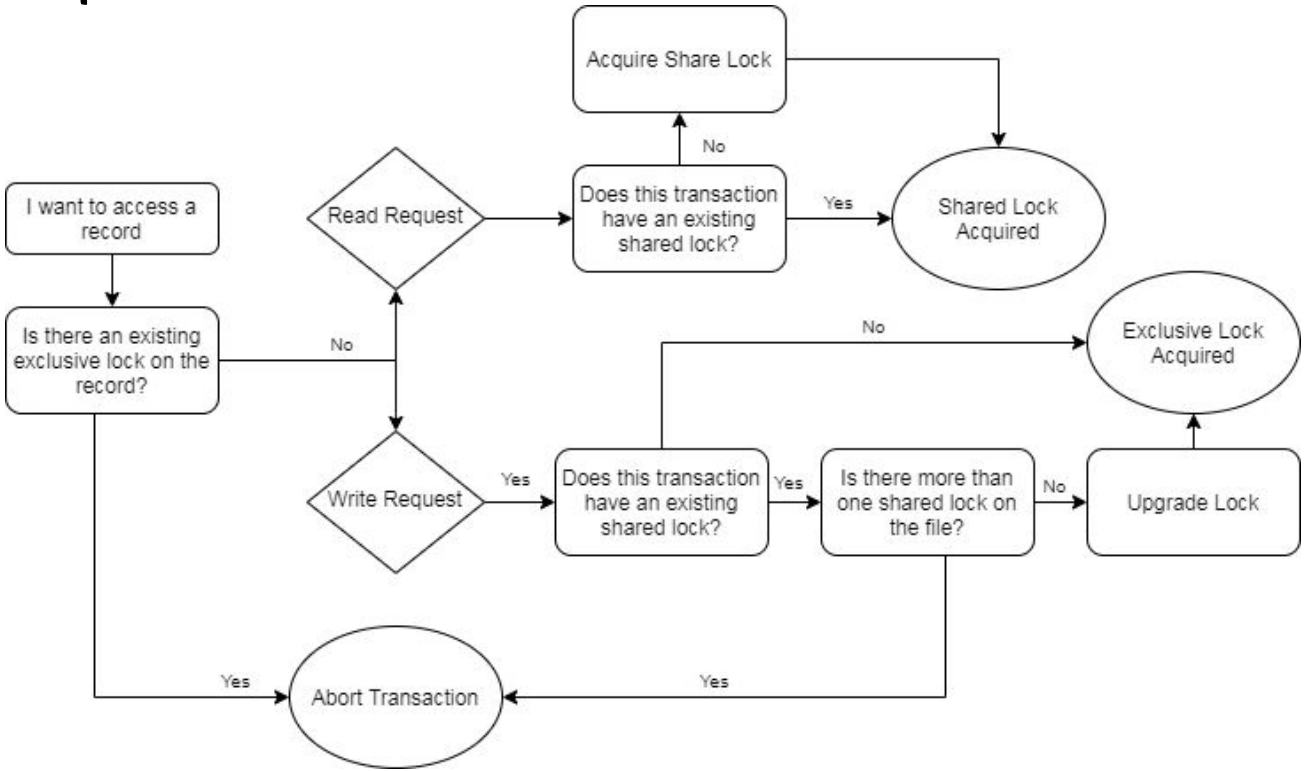
Shrinking Phase:

- Release all locks once the transaction has completed (either commit or abort)

Transaction IDs

- Each transaction assigned a **transaction ID**
  - ID is random hexadecimal number
- Table holds list of IDs for all active transactions
- IDs used to verify if transaction is already holding a lock
  - Avoid acquiring same lock twice
  - Upgrade shared to exclusive lock

Lock Acquisition



Transactions

Transactions are handled through **pre-check functions**.

For **selections** or **reads**:

1. Fails to add a shared lock to a record
    a. If there is an exclusive lock
    b. If the last request from the transaction was a read request.
2. Successfully adds a shared lock when there are no exclusive locks

Transactions

Transactions are handled through **pre-check functions**.

For **updates** or **writes**:

1. Fails to add an exclusive lock to a record
    a. If there is an existing exclusive lock
    b. If there are multiple read requests on the record
2. Successfully adds an exclusive lock otherwise

Commit and Abort

All locks can be acquired? ✔ Ready to commit

Cannot obtain one or more locks? ✘ Abort transaction

- Successfully **avoid rollbacks** by ensuring all locks collected before a transaction is ever committed
- Released all the locks on that record
- Empty the pointers to former completed queries within that transaction

Making Components Thread-Safe

Page directory

- Uses our **XSLock** class
- `get_record_location()` requires S lock
- `recreate_page_directory()` requires X lock
- Both functions “spin” until they acquire the lock

Indices

- Whole ‘Indices’ object is protected by an **IntentXSLock**
- Each column’s index is protected by an **XSLock**
- Throws exception if lock cannot be obtained

Making Components Thread-Safe

Bufferpool

- Uses threading.**RLock** class
- Fixed issue from Milestone 2 when main thread and merge thread tried to evict same page from bufferpool

RIDAllocator

- Uses threading.**RLock** class
- Locking prevents same RIDs being allocated to more than one page
- This lock eliminates the need to lock pages

Improved merging

Milestone 2 code would **spawn merge threads within update()** function

- Check: Are there page ranges eligible for merging?
- Check: Is there already a merge thread running?

Now, we have a **single thread** constantly running a **merge daemon function**

- Thread starts when table is created
- Loops continuously, checking if there are page ranges to merge
- Automatically killed when program exits

Next steps

- Lock acquisition and decision to commit/abort works
  - But there are bugs applying operations to database
  - Now working on adjusting select() and update() to work with the new lock implementation
- Performance testing
  - Will be included in code submission