Team5 Assignment5 Report1

107062313 黃寶萱　107062117 李采蓉　107062317 陳怡汝

◈ Implementation

■ **Part1 : Implement Conservative Concurrency Locking**

1. Conservative : ConservativeConcurrencyMgr

   在 concurrency 中的 conservative 資料夾中**新增一個 ConservativeConcurrencyMgr**，extends 自 ConcurrencyMgr，所有的 functions 參照 SerializableConcurrencyMgr 的寫法。負責實作 Conservative Concurrency，用來處理實際 LockTable 的維護以及不同層級的 lock function，包含管理 file/block/record 不同的讀寫 lock 等等。

2. Conservative : MyLockTable

   在 concurrency 中的 conservative 資料夾中**新增一個 MyLockTable**，負責在 ConservativeConcurrencyMgr 中引入型態為 MyLockTable 的 lock table *lockTbl_1*。由於在 ConservativeConcurrencyMgr 無法呼叫到原本在 concurrency 資料夾中型態為 LockTable 的 *lockTbl*，故自行在 concurrency 中的 conservative 資料夾內新增一個功能一樣的 lock table。

3. Conservative : MyLockAbortException

   在 concurrency 中的 conservative 資料夾中新增一個 MyLockAbortException，負責 MyLockTable 的 exception 處理。原因同上述第二點的 MyLockTable，因此需要另外新增一個功能一樣的檔案。

4. TransactionMgr

   在 TransactionMgr 的 *createTransaction()* 中新增 *case 16*，因為其他的 isolation level 都有自己的代表的數值(SERIALIZABLE 代表 8、REPEATABLE_READ 代表 4、TRANSACTION_READ_COMMITTED 代表 2)，因此我們將 16 作為我們要實作的 CONSERVATIVE 的代表數值。這個 function 會根據 isolationLevel 這個參數設定該 Transaction 的 ConcurrencyMgr，16 代表該 Transaction 的 ConcurrencyMgr 為 ConservativeConcurrencyMgr。

```
case 16:
   // System.out.println("In 16");
   try {
      Class<?> partypes[] = new Class[1];
      partypes[0] = Long.TYPE;
      Constructor<?> ct = conservativeConcurMgrCls.getConstructor(partypes);
      concurMgr = (ConcurrencyMgr) ct.newInstance(new Long(txNum));
   } catch (Exception e) {
      e.printStackTrace();
   }
   break;
```

5. StoredProcedure

StoredProcedure 的 *prepare()* 中在 createTransaction 時將傳入的 isolationLevel 更改成 16，使得該 Transaction 的 ConcurrencyMgr 為 ConservativeConcurrencyMgr。

```
public void prepare(Object... pars) {
    // prepare parameters
    paramHelper.prepareParameters(pars);

    // create a transaction
    boolean isReadOnly = paramHelper.isReadOnly();
    tx = VanillaDb.txMgr().newTransaction(
        16, isReadOnly);
}
```

6. org.vanilladb.core.vanilladb.properties

在 properties 中引入上方新增的 ConservativeConcurrencyMgr 以及將 default 的 isolation level 設為 16，使其 default 為 CONSERVATIVE。

```
org.vanilladb.core.storage.tx.TransactionMgr.CONSERVATIVE_CONCUR_MGR
                =org.vanilladb.core.storage.tx.concurrency.ConservativeConcurrencyMgr
org.vanilladb.core.remote.jdbc.RemoteConnectionImpl.DEFAULT_ISOLATION_LEVEL=16
```

■ Part2 : Modify StoredProcedure and MicroTxnProc

1. StoredProcedure

在 execute 裡我們用 *DoCheckTxs* 和 *CanDoTxs* 兩個 TreeSet 來保障 TXs 的執行順序。在執行 tx 時我們會先呼叫 *staticSql()* 統計 tx 的 read/write record 以及判斷是否可以執行 tx。

若不能執行，tx 會被加到 *DoCheckTxs* 中，因此我們使用 *while* 迴圈在 *tx.checkAgain* 時持續檢查當前 tx 是否可以執行。tx 可以執行便會從 *DoCheckTxs* 移到 *CanDoTxs*，因此我們的第二個 *while* 圈即在判斷當 tx 為 *CanDoTxs* 中的第一個 element (即 txNum 的 tx)，且 *CanDoTxs* 不是 empty 時，將其移出 *CanDoTxs* 並執行，用這樣的方式來確保我們的執行順序符合根據 txNum 從小到大執行 tx。

```
public SpResultSet execute() {
    boolean isCommitted = false;
    long txNum = tx.getTransactionNumber();

    try {
        staticSql();
        while (VanillaDb.txMgr().inDocheckTxs(tx))
        {
            if (tx.checkAgain) {
                tx.checkAgain = false;
                staticSql();
            }
        }

        while (txNum != VanillaDb.txMgr().getFirstCanDoTxs() && VanillaDb.txMgr().getFirstCanDoTxs()!=-1) {
        }

        VanillaDb.txMgr().removeCanDoTxs(txNum);
        executeSql();
```

2. MicroTxnProc

在這裡我們新增了 function *staticSql()*，先記錄所有要執行的 record 以及 read/wirte 動作到 *tx.RecordTypeMap* 中，接著判斷目前所有要讀取的 record 的 lock 狀態，若是該 tx 需要的 lock 與其他 tx 都沒有衝突時，*txCanDo* 為 true 代表可以執行該 tx，因此當 *txCanDo=true* 我們將這些 records lock 起來並將 tx 從 *DoCheckTxs* 移除，加到 *CanDoTxs* 中，反之若該 tx 目前無法拿到所有 lock 便加入 *DoCheckTxs* 中繼續進行拿 lock 的檢查，直到可以執行。

```
@Override
protected void staticSql()
{
    MicroTxnProcParamHelper paramHelper = getParamHelper();
    Transaction tx = getTransaction();
    TableInfo ti = VanillaDb.catalogMgr().getTableInfo("item", tx);

    if (tx.RecordTypeMap.isEmpty())
    {
        // SELECT
        for (int idx = 0; idx < paramHelper.getReadCount(); idx++) {
            //count
            long iid = paramHelper.getReadItemId(idx);
            // mem record type
            tx.RecordTypeMap.put(iid, "r");
        }
        // UPDATE
        for (int idx = 0; idx < paramHelper.getWriteCount(); idx++) {
            long iid = paramHelper.getWriteItemId(idx);
            // mem record type
            tx.RecordTypeMap.put(iid, "w");
        }
    }

    boolean txCanDo = VanillaDb.txMgr().checkRecord(tx.RecordTypeMap);

    if (txCanDo)
    {
        VanillaDb.txMgr().lockRecord(tx.RecordTypeMap);
        VanillaDb.txMgr().removeDoCheckTxs(tx);
        VanillaDb.txMgr().addCanDoTxs(tx.getTransactionNumber());
    }

    else {
        // add to DoCheckList
        VanillaDb.txMgr().addDoCheckTxs(tx);
    }
}
```

■ Part3 : Modify Transaction and TransactionMgr

1. Transaction

我們對每個 Tx 新增一個 boolean 變數 *checkAgain* 以及 HashMap RecordTypeMap。

a. checkAgain：用來記錄該 Tx 是否因為無法拿到所有需要的 lock 因而當有其他 Tx commit 時該 Tx 需要重新檢查一次。

b. RecordTypeMap：用來記錄該 Tx 要讀取的所有 record，並會記錄對每個 record 要 read/write。

2. TransactionMgr

我們新增兩個 TreeSet object : *doCheckTxs*, *canDoTxs*，*doCheckTxs* 用來記錄那些拿不到所有需要的 lock 的 TXs，若 tx 加到 *doChekcTxs* 中代表 tx 需要 waiting 並且等到有其他 tx commit 時他會再重新檢查能否拿到所有 lock，*canDoTxs* 則用來記錄那些確定可以拿到所有 lock 並執行的 TXs。

```
private TreeSet<Long> activeTxs = new TreeSet<Long>();
private TreeSet<Transaction> doCheckTxs = new TreeSet<Transaction>(new Helper());
private TreeSet<Long> canDoTxs = new TreeSet<Long>();


private long nextTxNum = 0;
// Optimization: Use separate lock for nextTxNum
private Object txNumLock = new Object();

//[TODO]
HashMap<Long, List<Long>> RecordLockMap = new HashMap<Long, List<Long>>();
```

HashMap *RecordLockMap* 用來記錄 record 的 read/write 情況，利用這個 map 查看是否可以拿到該 record 的 read lock 或 write lock，以下為 Tx 要拿 lock 時會呼叫到的 API：

a. boolean *checkRecord* (HashMap<Long, String> map) :

tx 在執行 *executeSql()*之前會在 *staticSql()*中呼叫到 *checkRecord()*檢查可否拿到所有的 lock，傳入的參數 Long type 代表某一個 record 的 iid，String type 代表"r"或"w"，因為可以同時有多個 Tx 一起讀取，但一次只能有一個 Tx update record 並且不可以有其他 Tx 讀取，所以我們需要知道該 Tx 是要拿該 record 的 read lock 還是 write lock。若該 Tx 一連串的 SELECT、UPDATE 指令全部都可以拿到 lock 便 return True，反之 return False。

	"r"	"w"	
iid_1	4	0	→ 可以同時有多個 Tx 拿取 read lock
iid_2	0	1	→ 不可以拿 read lock 與 write lock
iid_3	0	0	→ 同時都為 0 才可以拿 write lock

```java
public boolean checkRecord(Long iid, String type)
{
    if (!RecordLockMap.containsKey(iid))
    {
        ArrayList<Long> initRecord = new ArrayList<Long>(Arrays.asList(0L, 0L));
        RecordLockMap.put(iid, initRecord);
    }
    if (type == "r")
    {
        // check if nobody write
        if (RecordLockMap.get(iid).get(1) == 0)
            return true;
    }
    else
    {
        // check if nobody write and read
        if (RecordLockMap.get(iid).get(0) == 0 && RecordLockMap.get(iid).get(1) == 0)
            return true;
    }
    return false;
}

public boolean checkRecord(HashMap<Long, String> map)
{
    synchronized(this){
        for (Map.Entry<Long, String> set : map.entrySet())
        {
            boolean canUse = checkRecord(set.getKey(), set.getValue());
            if (canUse == false)
                return false;
        }
        return true;
    }
}
```

b. void *lockRecord*(HashMap<Long, String> map)

若 checkRecord() return True 便會呼叫 lockRecod() update 該 Tx 要讀取的 record 在 RecordLockMap 中紀錄的 read/write 值。

```java
public void lockRecord(Long iid, String type)
{
    if (type == "r")
    {
        Long rlockNum = RecordLockMap.get(iid).get(0);
        rlockNum++;
        Long wlockNum = RecordLockMap.get(iid).get(1);
        ArrayList<Long> readLock = new ArrayList<Long>(Arrays.asList(rlockNum, wlockNum));
        RecordLockMap.replace(iid, readLock);
    }
    else
    {
        Long lockNum = RecordLockMap.get(iid).get(0);
        Long wlockNum = RecordLockMap.get(iid).get(1);
        wlockNum++;
        if (wlockNum != 1) System.out.println("Wrong! wlockNum=" + wlockNum);
        ArrayList<Long> writeLock = new ArrayList<Long>(Arrays.asList(lockNum, wlockNum));
        RecordLockMap.replace(iid, writeLock);
    }
    return;
}

public void lockRecord(HashMap<Long, String> map)
{
    synchronized(this){
        for (Map.Entry<Long, String> set : map.entrySet())
        {
            lockRecord(set.getKey(), set.getValue());
        }
    }
}
```

c. *removeDoCheckTxs*(Transaction tx)、*addCanDoTxs*(long tx)

若 Tx 可以拿到所有 lock 便呼叫 *removeDoCheckTxs()*將該 Tx 從在等到檢查是否可以拿到 lock 的 doCheckTxs set 中移除，並呼叫 *addCanDoTxs()*將該 Tx 加入可以真正執行的 canDoTxs set 中。

d. *addDoCheckTxs*(Transaction tx)、*RemoveCanDoTxs*(long tx)

若 Tx 還無法拿到所有 lock 便呼叫 *addDoCheckTxs()*將他加入 *doCheckTxs* 中，代表需要重新檢查，而 Tx 開始執行便呼叫 *RemoveCanDoTxs()*將他移出 *canDoTxs*。

```
public void addCanDoTxs(long tx)
{
    synchronized(this){
        canDoTxs.add(tx);
    }
}
public void removeCanDoTxs(long tx)
{
    synchronized(this){
        canDoTxs.remove(tx);
    }
}
```

```
public void addDoCheckTxs(Transaction tx)
{
    synchronized(this){
        doCheckTxs.add(tx);
    }
}

public void removeDoCheckTxs(Transaction tx)
{
    synchronized(this){
        doCheckTxs.remove(tx);
    }
}
```

e. void *notifyDoCheckTxs*()

當 Tx commit 時，代表他會開始釋放他手上拿的 lock，其他在等待的 Tx 便有機會拿到他需要的 lock，因此呼叫 *notifyDoCheckTxs()*將在 *doCheckTxs* 中的所有 Tx *checkAgain* 變數設成 True，那些 waiting 的 TXs 便會重新進到 *staticSql()*檢查。

```
public void notifyDoCheckTxs()
{

    synchronized(this){
        for (Transaction tx : doCheckTxs)
        {
            tx.checkAgain = true;
        }
    }
}
```

```
@Override
public void onTxCommit(Transaction tx) {

    // activeTxsLock.readLock().lock();
    // try {
    // threadTxNums[(int) Thread.currentThread().getId()] = -1L;
    // } finally {
    // activeTxsLock.readLock().unlock();
    // }

    synchronized (this) {
        releaseRecord(tx.RecordTypeMap);
        notifyDoCheckTxs();
        activeTxs.remove(tx.getTransactionNumber());
    }
}
```

f. void *releaseRecord*(HashMap<Long, String> map)

Tx commit 後代表他要 release lock，因此也需要從新 update 該 Tx 讀取的 record 在 *RecordLockMap* 中的值，做法與 *lockRecord()*相似。

```
public void releaseRecord(Long iid, String type)
{
    if (type == "r")
    {
        Long rlockNum = RecordLockMap.get(iid).get(0);
        rlockNum--;
        Long wlockNum = RecordLockMap.get(iid).get(1);
        ArrayList<Long> readLock = new ArrayList<Long>(Arrays.asList(rlockNum, wlockNum));
        RecordLockMap.replace(iid, readLock);
    }
    else
    {
        Long rlockNum = RecordLockMap.get(iid).get(0);
        Long wlockNum = RecordLockMap.get(iid).get(1);
        wlockNum--;
        if (wlockNum != 0) System.out.println("Wrong! wlockNum=" + wlockNum);
        ArrayList<Long> writeLock = new ArrayList<Long>(Arrays.asList(rlockNum, wlockNum));
        RecordLockMap.replace(iid, writeLock);
    }
    return;
}
public void releaseRecord(HashMap<Long, String> map)
{
    synchronized(this){
        for (Map.Entry<Long, String> set : map.entrySet())
        {
            releaseRecord(set.getKey(), set.getValue());
        }
    }
}
```

◈ Challenge of implementing conservative locking for the TPC-C benchmark

因為 TPC-C benchmark 可以做 insert operation，而 conservative locking 需要在 Tx 執行前就拿到所有需要的 lock，並且只有當 Tx commit 才會 release lock。若該 Tx 要 insert 一個 record，他需要拿取該 record 與上層 block-level、file-level 的 X lock，該 Tx 的另一個 operation 是要 read，需要拿 S lock，但在 lock table 的設定上若拿了 X lock 便不允許任何人拿 S lock，需要 waiting，這樣的運作模式會導致該 Tx 自己在等自己，使他永遠無法開始執行，benchmark 的結果可能會因此有許多的 TXs 沒有執行就 abort。

◈ Experiments

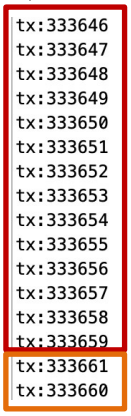
■ Environment setting :

Intel Core i5 @ 2 GHz, 16 GB RAM, 1TB SSD, MacOS Monterey 12.2.1

■ Experiments result

Conservative Concurrency with lower txNums acquire locks **before** higher txNums :

當 transaction 執行 *executeSql()*時我們於 system console 印出其 txNum，Output 如下



可以發現紅色框框的部分為所獲得的 lock 沒有衝突的情況下，較低的 txNums 會較高的 txNums 早執行。而橘色框框的部分則有出現等待 lock 的情況。

Compare the throughputs with different parameters :

1. RTE = 2, BUFFER_POOL_SIZE = 102400

	Committed txs	Aborted txs	Avg latency
Serialized	256762	7	0.017
Conservative	**275041**	**0**	0.011

2. RTE = 10, BUFFER_POOL_SIZE = 102400

	Committed txs	Aborted txs	Avg latency
Serialized	482560	89	0.694
Conservative	**542541**	**0**	0.532

3. RTE = 2, BUFFER_POOL_SIZE = 10240

	Committed txs	Aborted txs	Avg latency
Serialized	**272687**	8	0.014
Conservative	255934	**0**	0.017

4. RTE = 2, BUFFER_POOL_SIZE = 409600

	Committed txs	Aborted txs	Avg latency
Serialized	249532	3	0.041
Conservative	**243348**	**0**	0.032

- Explanation

  從上面三張表格可以發現我們的 conservative concurrency manager 可以正確避免 deadlock 的發生。

  1. 由第一張圖可以發現 RTE 數量相同時，conservative 的方式因為不會有 deadlock 發生，也不會有處理 deadlock 的 overhead，因此比 serialize 有更多 committed txs。
  2. 而在 RTE 增加時，因為有更多 transactions 同時要進行搶 lock 的動作，因此可以發現 serialize 的方法發生 deadlock 的次數明顯增加，而 conservative 的方式因為可以避免 deadlock 的發生，因此在更多 RTE 的情況下比起 *RTE=2* 時有更高的平行度，conservative 方式改善效果更為顯著。

  比較第一張及第三、四張表格，當 *BUFFER_POOL_SIZE* 從 102400 調整成 10240 時，conservative 的 committed txs 數量**減少** 19107，推測是因為 buffer pool size 縮小造成可以 cache 的 record 減少，因此 txs 執行時間增加，而在 *BUFFER_POOL_SIZE* 從 102400 調整成 409600 時，committed txs 數量**沒有增加**，推測有可能是因為我們執行的 transaction 都比較短小，下一個 tx fetch 到的 record 跟之前的 tx 之間可能相差比較遠，所以還是要重新 cache record，因此增加 buffer size 在執行多個短小 transaction 的情況下可能幫助不大。另外有可能因為系統內的 swap 空間被占用，導致操作變慢、降低性能。藉由這次實驗我們了解到 BUFFER_POOL_SIZE 的設置需要考量實際狀況，設置的值須在合理範圍內，太大太小都會讓表現更差。