Multicore Programming Project 4

-Simple & Fast Transaction Locking System–

11:59pm 20 DEC 2015

1. Objective

This project is to design and implement a simple transaction locking system, which should probably be developed based on the simple MVCC system developed in Project-2. The system you have to develop is to extend the transaction concepts to cover database locking in order to avoid numerous anomalies that may occur if your database doesn't have an appropriate transaction locking system.

2. Background

A transaction locking system is intended to arbitrate concurrent accesses to data items while it ensures all data integrity constraints. There is an important rule, called ***strict 2PL*** (two phase locking), that makes systems work properly. The name "two phase" comes from the way a transaction acquires/releases a database lock (logical lock protecting a database record, table, etc.). During ***the lock growing phase***, a transaction can only acquire database locks while it accesses database records (or items) in the table. After a transaction finishes reading/writing records, then ***the lock shrinking phase*** starts, which enforces a strict rule such that a transaction can ***NEVER*** acquire any lock once it released any of database locks it is holding. The time when a transaction releases a database lock is when a transaction commits.

This is the extremely short summary on transaction locking systems, and this is the one you have to design and implement.

3. Specifications

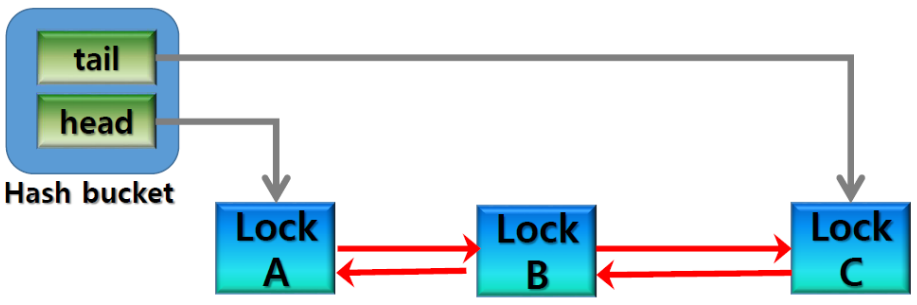
The system you have to develop is quite different from what you have built in Project-2, in a way that now each thread needs to have a logical transaction structure as shown below:

```
struct trx_t {
        unsigned long trx_id;    // Current transaction id assigned from the global counter
        vector<lock_t*> trx_locks; // A list of database locks this trx is holding
        enum transaction_state trx_state; // RUNNING, WAITING, IDLE
         mutex_t      trx_mutex: /* mutex protecting trx struct. This is needed to sleep/wakeup
                                    atomically, not to have lost-wake up problem */
         lock_t* wait_lock; /* This is the lock object that this transaction is waiting. This field
        should be used when you traverse a waiting cycle for detecting a deadlock. */
         …. /* more variables will be needed depending on your design */
}
```

The database lock should be detailed enough to identify which transaction is holding for what record (or item). It may look something like the one below:

```
struct lock_t {
        /* The following pair of <table_id, record_id> should be used for the input to your
        hash function (or the SHA-1 hash function:
        http://www.openssl.org/docs/manmaster/crypto/sha.html ) to get the hash key in the
        lock hash buckets. */
         unsigned long table_id: // database table id
         unsigned long record_id; // database record id
          unsigned int lock_mode; // RECORD_LOCK_SHARED or
                                   // RECORD_LOCK_EXCLUSIVE
         trx_t* trx; // backpointer to the lock holder
         int lock_state; // marker to represent the state of a lock (LOGICAL_RELEASE,
                         // WAITING, etc.
         unsigned long timestamp; // This may be useful for garbage collection.
          …. // More variables may be needed depending on your design
}
```

The transaction lock table should be designed using a hash table whose entry (or bucket) points to a list of database lock objects hashed to the bucket.



Your mini-database systems should create two tables (i.e., table-A and table-B), both have the same size, and your system should get --table_size (default: 10000), --num_thread (default: number of cores), --read_num (default: 10, range 0-10), --duration (default: 30 seconds). The record in the table has three tuples: {record id, record value, last updater trx id}. The record id should start from 1. The value should be initialized between 10000 and 100000. The transaction id field should be initialized to zero.

OPERATION TYPE:

1. READ record k : just read the value of a record k.
2. UPDATE record k : you have to perform one of the following transfer operations.
   A. table-A.record_k.value – 10 and table-B.record_k.value + 10, then change both records' updater_trx_id to the trx_id of the current transaction
   B. table-B.record_k.value – 10 and table-A.record_k.value + 10, then change both records' updater_trx_id to the trx_id of the current transaction

For the UPDATE operation, you should acquire two database locks before you perform the above operations.

For example, if you want to transfer value from table-A.record_k to table-B.record_k, then you first places an X-lock for table-A.record_k and another X-lock for table-B.record_k. Once you hold two locks, then you can change two record values.

The operation sequences for accessing database records in a single transaction lifetime are as follows:

1. **BEGIN**
2. A transaction gets a new transaction id from the global atomic counter.
3. A transaction pick a *random* record id, say k and *random* table id.
4. From record k to record k+9 (**10 consecutive records**), you have to perform READ operations on **NUM_READ** record items and get the summation. Once you finish READ operations, a transaction performs UPDATE operations for the remaining **10 – NUM_READ** record items, as described above.
5. Before you access each record k, you first have to place a {S (shared) | X (exclusive)} database lock object into the lock hash table and detects any conflicting locks in the table. If there is no conflicting lock, then you can read/update the record value. Otherwise the transaction first checks the deadlock cycle it may be involved. If it detects a deadlock cycle, then your transaction locking system should abort the current transaction (which means the aborting transaction should release any lock objects it is holding, which is contained in ***vector<lock_t*> trx_locks***. If there is no deadlock, then your system should put the current transaction (i.e., thread) into sleep, waiting until the lock holder (other transaction) wakes him up.
6. After the current transaction performs required operations against records (i.e., READ/UPDATE), then a transaction should **commit**, by releasing all database locks it

is holding. Lock releasing may require waking up other waiting transactions.

7. REPEAT 1-6 until timeout (--duration).

After the entire system finishes, your system first checks the consistency of the tables by checking the summation of all record values the same before/after the test. Then your system should print out the following information:

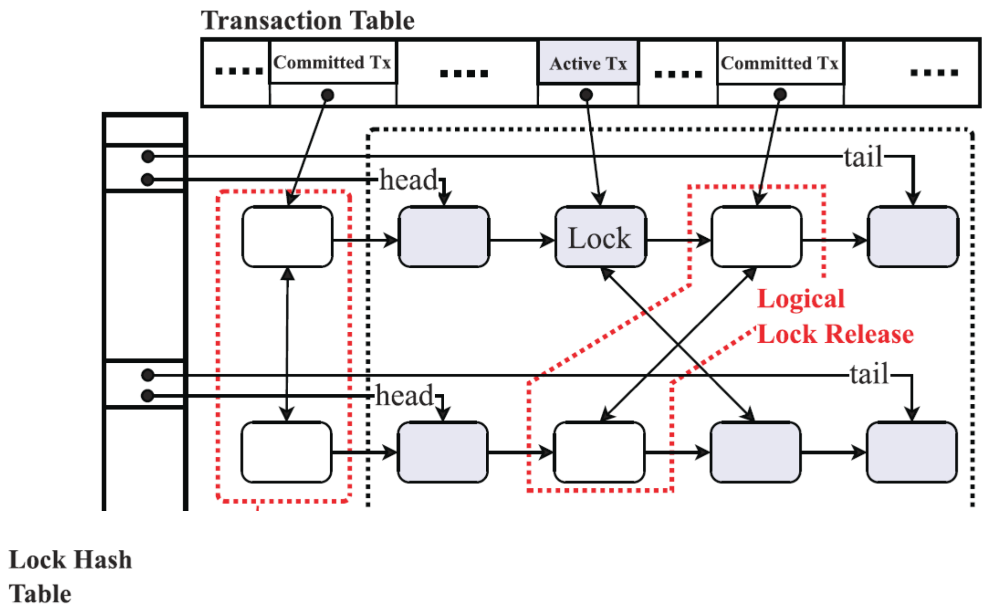
```
READ throughput: 1000000 READS and 123124512 READS/sec

UPDATE throughput: 1223123 UPDATES and 1232412312 UPDATE/sec

Transaction throughput: 12312213123 trx and 1234124124 trx/sec

Aborted transactions: 213123123 aborts and 123124125421 aborts/sec
```

The transaction locking system may look a bit like the following, after you completes all implementation with lots of optimizations:



IMPORTANT: *Your job is to design the transaction locking system as fast as you can by using any of concurrent programming techniques you learnt so far (or something else if you know), so that the system can scale well as you increase the number of threads while retaining high throughput.*

4. *Requirements & Due Date*

The due date is **<u>11:59pm 20 DEC 2015</u>** (firm deadline due to the grading pressure).

You have to submit **a very detailed document** explaining everything you design, implement, test, optimize and learn, with **a source tarball**.

Good luck

&

Have fun