Concurrency Control

Lock Types

- Shared (**SLock**): For Reads
- Exclusive (**XLock**): For Writes, but can Read too
Two Phase Locking (2PL)

- **Growing Phase**: Transactions only request locks, cannot release. It is up to the lock-manager to grant or deny access to a lock.

- **Shrinking Phase**: Transactions only release locks, cannot request new locks once the growing phase is over.
Two Phase Locking (2PL) Algorithm

- When a transaction Ti issues a read(Q) operation, the system issues a SLock(Q) instruction followed by the read(Q) instruction.
- When Ti issues a write(Q) operation, the system checks to see whether Ti already holds a shared lock on Q. If it does, then the system issues an upgrade(Q) instruction, followed by the write(Q) instruction. Otherwise, the system issues a XLock(Q) instruction, followed by the write(Q) instruction.
- All locks obtained by a transaction are unlocked after that transaction commits or aborts.
Example 1

<table>
<thead>
<tr>
<th>Tx1</th>
<th>Tx2</th>
</tr>
</thead>
<tbody>
<tr>
<td>XLock(1)</td>
<td></td>
</tr>
<tr>
<td>R(1)</td>
<td></td>
</tr>
<tr>
<td>W(1)</td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td></td>
</tr>
</tbody>
</table>

Lock Manager
Access granted (Tx1→ 1)
Access Denied
Released (Tx1→ 1)
Access granted (Tx2→ 1)
Released (Tx2→ 1)
Concurrency Control in SimpleDB

- A simple two phase locking
- Block level locking
- Offers simple deadlock detection
Two-phase Locking (2PL) SimpleDB

```java
package simpledb.tx.concurrency;

import java.util.*;

// The concurrency manager for the transaction.
public class ConcurrencyMgr {

    /**
     * The global lock table. This variable is static because
     * all transactions share the same table.
     */
    private static LockTable locktbl = new LockTable();
    private Map<BlockId,String> locks = new HashMap<BlockId,String>();

    // Keeps a global lock table which is shared among all transactions
    // Keeps a local list of locks that is private to this transaction only
```
Two-phase Locking (2PL) SimpleDB

```java
/**
 * Obtain an SLock on the block, if necessary.
 * The method will ask the lock table for an SLock
 * if the transaction currently has no locks on that block.
 * @param blk a reference to the disk block
 */
public void sLock(BlockId blk) {
    if (locks.get(blk) == null) {
        locktbl.sLock(blk);
        locks.put(blk, "S");
    }
}

/**
 * Obtain an XLock on the block, if necessary.
 * If the transaction does not have an XLock on that block,
 * then the method first gets an SLock on that block
 * (if necessary), and then upgrades it to an XLock.
 * @param blk a reference to the disk block
 */
public void xLock(BlockId blk) {
    if (!hasXLock(blk)) {
        sLock(blk);
        locktbl.xLock(blk);
        locks.put(blk, "X");
    }
}

/**
 * Release all locks by asking the lock table to
 * unlock each one.
 */
public void release() {
    for (BlockId blk : locks.keySet())
        locktbl.unlock(blk);
    locks.clear();
}
```

Notice how SLock, XLock follows the algorithm stated earlier.

This function is called when a transaction commits and releases all locks.
Block Level Locking (SLock)

/**
 * Grant an SLock on the specified block.
 * If an XLock exists when the method is called,
 * then the calling thread will be placed on a wait list
 * until the lock is released.
 * If the thread remains on the wait list for a certain
 * amount of time (currently 10 seconds),
 * then an exception is thrown.
 * @param blk a reference to the disk block
 */

public synchronized void sLock(BlockId blk) {
    try {
        long timestamp = System.currentTimeMillis();
        while (hasXlock(blk) && !waitingTooLong(timestamp))
            wait(MAX_TIME);
        if (hasXlock(blk))
            throw new LockAbortException();
        int val = getLockVal(blk);  // will not be negative
        locks.put(blk, val+1);
    }
    catch(InterruptedException e) {
        throw new LockAbortException();
    }
}
Block Level Locking (XLock)

```java
/**
 * Grant an XLock on the specified block.
 * If a lock of any type exists when the method is called,
 * then the calling thread will be placed on a wait list
 * until the locks are released.
 * If the thread remains on the wait list for a certain
 * amount of time (currently 10 seconds),
 * then an exception is thrown.
 * @param blk a reference to the disk block
 */
synchronized void xLock(BlockId blk) {
    try {
        long timestamp = System.currentTimeMillis();
        while (hasOtherSLocks(blk) && !waitingTooLong(timestamp))
            wait(MAX_TIME);
        if (hasOtherSLocks(blk))
            throw new LockAbortException();
        locks.put(blk, -1);
    } catch (InterruptedException e) {
        throw new LockAbortException();
    }
}
```

Deadlock is detected based on timeout
Concurrent Test

- Tx A: request xlock 1
- Tx B: request xlock 2
- Tx A: receive xlock 1
- Tx B: receive xlock 2
- Tx C: request xlock 1
- Tx B: request xlock 1
- Tx A: request xlock 2
- Tx B: receive xlock 1
  transaction 2 committed
- Tx A: receive xlock 2
- Tx B: commit
  transaction 1 committed
- Tx A: commit
- Tx C: receive xlock 1
- Tx C: request xlock 2
- Tx C: receive xlock 2
  transaction 3 committed
- Tx C: commit
Rooms for improvement?

- Better deadlock detection and handling?
- Cascading Rollback handling?
- Page level or tuple level locking?
Reference

- https://www.db-book.com/db7/