Team5 Assignment4 Report1

107062313 黃寶萱　　107062117 李采蓉　　107062317 陳怡汝

◈ BufferMgr

因為每個 transaction 都會 create 自己的 BufferMgr，而 BufferPoolMgr 在整個系統中是一個 singleton object，因此我們可以藉由縮小 critical section 來優化 BufferMgr，修改 function pin()、pinNew()、unpin()、unpinAll()、repin()、available()、flushAll()、flushAllMyBuffers()中 critical section，只需要在與 BufferPoolMgr 有關的地方(ex. bufferPool.wait(MAX_TIME))做 synchronize 即可，以下以 pin()、flushAll()、flushAllMyBuffers()作為範例解釋。

1. public Buffer pin(BlockId blk)

```
// Pinning process
try {
    Buffer buff;
    long timestamp = System.currentTimeMillis();

    // Try to pin a buffer or the pinned buffer for the given BlockId
    buff = bufferPool.pin(blk);

    // If there is no such buffer or
    // wait for it
    if (buff == null) {
        waitingThreads.add(Thread.currentThread());

        while (buff == null && !waitingTooLong(timestamp)) {
            synchronized (bufferPool) {
                bufferPool.wait(MAX_TIME);
            }
            if (waitingThreads.get(0).equals(Thread.currentThread()))
                buff = bufferPool.pin(blk);
        }

        waitingThreads.remove(Thread.currentThread());

        // Wake up other waiting threads (after leaving this critical section)
        synchronized (bufferPool) {
            bufferPool.notifyAll();
        }
    }
```

因為在 class BufferPoolMgr 中定義 function pin()為 synchronized，所以在 BufferMgr level 可以不用做 synchronize

2. public void flushAll()、public void flushAllMyBuffers()

```
public void flushAll() {
    bufferPool.flushAll();
}

public void flushAllMyBuffers() {
    for (Buffer buff : buffersToFlush) {
        buff.flush();
    }
}
```

因為在 flushAll()和 flushAllMyBuffers()中的 function call 最後都會呼叫到 Buffer level 的 buff.flush()，可以由 Buffer 控制寫回到 disk 的動作，因此在 BufferMgr 這層可以不用做 synchronize。

◈ BufferPoolMgr

1. 我們只修改了 flushAll()的部分，因為可以同時有多個 thread 呼叫 flushAll()，當執行到 buff.flush()時 Buffer 會控制一次只有一個 thread 可以將 dirty buffer 寫回到 disk，因此在這邊可以不用做 synchronize。

```
void flushAll() {
    for (Buffer buff : bufferPool)
        buff.flush();
}
```

2. 我們保留了其他的 synchronized function (ex. pin()、pinNew()、unpin()、available())為 critical section，因為畢竟 BufferPoolMgr 為 singleton，為了確保系統不會出現錯誤，我們選擇保留原本 synchronized 的寫法。

◈ Buffer

```
private final ReentrantReadWriteLock ReadWrteLock= new ReentrantReadWriteLock();
private final Lock ReadLock = ReadWrteLock.readLock();
private final Lock WriteLock = ReadWrteLock.writeLock();
```

在 Buffer.java 中我們發現並非所有的 function 都只允許一次一個 thread 操作，所以我們以 MRSW(Multi-Reader Single-Writer) lock 的方式新增 ReentrantReadWriteLock 取代放在範圍操作符後、返回型別宣告前的 synchronized，如此可以允許同時多個 threads 讀取某段 critical section，但是只允許同時一個 thread 進行寫入，是一種 synchronize 上的優化。以下分成新增 ReadLock 和 WriteLock 兩個部分解釋：

1. 新增 ReadLock 的 functions
    a. public Constant getVal(int offset, Type type)
    b. public LogSeqNum lastLsn()
    c. public BlockId block()
    d. boolean isPinned()
    e. boolean isModifiedBy(long txNum)
    f. Page getUnderlyingPage()

由於可以允許同時多個讀取操作，故在 function 的一開始 ReadLock 先呼叫 lock()，確認沒有 WriteLock 呼叫過 lock() 後才能獲得讀取的鎖定。之後在 try 中放入 critical section 的 code。最後的 finally 則以 unlock()釋放 ReadLock 的鎖定，無論 try block 是否發生例外，程式一定會執行 finally block，故能保證 ReadLock 的使用正確。

```
ReadLock.lock();
try {
    //critical section;
}
finally {
    ReadLock.unlock();
}
```

2. 新增 WriteLock 的 functions
   a. void setVal(int offset, Constant val)
   b. public void setVal(int offset, Constant val, long txNum, LogSeqNum lsn)
   c. void flush()
   d. void pin()
   e. void unpin()
   f. void assignToBlock(BlockId blk)
   g. void assignToNew(String fileName, PageFormatter fmtr)

由於只允許同時一個寫入操作，故在 function 的一開始 WriteLock 先呼叫 lock()，確認沒有任何 ReadLock 或 WriteLock 呼叫過 lock()後才能獲得寫入的鎖定。之後 try 和 finally 的用法同 ReadLock，在 finally 以 unlock()釋放 WriteLock 的鎖定，可以保證 WriteLock 的使用正確。

```
WriteLock.lock();
try {
    //critical section
}
finally {
    WriteLock.unlock();
}
```

◈ Page

由於一個 page 對應到一個 buffer，在 buffer 中的 critical section 已經有其他 synchronize 的優化，原先打算移除所有 function 的 synchronized。但由於在跑 test case 時，以下兩個 functions 不加入 synchronized 會出現錯誤，推測是因為 test case 只會確認 Page.java 內的 function 使用，在嘗試其他 synchronized 優化後仍會報錯，故只有以下兩個 functions 保留 synchronized，其餘刪除。

1. public synchronized Constant getVal(int offset, Type type)
2. public synchronized void setVal(int offset, Constant val)

◈ FileMgr

這個檔案使用到 Java NIO 的 channel 與 buffer 的功能，根據我們查到的資料，Java NIO 本身就支援 multi-thread 的運作，因此我們認為 fileMgr 的所有 synchronized 都可以刪除。

◈ BlockId

參考提示 Never do it again，我們將 hashCode 定義成 BlockId 的 local 變數，並在 Constructor 中就呼叫 hashCode2() 計算 hashcode，之後在原 hashCode() 中只需要回傳一開始算好的 hashcode 即可。

```
public BlockId(String fileName, long blkNum) {
    this.fileName = fileName;
    this.blkNum = blkNum;
    this.hashCode = hashCode2(this.fileName, this.blkNum);
}
```

```
@Override
public int hashCode() {
    //return toString().hashCode();
    return this.hashCode;
}

public int hashCode2(String fileName, long blkNum) {
    String str = "[file " + fileName + ", block " + blkNum + "]";
    return str.hashCode();
}
```

Experiments

■ Micro-benchmark

A. Environment setting :

Intel Core i5-7200U CPU @ 2.50GHz, 12 GB RAM

B. Parameter setting：

NUM_RTES = 10

RW_TX_RATE = 0.25

TOTAL_READ_COUNT = 10

LOCAL_HOT_COUNT = 1

WRITE_RATIO_IN_RW_TX = 0.5

HOT_CONFLICT_RATE = 0.01

C. Experiment result :

用 Parameter setting 所列的數值為基準，觀察不同優化方式下的結果。另外，Parameter setting 數值的選擇是以已經全部優化後的版本，個別比較不同的數值，並以最能顯示優化效果的數值作為基準，其中以 NUM_RTES、RW_TX_RATE 和 HOT_CONFLICT_RATE 作為示範。

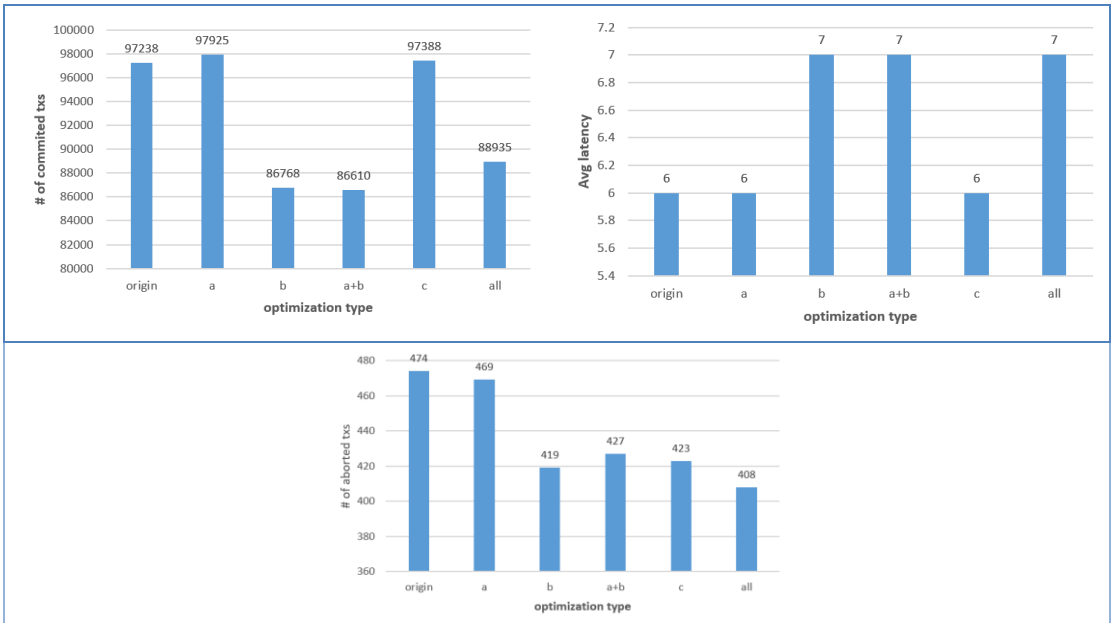
圖表橫軸為不同優化方式：

a.： 只進行 BufferMgr 及 Buffer pool 優化

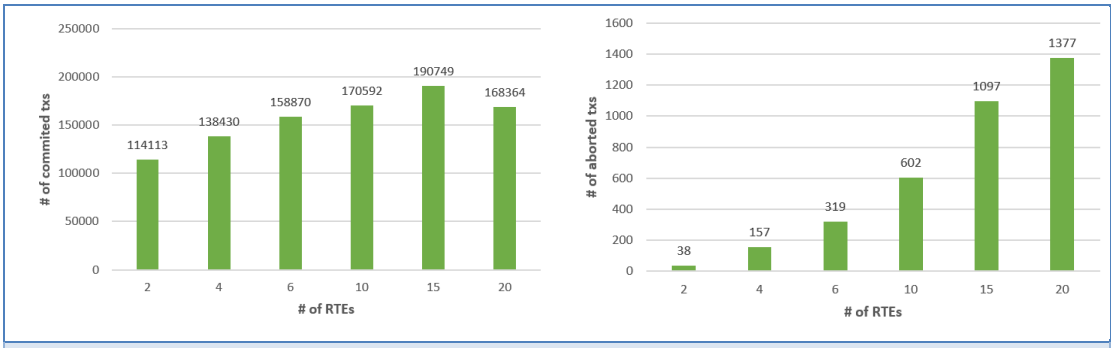
b.： 進行 Buffer read write lock 優化

c.: BlockId 計算方式優化

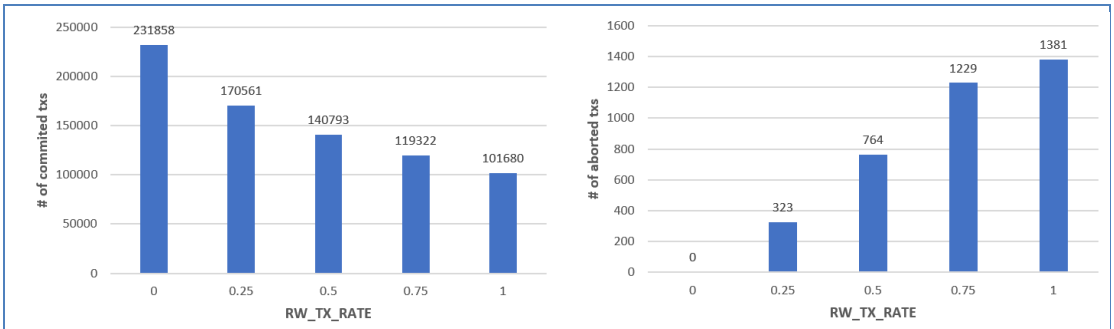
圖表縱軸為 Commited txs 數量、Aborted txs 數量及 Avg latency。



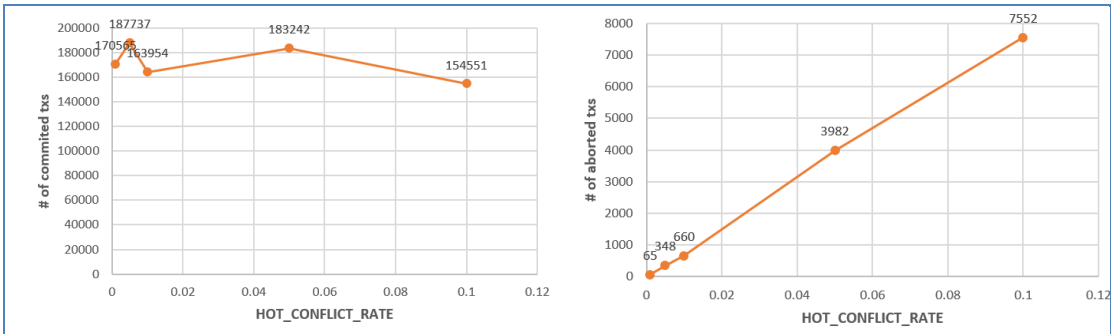
從上圖可以發現，我們只進行 BufferMgr 及 Buffer pool 的優化比原本的提高了 687 個 commited transection 數量；若只進行 BlockId 計算方式優化則比原本的提高了 150 個 commited transection 數量，而兩者的 avg latency 則跟原本差異不大。雖然對個別優化有明顯效果但最終優化的結果反而比原本的 commited transection 數量低、avg latency 也提高，而 aborted transection 數量卻減少。顯示我們選擇的參數雖然是以最終版本在個別比較中的最好，但在 commited transection 數量和 avg latency 卻是個別優化勝於全部優化，全部優化只有在 aborted transection 數量上顯示出其優勢。



從上圖可以發現，NUM_RTES 愈大 commited transection 數量會愈高，但在最高的 20 時因為會因為開始互搶資源稍微降低，而 aborted transection 數量也是隨著 NUM_RTES 增加而提高。最終我們選擇有良好 commited transection 數量以及沒有太高 aborted transection 數量的 NUM_RTES = 10 作為基準。



從上圖可以發現，RW_TX_RATE 愈大 commited transection 數量會愈低，而在 aborted transection 數量是隨著 NUM_RTES 增加而提高，是因為 WRITE 本來就比 READ 繁瑣，愈高機率的 WRITE 表現會降低，但為了保持 READ 和 WRITE 皆有機率出現，故選擇表現稍好的 RW_TX_RATE = 0.25。



HOT_CONFLICT_RATE 我們比較 0.001, 0.01 和 0.1 三種數值，從上圖可以發現，在 0.01 時 commited transection 會最多，而在 aborted transection 數量是隨著 HOT_CONFLICT_RATE 增加而提高。最終我們選擇有良好 commited transection 數量以及沒有太高 aborted transection 數量的 0.01 作為基準。

■ **TPC-C**

A. Environment setting :

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

B. Initial parameters :

NUMVER_WAREHOUSES = 1

BUFFER_POOL_SIZE = 1024

C. Experiment result :

用 Initial parameters 為基準，每次調整一個參數觀察結果。

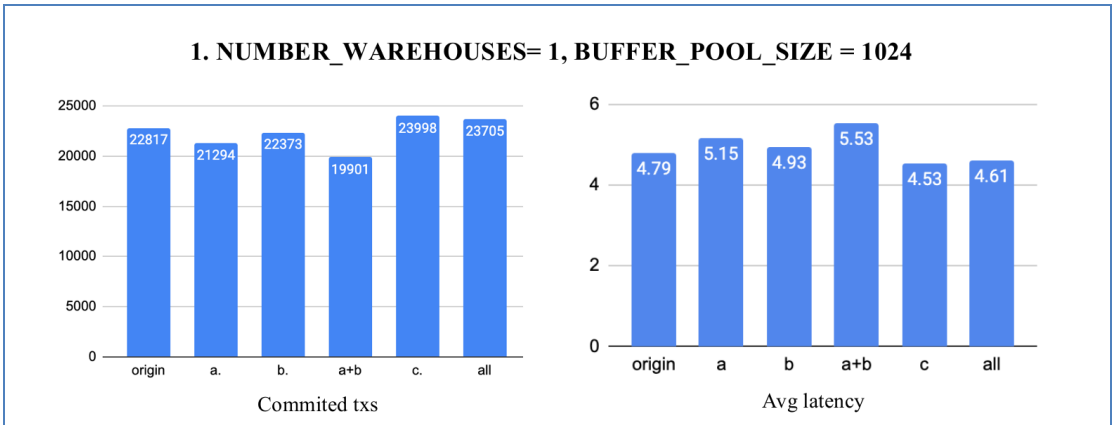
圖表橫軸為不同優化方式：

a.： 只進行 BufferMgr 及 Buffer pool 優化

b.： 進行 Buffer read write lock 優化

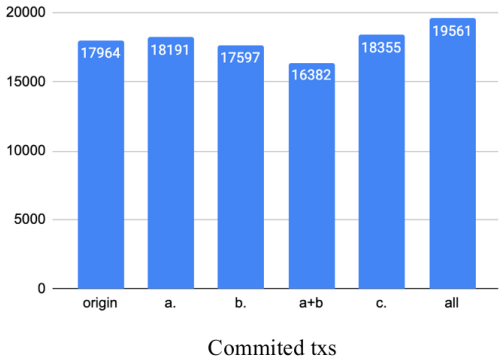
c.: BlockId 計算方式優化

圖表縱軸為 Commited txs 數量及 Avg latency。

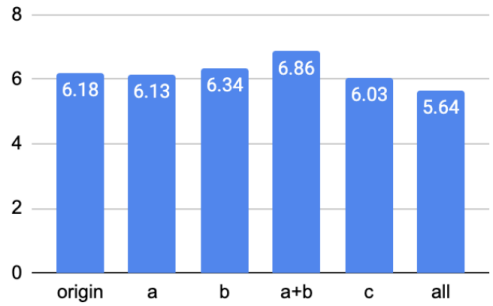


從上圖可以發現，我們最終優化的結果相比原本的提高了 888 個 commited transection 數量，avg latency 降低 0.18ms。

2. NUMBER_WAREHOUSES = 5, BUFFER_POOL_SIZE = 1024



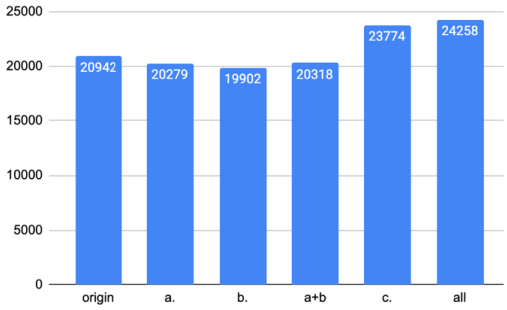
Commited txs



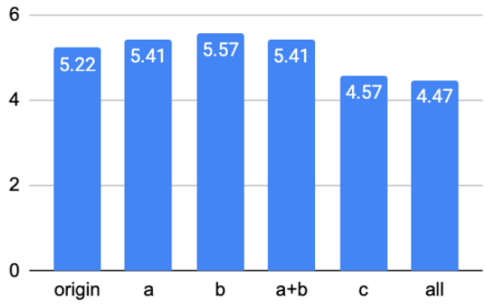
Avg latency

從上圖可以發現，我們最終優化的結果相比原本的提高了 1597 個 commited transection 數量，avg latency 降低 0.54ms。

3. NUMBER_WAREHOUSES= 1, BUFFER_POOL_SIZE = 256



Commited txs



Avg latency

從上圖可以發現，我們最終優化的結果相比原本的提高了 3316 個 commited transection 數量，avg latency 降低 0.75ms。

■ Explanation

我們主要的優化可以分成下列三個部分 ：

1. Smaller critical section : 對 bufferMgr 及 bufferPoolMgr 縮小 critical section 進行優化，並且拿掉其他檔案 Page, fileMgr 中不需要的 synchronized。在這個部分我們透過縮小 critical section 的方式，讓程式有更多地方可以平行處理，而不用等整個 function 結束下一個 thread 才能運行整個 function，因此打到更高的平行度，藉此增加效能。
2. Read write lock : 原本 Buffer 的寫法 read 和 write 是同一個 lock，一次只有一個 thread 可以讀或寫 buffer，但 read buffer 其實可以平行處理，

因此我們 Buffer 進行 read write lock 優化以讓每次有多個 thread 可以同時 read buffer，write buffer 時則是只有一個 thread 可以寫，同時不能有其他 thread 讀 buffer，我們使用 java 的 ReentrantReadWriteLock 來做到這件事，藉此提升效能。

3. Never do it again : 對 blockId 計算 hashCode 的方式進行優化，減少不必要的重複運算，藉此提升效能。

綜合上面的實驗結果，我們發現在 Never do it again 的部分對效能有最好的提升，而其他兩項的效果則不太明顯，在 buffer pool 的地方推測可能是因為我們雖然減少了 critical section 的大小，但是對於大部分的操作都還是有牽涉到 BufferPoolMgr 或是 Buffer 內部的 Synchronize，因此需要等待 lock 釋放次數是差不多的，所以優化效果不顯著。而 Never do it again 直接牽涉到運算次數的優化，因此有較明顯的效果。

首先我們在嘗試 Micro-benchmark 時發現只縮小 critical section 以及 Never do it again 有明顯的優勢，但因為在 ReadWriteLock 的部份表現降低則讓全部優化的版本只有在 aborted 上有優勢，推測可能是不會無效等待 lock 的釋放，故 aborted transaction 數量減少，因此在全部優化的版本上有加成的效果。

而我們在嘗試 TPC-C 時也發現當 WareHouse =5 的時候，縮小 critical section 可以帶來相較 WareHouse = 1 的時候更能帶來效能的提升，推測可能是由於 conflict 次數降低，因此可以更加凸顯出縮小 critical section 可以帶來的幫助。