Cloud Databases Assignment 4

Group 2
102061105 Yun-Sheng, Chang
102061212 Jian-Hao, Huang

May 3, 2017

In this assignment, we implement the **conservative locking scheme** for concurrency control, and compare the performance result with the original **timeout-rollback locking scheme**.

1 Implementation

1.1 New Class: RecordKey

SerializableConcurrencyMgr use *String*, *BlockId* and *RecordId* as the **synchronized objects** to protect files, blocks and records from race condition, respectively; our implementation of *ConcurrencyMgr*, *ConservativeConcurrencyMgr*, use a particular class, *RecordKey*, to protect records from race condition.

```java
private String tableName;
private String field;
private Constant value;
private boolean isRead;
private int hashCode;

public RecordKey(String tableName, String field, Constant value, boolean isRead) {
  this.tableName = tableName;
  this.field = field;
  this.value = value;
  this.isRead = isRead;

  String s;
  s = tableName + field + value.toString();
  this.hashCode = s.hashCode();
}
```

RecordKey.java

Although *RecordKey* has 5 member variables, only 3 of them, *tableName*, *field*, *value*, are used to represent a record. Member variable *isRead* is used to indicate which type of locks (shared or exclusive) to acquire, and regardless of the lock type, locks should always take effect on the same record if the *tableName*, *field* and *value* are identical. Member variable *hashCode* is initialized in constructor to avoid the recomputation of hash code.

```java
@Override
public boolean equals(Object obj) {
  if (obj == null || !(obj.getClass().equals(RecordKey.class)))
    return false;
  RecordKey r = (RecordKey) obj;
  return tableName.equals(r.tableName) && field.equals(r.field) &&
      value.equals(r.value);
}

@Override
public int hashCode() {
  return hashCode;
}
```

RecordKey.java

1.2 LockTable

We modify *LockTable* based on the following two requirements:

1. locks can be tested for their availability (lockable or not) **outside** the *LockTable*;
2. tx with higher tx number should always acquire the same lock **after** the tx with lower tx number.

We first describe the reason for requirement 1 and how we modify the API. By definition, a transaction should **atomically** acquire all the locks it needs before its execution under the conservative locking scheme; previous implementation of *LockTable*, however, only exposes interfaces like *sLock()* and *xLock()* which invoke *wait()* immediately when the lock is unlockable, failing to make the process of acquiring all locks atomic. Thus, we define two synchronized wrapper functions in *LockTable* so that the concurrency manager can test the availability of locks before actually acquire the locks.

```java
synchronized boolean sLockableSync(Object obj, long txNum) {
  return sLockable(obj, txNum);
}

synchronized boolean xLockableSync(Object obj, long txNum) {
  return xLockable(obj, txNum);
}
```

LockTable.java

Requirement 2 is in the specification of this assignment and we describe our implementation. We add a queue of type *TreeSet* to the inner class *Lockers*.

```java
class Lockers {
  // a variety of lockers ...
  TreeSet<Long> queue;

  Lockers() {
    // initialize lockers ...
    queue = new TreeSet<Long>();
  }
}
```

LockTable.java

Everytime a transaction fails to atomically acquire the locks for its read/write sets, the transaction would enqueue itself to every *Lockers* that the read/write sets associate with; and when the transaction can finally acquire all the locks, it will dequeue itself before it actually acquires those locks (more detail in Section 1.4).

```java
synchronized void enqueue(Object obj, long txNum) {
  Lockers lks = prepareLockers(obj);
  lks.queue.add(txNum);
}

synchronized void dequeue(Object obj, long txNum) {
  Lockers lks = lockerMap.get(obj);
  lks.queue.remove((Long) txNum);
}
```

LockTable.java

To forbid tx with higher tx number acquiring the same lock prior to tx with lower tx number, we modify the *Lockable()* interface, with one additional condition: **a lock is only available to the tx that has the minimal value of tx number among the queue**. We only modify the *sLockable()* and the *xLockable()* since the MGL is removed from the specification.

```java
private boolean havePriority(Object obj, long txNum) {
  Lockers lks = lockerMap.get(obj);
  return lks == null || lks.queue.size() == 0 || txNum <= lks.queue.first();
}

private boolean sLockable(Object obj, long txNum) {
  return (!xLocked(obj) || hasXLock(obj, txNum))
      // unchanged conditions ...
      && havePriority(obj, txNum);
}
```

```java

private boolean xLockable(Object obj, long txNum) {
  return (!sLocked(obj) || isTheOnlySLocker(obj, txNum))
      // unchanged conditions ...
      && havePriority(obj, txNum);
}
```

LockTable.java

1.3 MicrobenchmarkProc

The conservative locking scheme only works when the read/write sets are collected in advance. Thus, we construct a *List* of *RecordKey*, with each *RecordKey* being one record that will be accessed in a transaction.

```java
private List<RecordKey> recordKeys = new ArrayList<RecordKey>();

public void prepare(Object... pars) {
  paramHelper.prepareParameters(pars);
  for(int i = 0; i < paramHelper.getReadCount(); i++){
    Constant id = new IntegerConstant(paramHelper.getItemId(i));
    RecordKey recKey = new RecordKey("item", "i_id", id, true);
    recordKeys.add(recKey);
  }
  for(int i = 0; i < paramHelper.getWriteCount(); i++){
    Constant id = new IntegerConstant(paramHelper.getItemId(i));
    RecordKey recKey = new RecordKey("item", "i_id", id, false);
    recordKeys.add(recKey);
  }
}
```

MicrobenchmarkProc.java

Then, the *ConservativeConcurrencyMgr* of a transaction will try to acquire all the locks with the collected read/write sets before the transaction is actually being executed.

```java
public SpResultSet execute() {
  tx = VanillaDb.txMgr().newTransaction(Connection.TRANSACTION_SERIALIZABLE,
      false);
  ((ConservativeConcurrencyMgr) tx.concurrencyMgr()).acquireAllLocks(recordKeys);
  // execute the transaction ...
}
```

MicrobenchmarkProc.java

1.4 ConservativeConcurrencyMgr

To acquire all the locks atomically, the *ConservativeConcurrencyMgr* would perform the following three steps in a critical section:
Step 1: Test if every lock is lockable; if there exist at least one lock that is unlockable (2 possible reasons: the lock is currently hold by another transaction or there exist a transaction with lower transaction number also waiting for the same lock), the *ConservativeConcurrencyMgr* would queue **all** the locks it requires and invoke *wait()*.

```java
public void acquireAllLocks(List<RecordKey> recordKeys) {
  boolean allLockable = false;
  boolean queued = false;

  synchronized(lockTbl){
    // Step 1
    while (!allLockable) {
      allLockable = true;
      for (int i = 0; allLockable && i < recordKeys.size(); i++)
        if (!tryRecordKey(recordKeys.get(i)))
          allLockable = false;

      if (!allLockable) {
        if (!queued) {
          enqueue(recordKeys);
          queued = true;
```

```java
            }
            try {
              lockTbl.wait();
            } catch (Exception e) {
              e.printStackTrace();
            }
          }
        }
        ...
      }
    }
```

ConservativeConcurrencyMgr.java

Step 2: When all the locks can certainly be acquired, the *ConservativeConcurrencyMgr* needs to dequeue the waiting locks if it has failed to acquire all the locks atomically at least one time.

```java
public void acquireAllLocks(List<RecordKey> recordKeys) {
  synchronized(lockTbl){
    ...
    // Step 2
    if (queued)
      dequeue(recordKeys);
    ...
  }
}
```

ConservativeConcurrencyMgr.java

Step 3: Finally, acquire all the locks for the read/write sets.

```java
public void acquireAllLocks(List<RecordKey> recordKeys) {
  synchronized(lockTbl){
    ...
    // Step 3
    for (RecordKey r : recordKeys) {
      if(r.isRead())
        readRecordKey(r);
      else
        modifyRecordKey(r);
    }
    ...
  }
}
```

ConservativeConcurrencyMgr.java

2 Results

2.1 Experiment Environment

Intel Core i7-4790 CPU @ 3.60 GHz
32 GB RAM
1TB HDD
Ubuntu Linux 16.04
Java Direct I/O (Jaydio)

2.2 Speed Up

We first compare the performance result using conservative locking scheme with the one using timeout-rollback locking scheme.

	Throughput (tx)	Average Latency (ms)
Timeout-Rollback	5788	45
Conservative	106720	28

The experiment is conducted with **50 RTEs**, **conflict ratio = 0.001**, **write ratio = 0.5** and **bufferpool size = 100000**. The speed up is roughly **18x**.

2.3 Vary Bufferpool Size

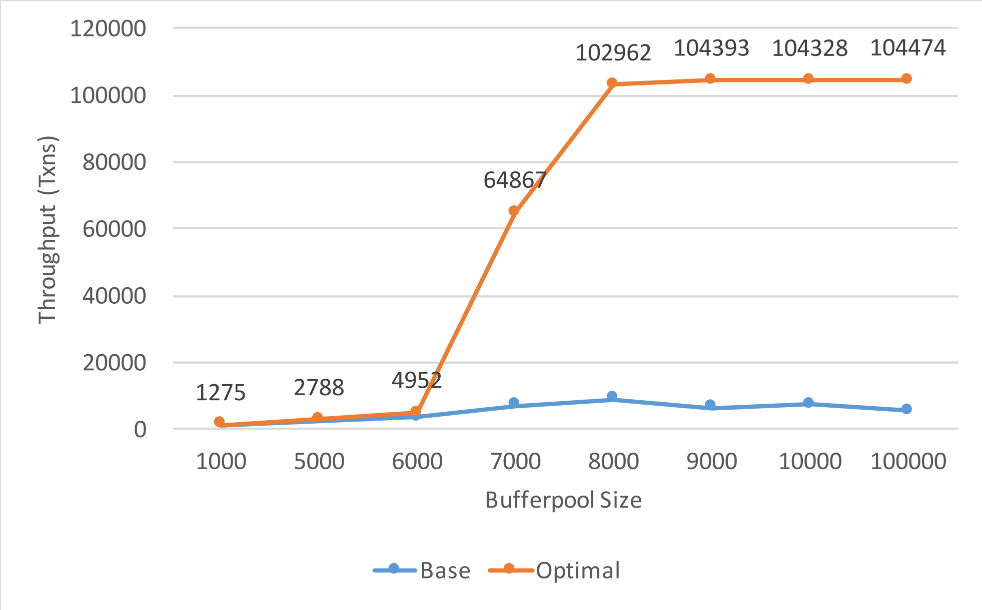

Figure 1: Throughput vs. Bufferpool Size

We vary the bufferpool size from 1000 to 100000 and we find the same trend as in assignment 3 for our conservative locking scheme: **before the match point, where the working sets just fit into the bufferpool, the throughput is low; after the match point, the throughput is high**. However, we do not find the same trend with the timeout-rollback locking scheme, we think it's because **the bottleneck is not in the size of bufferpool but in the deadlock issues**.

2.4 Vary Number of RTEs

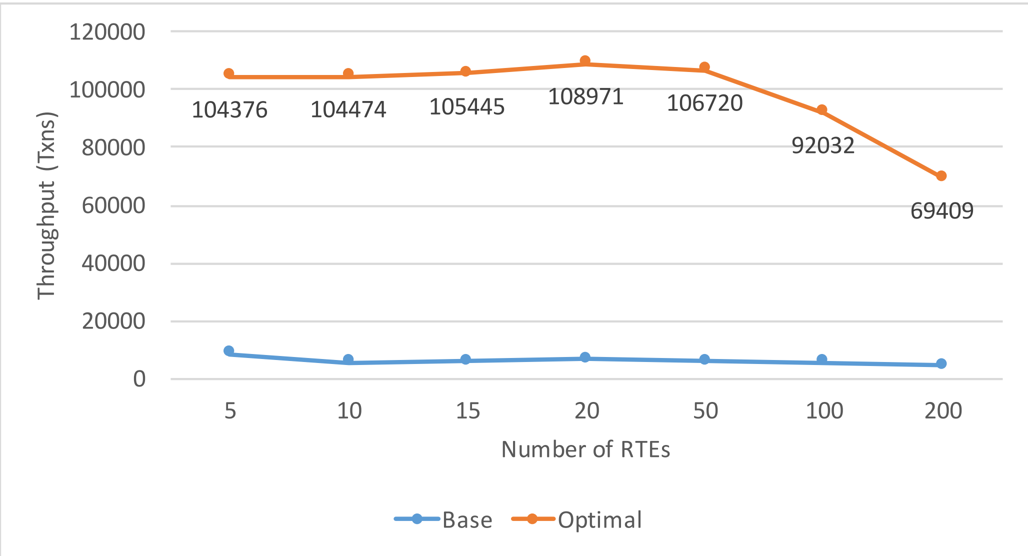

Figure 2: Throughput vs. Number of RTEs

We vary the number of RTEs from 5 to 200 and we find that the throughput starts to drop when the number of RTEs exceeds 100. We think it's because **the more RTEs we have, the more conflict operations there will be, and naturally, the more time we will have to spend on the locks**.