Assingment 5 report

Group members

1. 107062321 王劭元
2. 107062214 陳伯瑾
3. 107062228 陳劭愷

Implementation

RecordPage

1. 我們在RecordPage中的method setVal做了修改，我們不讓他直接去寫log以及call currentBuffer寫回buffer，以實現在private workspace去做write，因此我們一樣先去拿到這個record的X lock後，再直接去call transaction中我們自己新實作的setVal，讓每個transaction在做write的時候可以有自己的private workspace。

```
private void setVal(int offset, Constant val) {
    if (tx.isReadOnly() && !isTempTable())
        throw new UnsupportedOperationException();
    if (!isTempTable())
        tx.concurrencyMgr().modifyRecord(new RecordId(blk, currentSlot));
    tx.setVal(new RecordId(blk, currentSlot), offset, val);
    // LogSeqNum lsn = doLog ? tx.recoveryMgr().logSetVal(currentBuff, offset, val) : null;
    // currentBuff.setVal(offset, val, tx.getTransactionNumber(), lsn);
}
```

2. 我們也修改了RecordPage中的getVal，由於現在新write的資訊沒有直接放回buffer，因此若去buffer找不一定會找得到，因此我們會先從transaction的private workspace中查找，若找不到再去buffer中查找。

```
private Constant getVal(int offset, Type type) {
    if (!isTempTable())
        tx.concurrencyMgr().readRecord(new RecordId(blk, currentSlot));
    // return currentBuff.getVal(offset, type);
    Constant val = tx.getVal(new RecordId(blk, currentSlot), offset);
    if (val == null)
        return currentBuff.getVal(offset, type);
    return val;
}
```

LockTable

1. 首先我們先多定義了一個c lock，並且因為他只能lock single item，因此在Locker這個class的建構子時，c locker不需要是一個HashSet。並且也將Locker class的 to string多加了 c lock
2. 在avoidDeadLock的部分，根據gitlab上的Compatibility table，我們也將c lock加進去。例如：當今天有人拿著sLock，但若有cLock, IxLock, SixLock想去拿去同一level的Lock時，就會去判斷他們的timestamp，比較年輕的就會被Abort。
3. 在寫cLock這個method時，基本上跟其他的Lock method差不多。
4. 在release、releaseAll, releaseLock的地方也把cLock相關的補上去
5. 加了cLocked, hasCLock，來判斷這個item有沒有被cLock lock，以及有沒有拿著cLock
6. 最後新增一個cLockable來判斷現在這個item可不可以被cLock Lock起來。

ConcurrencyMgr

我們在這裡新增的一個method，realModifyRecord

1. 這個method的意思是，當今天一個Transaction真的要將private workspace中的值，寫回buffer時，會像ConcurrencyMgr索取這個lock，這時候，再根據Lock policy定義好的policy，將這個c Lock 交給transaction。

```
@Override
public void realModifyRecord(RecordId recId) {
    lockTbl.cLock(recId, txNum);
}
```

2.

Transaction

我們在這裡新增了一個資料結構以及兩個method，最後修改了commit method

1. HashMap<RecordId, HashMap<offset, Val>> workspace

   這是我們新增的資料結構，是雙層的HashMap，第一層的key是選擇用RecordId，會記住我們是在哪個block上的哪個slot，後面還要包一個HashMap是因為有可能對一個Record上的許多field做修改，因此會需要再知道offset是多少，才能找到要修改的field，因此第二層的HashMap的key就是offset，最後才是要將這個field改為什麼樣的Val。

```
private HashMap<RecordId, HashMap<Integer, Constant>> workspace;
```

2. setVal(RecordId recordId, int offset, Constant val)

   這是我們新增的method，會由RecordPage的setVal來呼叫，transaction會拿到RecordPage所傳來的RecordId以及offset和val。再根據我們上面所新增的資料結構，把從RecordPage拿到的資訊存進workspace中

```
public void setVal(RecordId recordId, int offset, Constant val) {
    if (!workspace.containsKey(recordId)) {
        workspace.put(recordId, new HashMap<Integer, Constant>());
    }
    workspace.get(recordId).put(offset, val);
}
```

3. getVal(RecordId recordId, int offset)

   這是我們新增的method，會由RecordPage中的getVal呼叫，由於現在會先將資訊寫在transaction's private workspace，因此會先需要到transaction中查找有沒有曾經更新過這個record的field的資訊，因此我們會先搜尋transaction's中的workspace有沒有contains這個RecordId，若沒有，我們就return null，若有，再去看這次要get的field有沒有被更新放在workspace，最後return回給RecordPage。

```
public Constant getVal(RecordId recordId, int offset) {
    if (!workspace.containsKey(recordId))
        return null;
    return workspace.get(recordId).get(offset);
}
```

4. commit

   我們現在要在commit原本做的事情前，先把儲存在private workspace中的資訊寫回到buffer中。因此這時候，我們會跑過我們的雙層HashMap，首先，幫每一個有被更新到的RecordId，去跟ConcurrencyMgr要最高層級的c lock後，拿出他修改的field的值，並且請BufferMgr去pin一個buffer，然後請RecoveryMgr根據我們做的操作寫log到 LogFile，接著在pin到的那個buffer中寫回我們的更改的值，最後再Unpin buffer。持續這樣子的動作直到跑完整個 workspace。最後會讓workspace釋放記憶體空間去call worksapce.clear()。

   等到把private workspace中的資訊寫回buffer後，才去call每一個TransactionLifeCycleListener的call back function。

```
public void commit() {
    for (Map.Entry<RecordId, HashMap<Integer, Constant>> entry : workspace.entrySet()) {
        RecordId recordId = entry.getKey();
        concurMgr.realModifyRecord(recordId);
        for (Map.Entry<Integer, Constant> op : entry.getValue().entrySet()) {
            int offset = op.getKey();
            Constant val = op.getValue();

            Buffer buff = bufferMgr.pin(recordId.block());
            LogSeqNum lsn = recoveryMgr.logSetVal(buff, offset, val);
            buff.setVal(offset, val, txNum, lsn);
            bufferMgr.unpin(buff);
        }
    }
    workspace.clear();

    for (TransactionLifecycleListener l : lifecycleListeners)
        l.onTxCommit(this);
```

Experiments(micro benchmark)

Environment

MacBook Pro (13-inch, 2020, Four Thunderbolt 3 ports)	
Processor	2 GHz Quad-Core Intel Core i5
Memory	16 GB 3733 MHz LPDDR4X
Graphics	Intel Iris Plus Graphics 1536 MB

Experiment Result

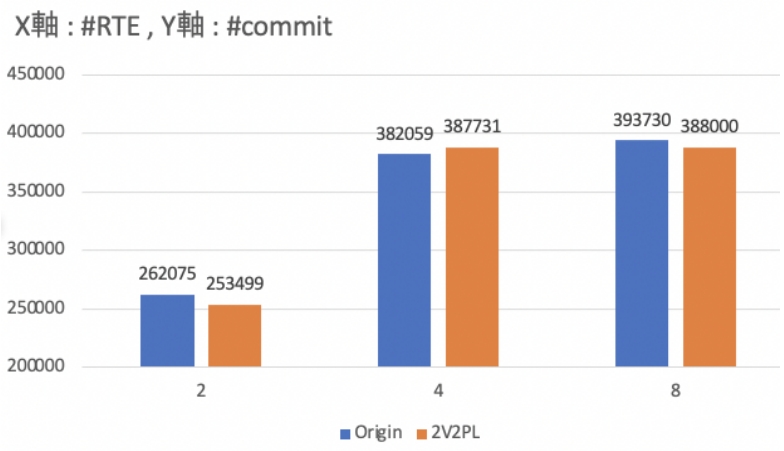
- parameters:（以下的experiment都會以這個基準來調整）

```
RTE=4
RW_TX_RATE=0.2
TOTAL_READ_COUNT=10
LOCAL_HOT_COUNT=1
HOT_CONFLICT_RATE=0.01
BUFFER_POOL_SIZE=102400
```

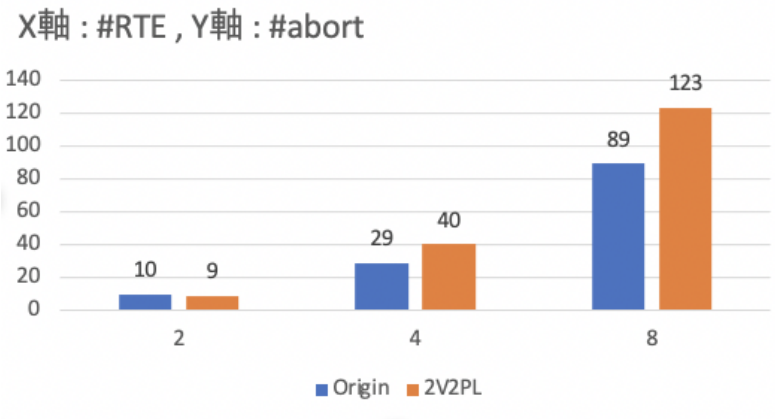
Compare different RTE

我們比對不同的RTE，implement 2V2PL 之前(Origin)/之後(2V2PL)的結果

- Commited



- Aborted



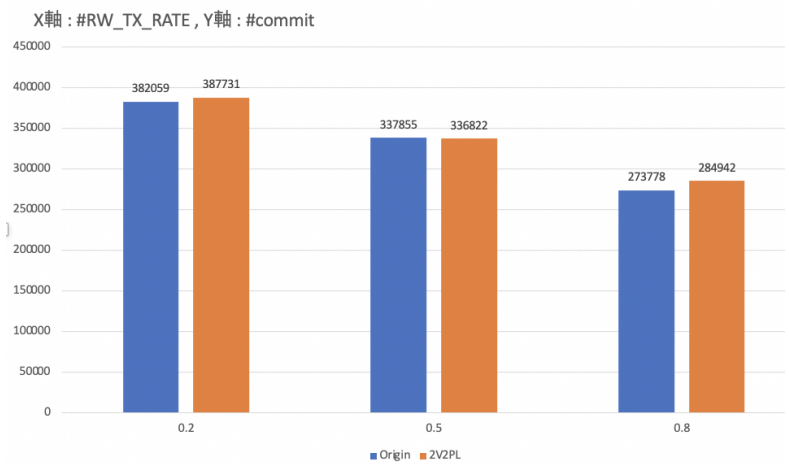
1. 同一版本(皆為implement之前或皆為implement之後)且不同RTE的效果，我們在AS4時已有做過實驗。

2. 比較不同版本，同RTE，我們發現在commit的數量上，相差不多，但在Abort的數量上，我們implement 2V2PL後的結果，似乎普遍變多了，推測可能是因為多了一種lock，也就是這次implement的c_lock，導致transaction遇到incompatibility的可能狀況變多了，而且c_lock被一個transaction拿到時，其他所有transaction都不能對該database有任何動作，導致被abort的機會變多了。
3. 另外可以發現，隨著RTE增加，abort數量的差距也變大，推測是因為有更多的RTE可能會遇到上述第二點的狀況。並且也跟之前也一樣，更多的RTE就會去搶變得相對稀少的資源。

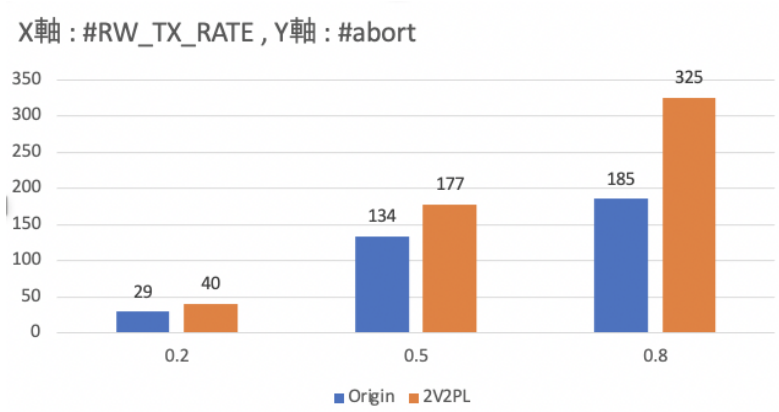
Compare different RW_TX_RATE

我們比對不同的RW_TX_RATE，implement 2V2PL 之前(Origin)/之後(2V2PL)的結果

- Commited



- Aborted



1. 要write to disk時，implement 2V2PL之前的版本由於會拿取x_lock，而implement 2V2PL之後則是會拿取c_lock，導致大家都不行access data，因此當write的transaction

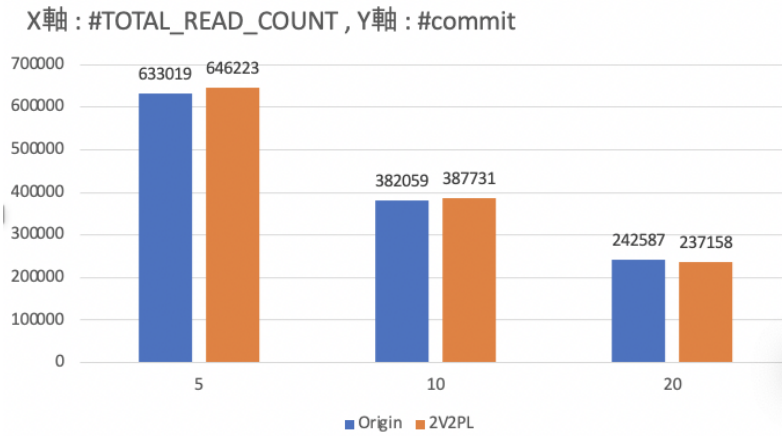
數量增加，大家不能access的時候變多，commit數量自然在兩個版本都減少，且abort數量皆增加了。

2. 而abort在implement 2V2PL之後的版本會較多，推測原因和上述RTE時類似。

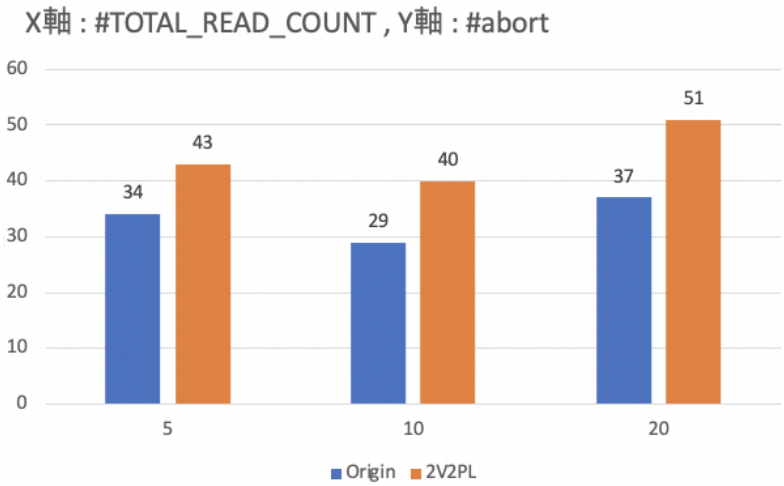
Compare different TOTAL_READ_COUNT

我們比對不同的TOTAL_READ_COUNT，implement 2V2PL 之前(Origin)/之後(2V2PL)的結果

- Commited



- Aborted



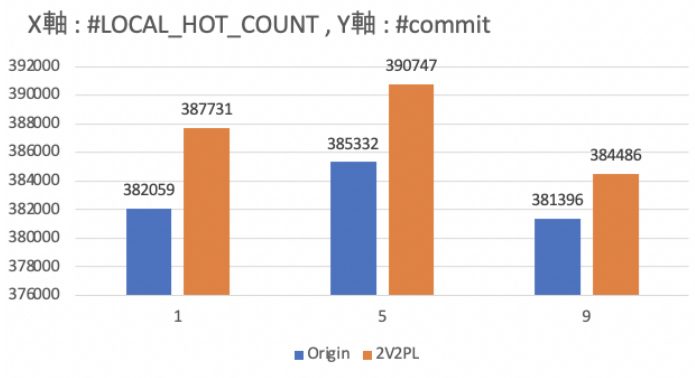
1. 上述結果我們可以看出，比較同一版本不同TOTAL_READ_COUNT，commit數量變少，而abort數量沒有差很多，猜測是因為當一個transaction中，Read的statement越多，代表一個transaction中的statement變多，要完成要花的時間就變多了，而能執行的transaction也變少，所以commit數也變少了。

2. 而同TOTAL_READ_COUNT不同版本，Abort數量較多，推測和RTE狀況類同。

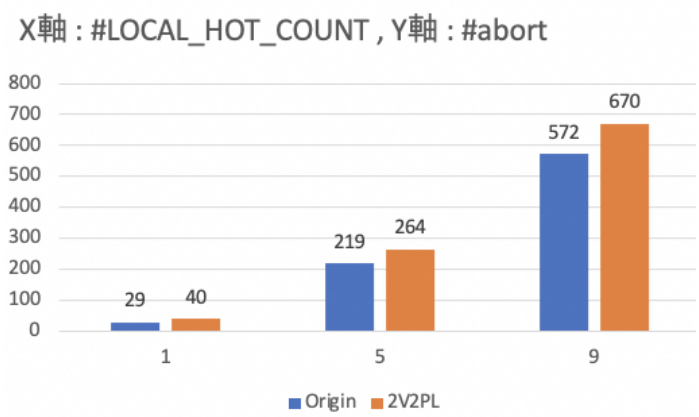
Compare different LOCAL_HOT_COUNT

我們比對不同的LOCAL_HOT_COUNT，implement 2V2PL 之前(Origin)/之後(2V2PL)的結果

- Commited



- Aborted

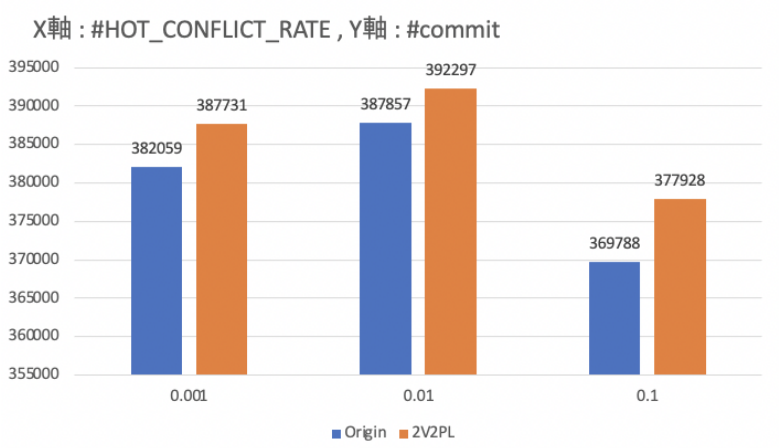


1. 不同LOCAL_HOT_COUNT的影響，在AS4已有討論過。
2. 而在commit數量上，在implement 2V2P後，因為我們會將record拉到local端做處理，因此同一record雖然會被多個transaction access，但和別的transaction發生conflict的狀況就比較不會發生，因此commit成功就會變多。
3. 而abort數量較多的原因推測也和RTE同。

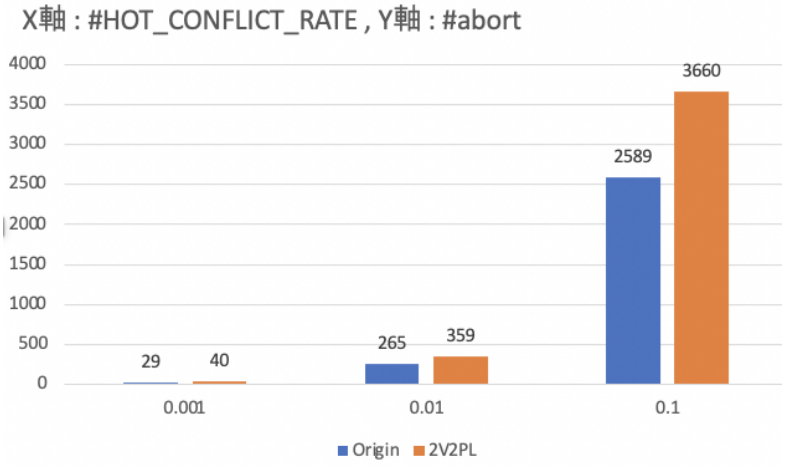
Compare different HOT_CONFLICT_RATE

我們比對不同的HOT_CONFLICT_RATE，implement 2V2PL 之前(Origin)/之後(2V2PL)的結果

- Commited



- Aborted

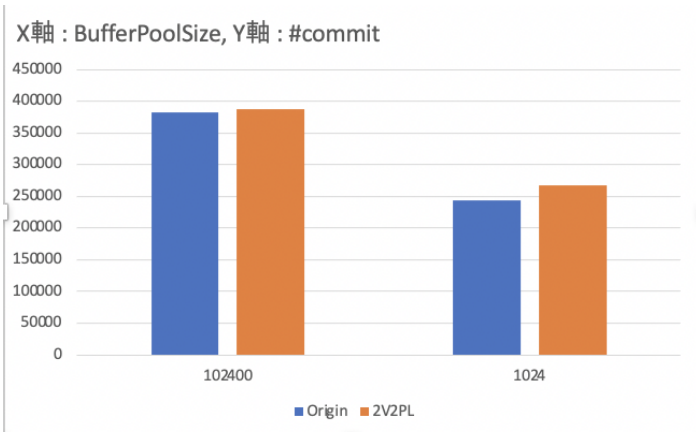


1. 不同的HOT_CONFLICT_RATE在AS4也有討論過了。
2. 同HOT_CONFLICT_RATE且不同版本的比對，和上述LOCAL_HOT_COUNT狀況類似

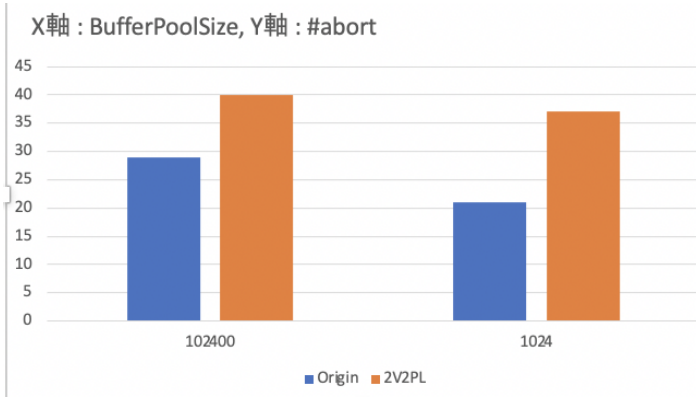
Compare different BufferPoolSize

我們比對不同的BufferPoolSize，implement 2V2PL 之前(Origin)/之後(2V2PL)的結果

- Commited



- Aborted



1. 不同pool size的影響，在AS4已有討論過。
2. 而abort在implement 2V2PL後會較多，推測原因也和上述RTE時類同。