Assignment 5 Phase 1 Report

110062229 翁語辰

110081014 程詩柔

110062171 陳彥成

Implementation

1.ConservativeLockTable.java

在bookLock中會去把該txNum加入對應的locker的requestQueue中

```
public void bookLock(PrimaryKey p, long txNum) {
    synchronized (getAnchor(p)) {
        Lockers lockers = prepareLockers(p);
        lockers.requestQueue.add(txNum);
    }
}
```

在 getXLock 和 getSLock 方法中，當要求鎖時，它會先看你是否已經有鎖了，再利用鎖的requestQueue和鎖當前的狀態，來判斷是否可以獲取鎖。如果無法獲取鎖，則該tx會進入等待狀態，直到它能夠獲取到所需的鎖。

```
public void getSLock(Object p, long txNum) {
    Object anchor = getAnchor(p);
    synchronized (anchor) {
        Lockers lockers = prepareLockers(p);
        if (lockers.slockers.contains(txNum)) {
            lockers.requestQueue.remove(txNum);
            return;
        }
        try {
            while ((lockers.requestQueue.peek() != null && lockers.requestQueue.peek().longValue() != txNum)
                    || (lockers.xlocker != -1 && lockers.xlocker != txNum)) {
                getAnchor(p).wait();
                lockers = prepareLockers(p);
            }

            lockers.requestQueue.poll();
            lockers.slockers.add(txNum);

            anchor.notifyAll();
        } catch (Exception e) {
            throw new LockAbortException();
        }
    }
}
```

在釋放鎖時，會通知其他等待鎖的執行緒，讓它們有機會繼續執行並獲取鎖。這樣的等待和通知機制確保了沒有tx會永遠等待，而避免deadlock。

```
public void releaseSLock(Object p, long txNum) {
    Object anchor = getAnchor(p);
    synchronized (anchor) {
        Lockers lockers = lockerMap.get(p);
        if (lockers == null)
            return;
        if (lockers.slockers.contains(txNum)) {
            lockers.slockers.remove(txNum);
            if (lockers.slockers.size() == 0)
                anchor.notifyAll();
        }
        if (lockers.slockers.isEmpty() && lockers.xlocker == -1 && lockers.requestQueue.isEmpty())
            lockerMap.remove(p);
        anchor.notifyAll();
    }
}
```

在獲取 X Lock時，如果發現已經有其他tx持有了 X Lock，或者有其他事務持有了 S Lock並且不是當前tx，那麼當前tx將進入等待狀態。這保證了同一時間只有一個tx能夠擁有 X Lock。

在獲取 S Lock時，如果發現已經有其他tx持有了 X Lock，那麼當前tx將進入等待狀態；如果發現已經有其他tx持有了 S Lock，但不是當前tx本身，那麼當前tx也將進入等待狀態。這保證了當一個事務持有 X 鎖時，其他tx不能獲取S Lock，從而確保了寫入操作不會被打擾。

2.ConservativeConcurrencyMgr.java

bookPrimaryKeys 方法可以book要用的主鍵。它會對每個要write的主鍵進行X_Lock，並將其添加到bookKeys和write set中；對於每個要read的主鍵，如果它還未被預訂，則對其進行 S_Lock，並將其添加到bookKeys和read set中。

acquireAllLock 方法會獲取事務所有Lock。它對write set中的每個主鍵拿到X_Lock，對於read set中的每個主鍵，如果它不在寫入集合中，則進行 S_Lock。

3.PrimaryKey.java

使用了教授提供的架構，實作getter方法。另外選用幾個質數用作hashcode的值。還實作equals算法，比較兩個屬性tableName和keyEntryMap後回傳。

4.StoredProcedure.java

新增加prepareRWSet 抽象方法，用於準備write/read set。子類必須實現這個方法以確定存儲過程中需要讀取和寫入的data。使用ConservativeConcurrencyMgr來serial的管理Lock。

5.MicroTxnProc.java

實作了從StoredProcedure繼承的方法prepareRWSet，確定transaction中需要讀取和寫入的數據，對於每個讀取和寫入的項目，它會根據paramhelper中提供的資料，創建對應的PrimaryKey並put到讀取read set或write set中。

What is the challenge of implementing conservative locking for the TPC-C benchmark?

TPC-C benchmark由於會有predicate的存在，我們在還沒執行sql之前沒辦法求出該tx需要的readSet和writeSet，所以若是照著目前的implementation來執行會出現問題。

Experiments

- **Environment**:
  11th Gen Intel(R) Core(TM) i5-1135G7@ 2.40GHz,16GB RAM,512GB NVMe INTEL,Windows11
- **Compare the throughputs before and after your modification using the given benchmark & loader**

  Prameter : RW_TX_RATE=0.8

  before modification

```
# of txns (including aborted) during benchmark period: 210731
MICRO_TXN - committed: 210664, aborted: 67, avg latency: 0 ms
TOTAL - committed: 210664, aborted: 67, avg latency: 1 ms
```

after modification

```
20240508-214945-microbench.txt
1   # of txns (including aborted) during benchmark period: 29151
2   MICRO_TXN - committed: 29151, aborted: 0, avg latency: 4 ms
3   TOTAL - committed: 29151, aborted: 0, avg latency: 4 ms
```

修改前後比較:

1. 修改後abort數降為0:

   由於使用conservative locking 後，transaction會拿到所有lock把所有object都lock住，執行完後才一次放掉拿到的lock，如此一來，就不會像原先的2PL會發生兩個transcation同時hold住對方需要的lock導致雙方無法執行，deadlock發生。當wait到一定時間後，等待時間過長就會被abort掉的問題。

2. 修改後committed數減少,avg latency上升:

由於conservative locking會使tx沒辦法同步執行，一次只會有一個travsaction執行，所以會使committed數大幅降低。

- **Observe and discuss the impact of buffer pool size to your new system**

buffer pool size	committed	aborted	avg latency
1024000	26726	0	4
102400	29151	0	4
50000	36214	0	3
10240	34937	0	3
5000	26054	0	5

1. 當**buffer pool size**過大

將buffer pool size 從1024000調降至50000會發現commited數逐漸增加，latency逐漸減少推測可能是因為若buffer size過大，RecordPage在

pin buffer的時候需要iterate 尋找尚未被使用的buffer會需要花費比較多時間。

2. 當**buffer pool size**過小

將buffer pool size 從50000調降至5000會發現commited數逐漸減少，latency逐漸增加，推測可能是因為buffer pool size太小會導致能存放在memory裡的資料減少，hit rate降低，若在buffer找不到資料就要不斷將disk的資料swap進buffer中，相對而言變得比較耗時，所以committed數降低，latancy值變多。

補充:

我們的code在一開始依序執行Start benchmark server、Loadtestbed 、launch benchmark。在benchmark result會因為deadlock導致committed數為0，推測是因為loadtestbed所產生出的Tx到了launch benchmark的時候被直接拿來用而造成的deadlock（可能是loadtestbed產生的Tx沒有被正確處理掉）

但是!如果是benchdb已經存在的情況下在Start benchmark server後直接launch benchmark是ok沒問題的!再麻煩助教驗證我們的code時稍微注意。