Assignment4 report

Group members

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

Optimization

Smaller Critical Section

BufferMgr

1. 每一個 transaction 都有一個 BufferMgr，但是所有的 BufferMgr 都共用一個 BufferPool，因此函數中，只需要對 BufferPool 進行 Lock 就好，所以我們把 pin、pinNew、unpin、flushAll、unpinAll、repin 的 critical section 都縮小到有對 BufferPool 進行操作的部分。

   以下只截圖部分程式碼以節省篇幅：

   1. pin：將 method 的 synchronized 移除，放置到以下兩個 crytical section。

```
    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();
    }
  }
```

2. pinNew

```
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.pinNew(fileName, fmtr);
  }

  waitingThreads.remove(Thread.currentThread());

  synchronized(bufferPool) {
    bufferPool.notifyAll();
  }
}
```

3. unpin

```
public void unpin(Buffer buff) {
  BlockId blk = buff.block();
  PinningBuffer pinnedBuff = pinningBuffers.get(blk);
  if (pinnedBuff != null) {
    pinnedBuff.add(-1);
    if (pinnedBuff.pinCount == 0) {
      synchronized (bufferPool) {
        bufferPool.unpin(buff);
        pinningBuffers.remove(blk);
        bufferPool.notifyAll();
      }
    }
  }
}
```

BufferPoolMgr

1. 移除沒有 crytical section 的 method 的 synchronized 詞綴，例如：flushAll、available。

2. 試圖將 pin、pinNew、unpin 的 critical section 縮小，我們認為此三個函數應該在相同的 buff 時有 lock 即可，但是試了一些方法後還是無法成功，因此保留原樣。

Buffer

1. 我們發現 modifiedBy 這個 Set 是不必要的，因為函數 isModifiedBy 只需要知道 Buffer 是否有被修改過，所以直接將 modifiedBy 這個 Set 替換成一個 boolean isModified 來記錄是否有修改過這個 buffer。
2. 我們發現 buffer 可以同時多個 threads 讀，但是有人在寫入時其他 threads 不能讀也不能寫。因此加入讀寫鎖以取代每一個函數的 synchronized 詞綴，這樣可以對寫入操作更加平行。

```
// Buffer allows to 1 write thread & multiple read threads
private final ReadWriteLock internalLock = new ReentrantReadWriteLock();
```

以下只截圖部分程式碼以節省篇幅：

1. getVal → Read Only → Read Lock

```
public Constant getVal(int offset, Type type) {
  internalLock.readLock().lock();
  try {
    return contents.getVal(DATA_START_OFFSET + offset, type);
  } finally {
    internalLock.readLock().unlock();
  }
}
```

2. setVal → Write → Write lock

```
void setVal(int offset, Constant val) {
  internalLock.writeLock().lock();
  try {
    contents.setVal(DATA_START_OFFSET + offset, val);
  } finally {
    internalLock.writeLock().unlock();
  }
}
```

3. pin → Write → Write lock

```
void pin() {
  internalLock.writeLock().lock();
  try {
    ++pins;
  } finally {
    internalLock.writeLock().unlock();
  }
}
```

4. isPinned → Read Only → Read Lock

```
boolean isPinned() {
  internalLock.readLock().lock();
  try {
    return pins > 0;
  } finally {
    internalLock.readLock().unlock();
  }
}
```

Page

1. 因為每一個 Buffer 只有一個 Page 且 Buffer 已經將 crytical section 都處理完，所以我們認為 Page 的所有 method 都不需要 synchronized 詞綴，但是發現 unit testing 中有直接的使

用到 Page 內的函數操作，因此我們保留 setVal 以及 getVal 兩個 methods 的 synchronized。

FileMgr

1. 我們發現，Java NIO 是支援 concurrenty & multi-threading 的操作的，所以對於 FileMgr 的所有操作，其 concurrency 都由底層的 Java NIO channel 來實現，因此可以將所有的 methods 的 synchronized 移除。

Never Do It Again

BlockId

1. 每次執行 BlockId::hashCode() 都必須花費等同字串長度的時間，然而字串本身是不會變的，因此將其 hashcode 在 constructor 時算出並記錄，後面的詢問直接回傳紀錄的 hashcode 即可。

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

```
public int hashCode() {
  return hashcode;
}
```

Replacement strategy

1. 經過討論後，我們認為將現在使用迴圈的 replacement strategy 換成用 queue 並不會比較快，因為 queue 的操作常數較大，可能無法加速太多，尤其是在 bufferPool 不大的時候，因此決定不更改 replacement strategy。

Experiments(micro benchmark)

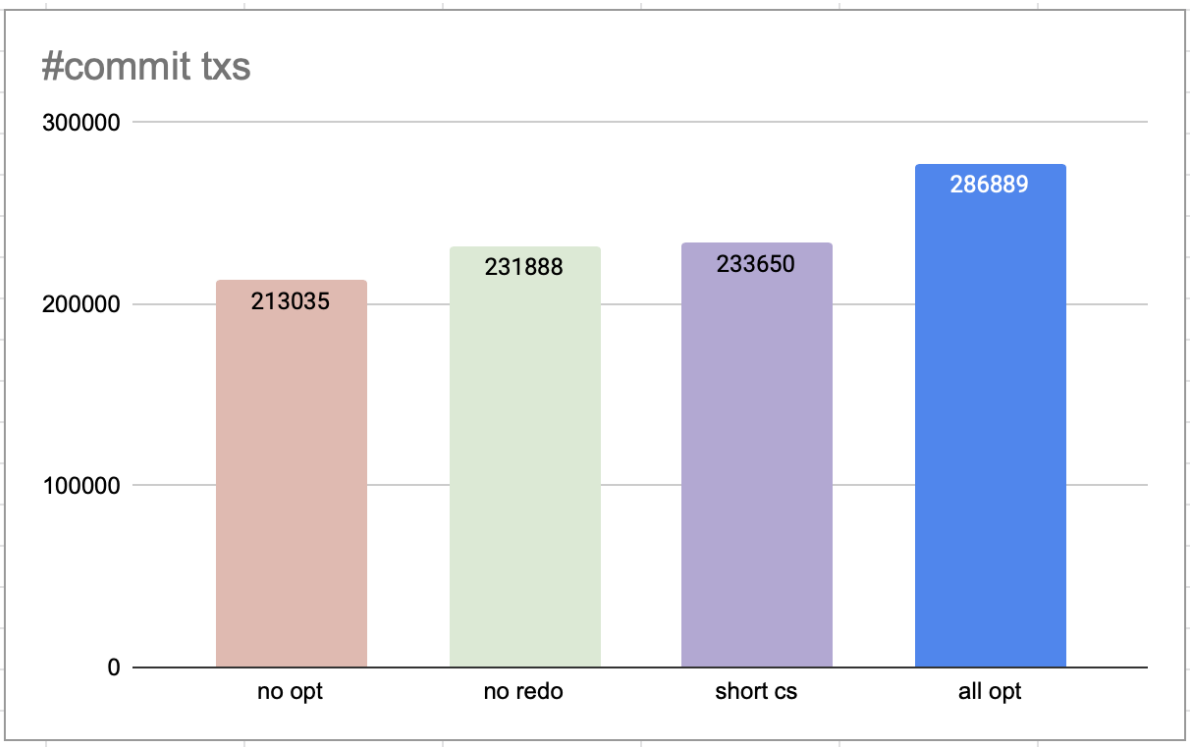
Environment

Intel i5-8365 @ 2.4GHz, 16GB RAM, 512GB SSD, macOS Big Sur 11.2.3

Optimization Results

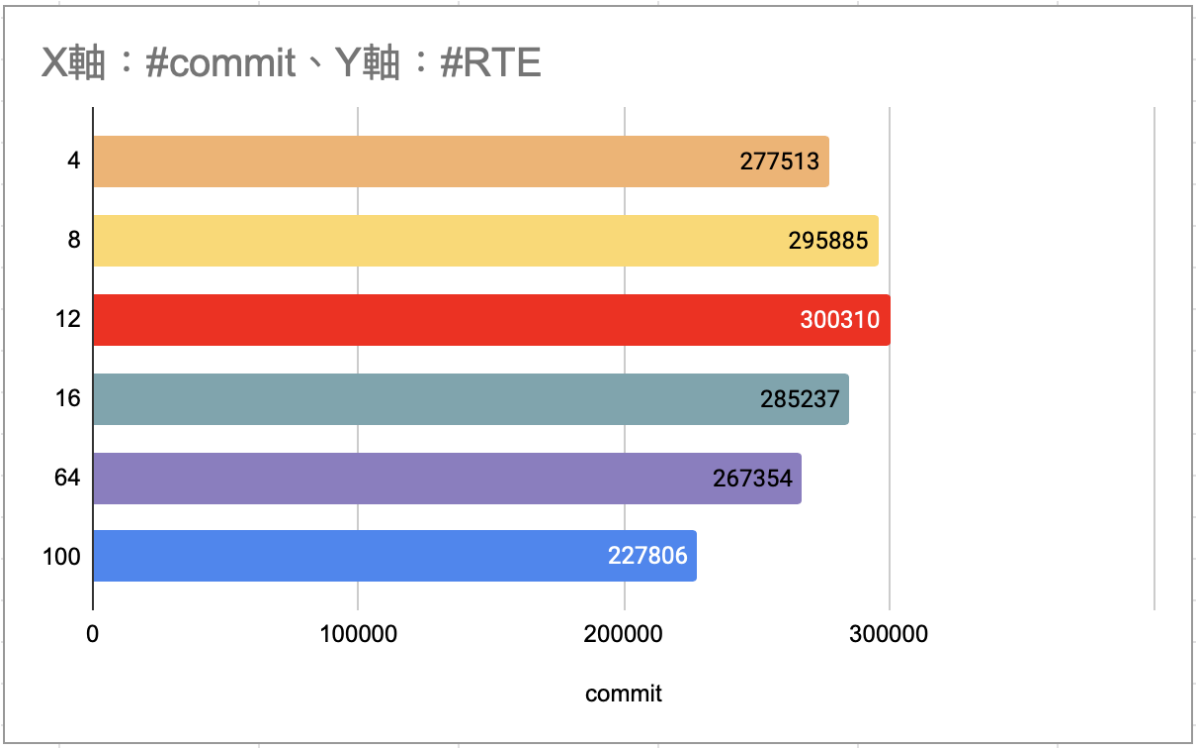
- parameters:（以下的experiment都會以這個基準來調整）

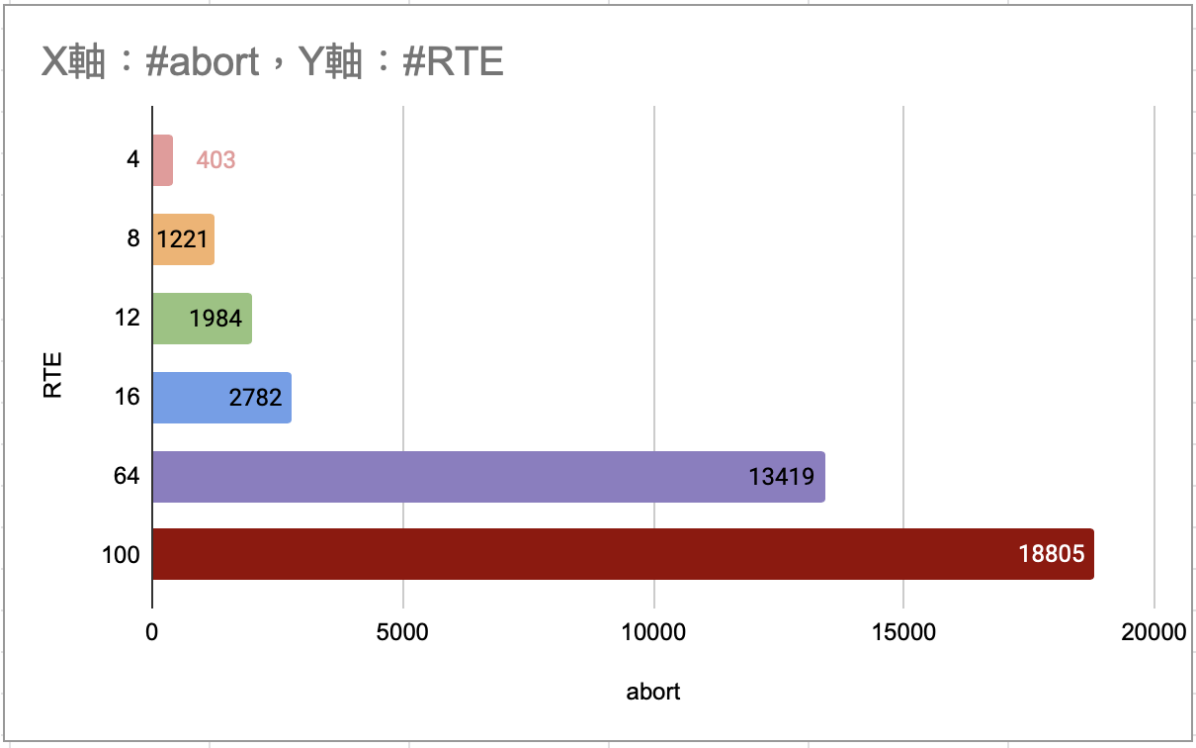
```
RTE=4
BUFFER_POOL_SIZE=102400
NUM_ITEMS=100000
RW_TX_RATE=0.3
LONG_READ_TX_RATE=0.0
TOTAL_READ_COUNT=10
LOCAL_HOT_COUNT=1
WRITE_RATIO_IN_RW_TX=0.5
HOT_CONFLICT_RATE=0.01
```



Compare different RTE

- 我們使用做了所有optimization後的版本，來看不同RTE數量的影響



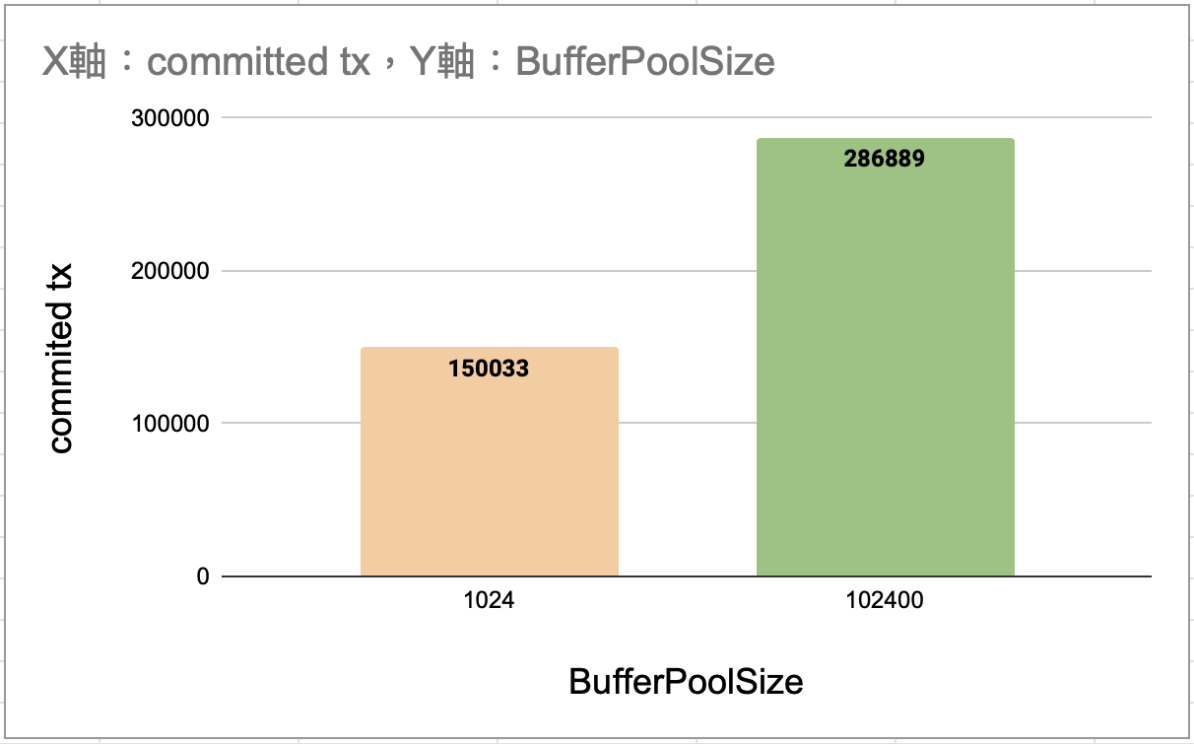


1. 可以看到RTE在12的時候committed transaction的數量到達了最高，但超過12後，我們做的實驗會發現，committed的數量因為RTE之間會互相搶資源，因此反而造成類似thread thrashing的結果

2. 可以看到當RTE數量越高後，被abort的transaction也會越來越多

Compare different BufferPoolSize

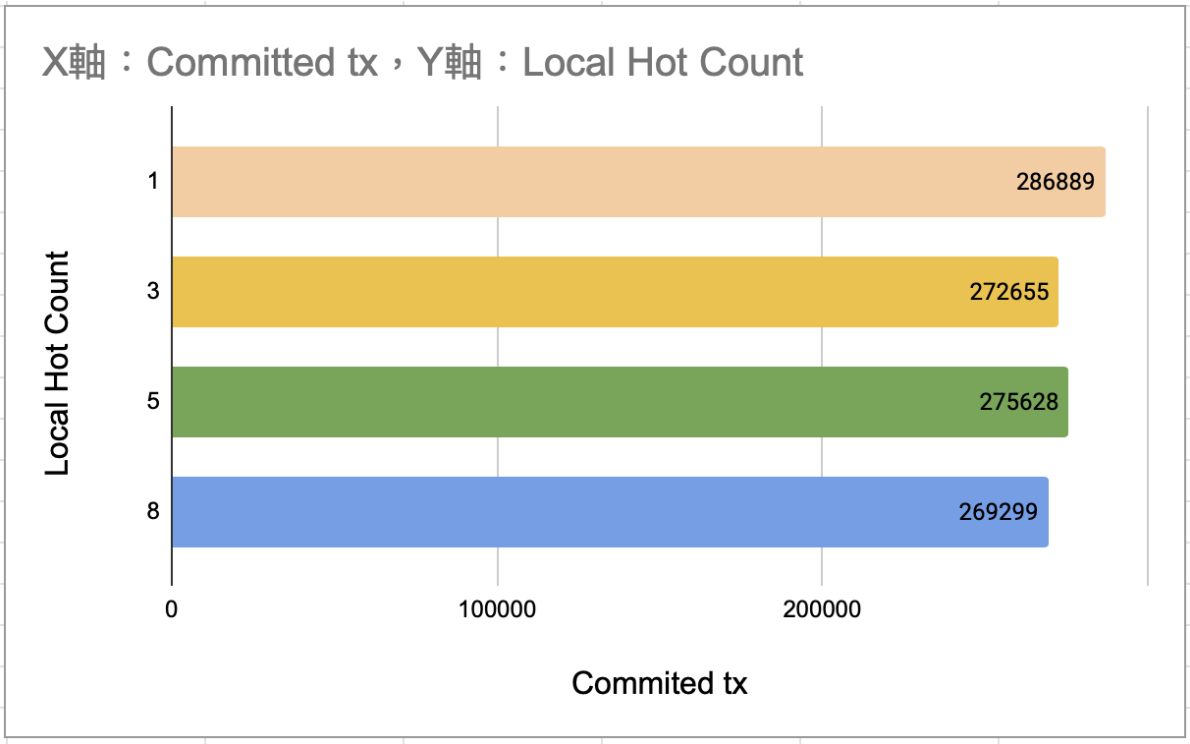
- 我們使用做了所有optimization後的版本，來看不同Buffer Pool Size的影響

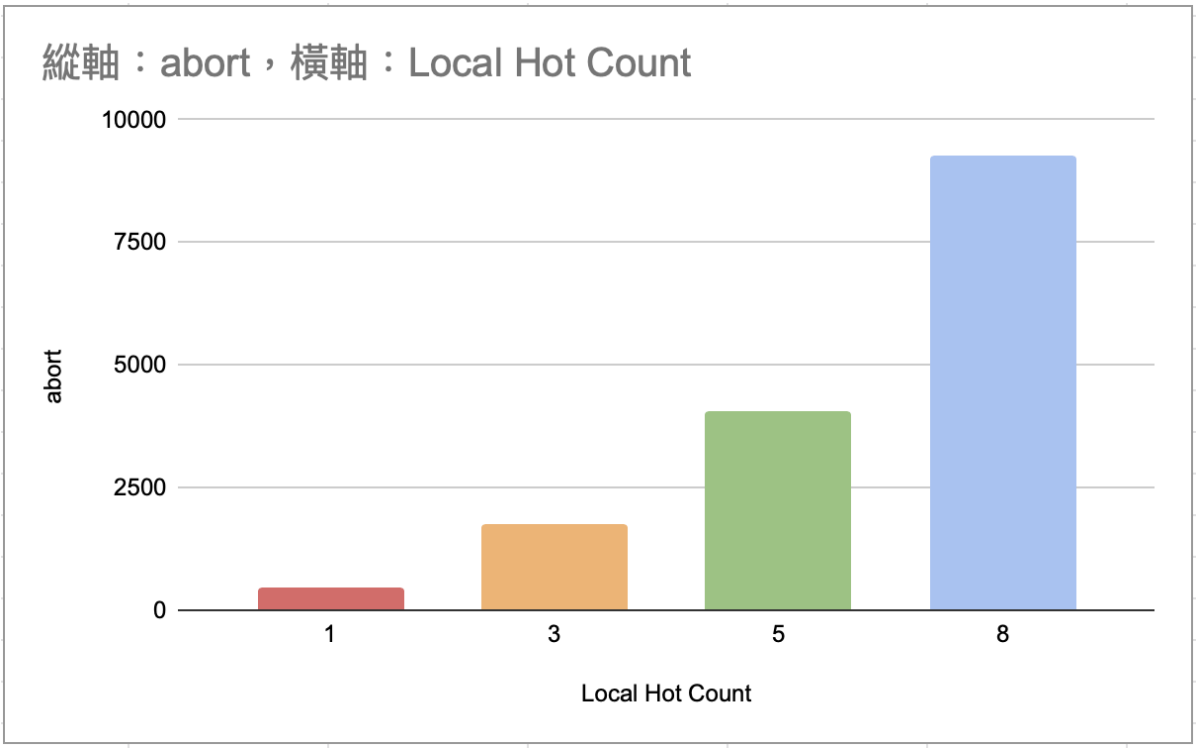


  - 可以發現在BufferPoolSize較大的情況下，commit數可以比較多，throughtput比較大，因為可以cache更多的buffer的話，就可以加速執行query的速度

Compare different Local Hot Count

- 我們使用做了所有optimization後的版本，來看不同Local Hot Count的影響

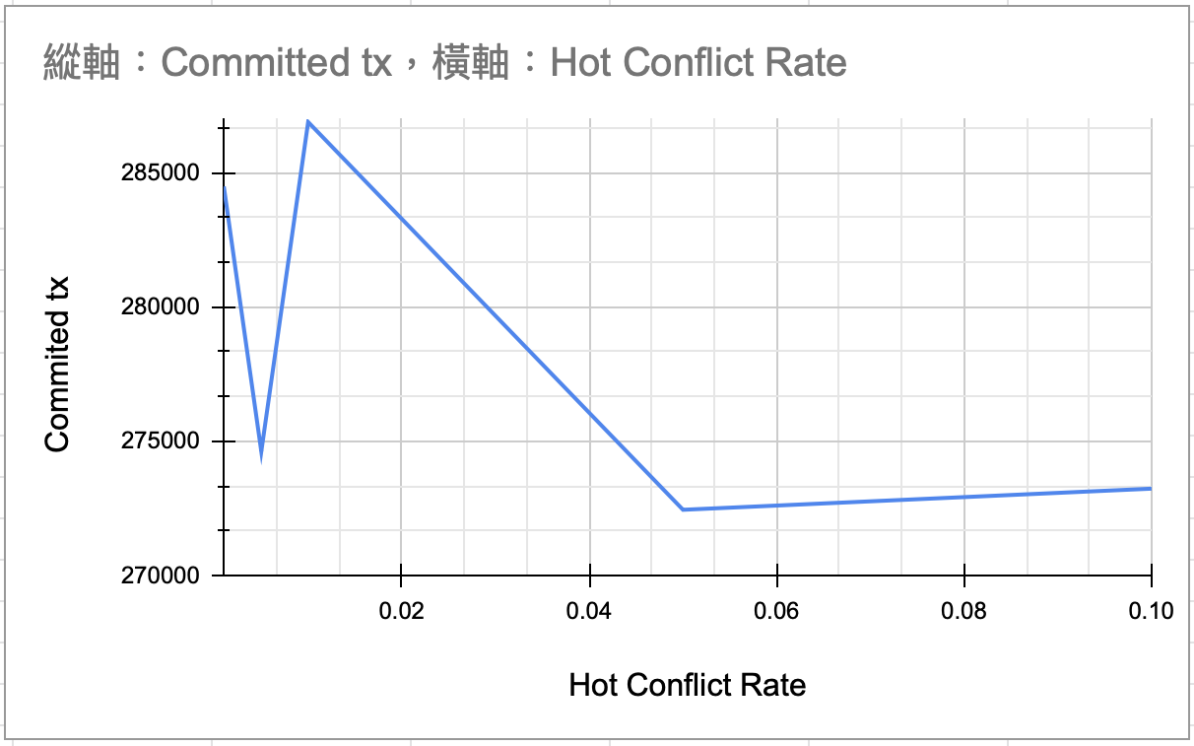


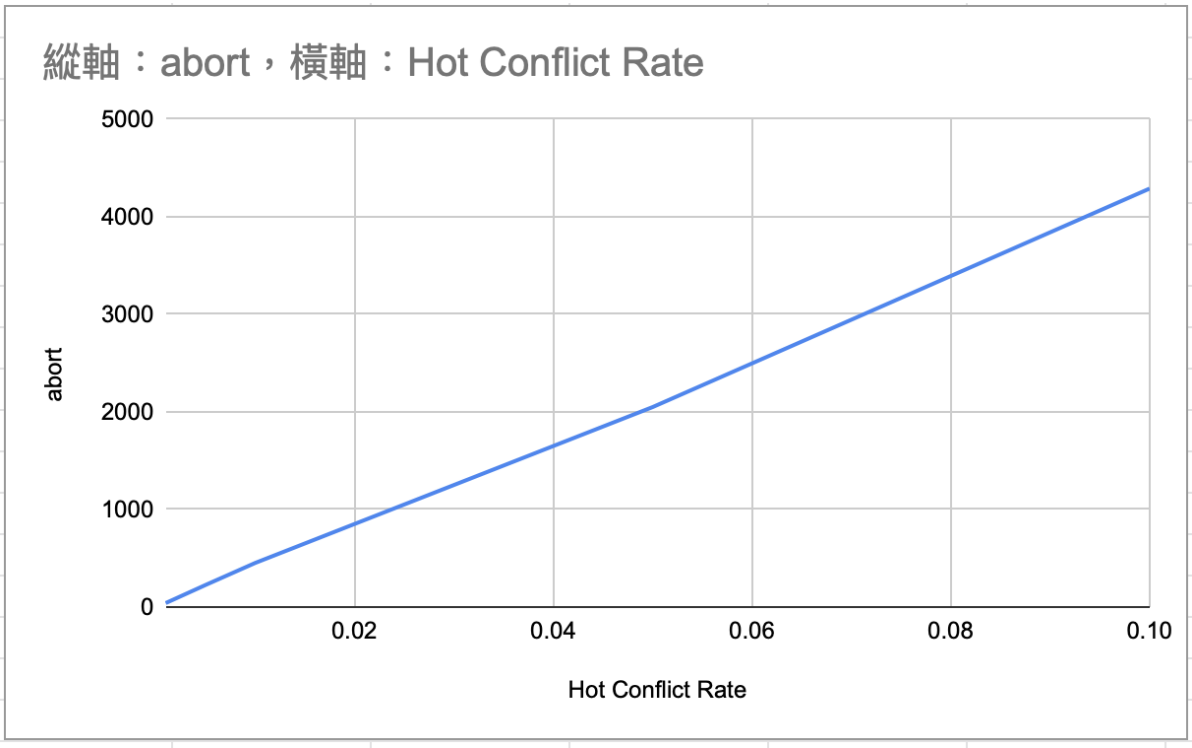


- 可以看到commit數的數量影響沒有很大，但因為從Hot區取的record越來越多，因此會更容易產生conflict，導致為了防止deadlock的aborting transactions

Compare different Hot Conflict Rate

- 我們使用做了所有optimization後的版本，來看不同Hot Conflict Rate的影響





- 這邊跟Local Hot Count也是有異曲同工的感覺，都會增加Conflict的機率，因此也可以看到會導致abort的transaction變多，total committed transactions數，雖然根據每次跑的情況不盡相同，但整體來說也是下降了大約一萬。
- 可以從折線圖看出abort數與Hot Conflict Rate彼此之間呈正相關

Experiments(TPCC benchmark)

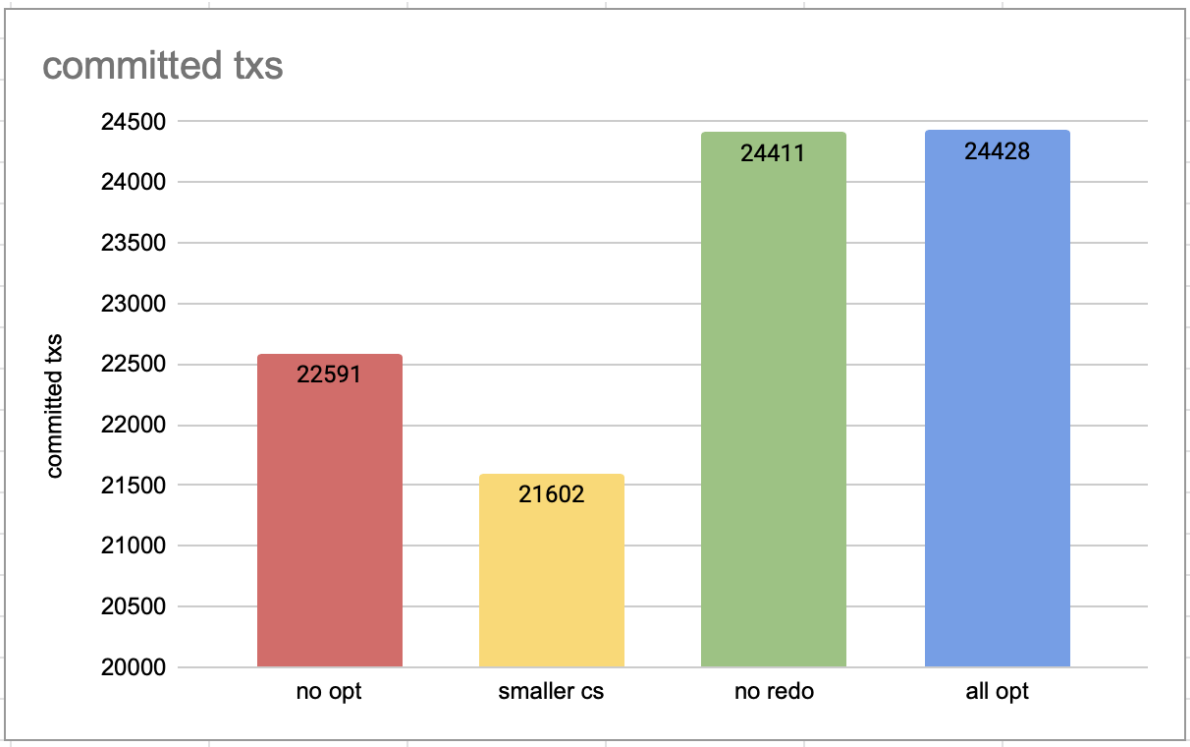
Environment

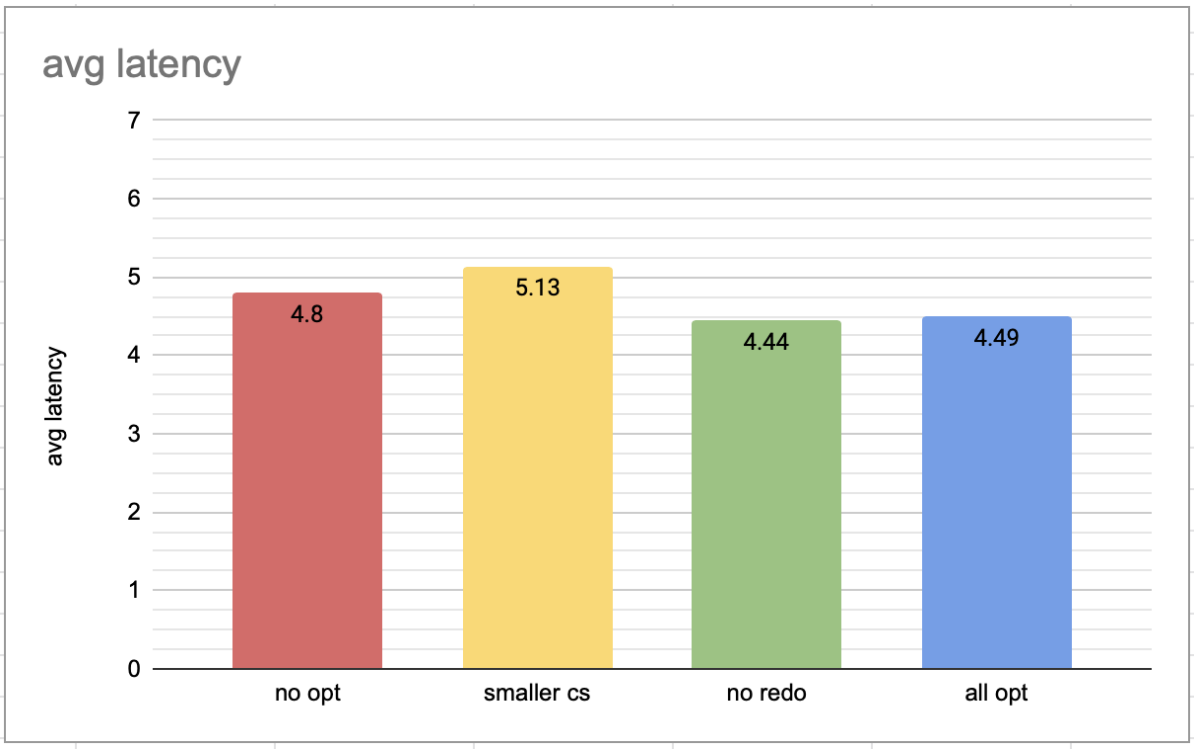
Intel i5-8365 @ 2.4GHz, 16GB RAM, 512GB SSD, macOS Big Sur 11.2.3

Optimization Results

- parameters:

```
RTE=2
BUFFER_POOL_SIZE=102400
NUM_WAREHOUSE=1
```

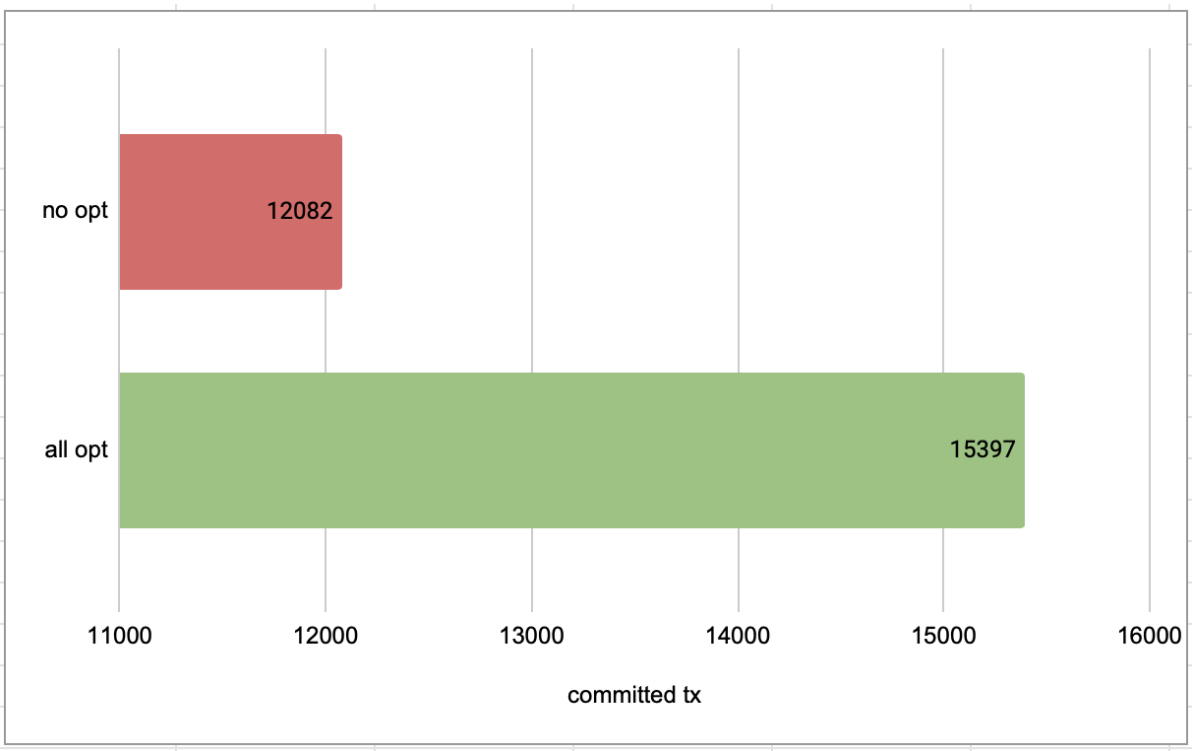




- 可以看到整體來說throughput是進步了大約2000個committed txs
- 而average latency在最終優化版本也有下降

Compare different Num Warehouse

- parameters: buffer pool size = 1024
- warehouse = 5



- 可以看出我們在warehouse是5的時候，優化的效果是較為明顯的，猜測原因是warehouse變多，conflict機率下降，並且綜合上面warehouse在 improve hash code那部份的效果會更明顯，少了一些conflict的話，就不會拖住hash code那邊優化出來的數量。

Discuss Why Our Optimization works

總結來說，我們主要做了兩種優化。

第一個是縮減synchronized涵蓋的範圍。

原本的code(上述no opt)中幾乎各個地方都掛上了synchronized的keyword，如此一來，運作上雖然安全，但是卻很冗贅，因為並非所有synchronized的地方，都有用到share resource，因此移除非必要的synchronized，以及縮小synchronized包起來的區塊，盡可能讓critical section變小，讓可以concurrency處理的地方變多，自然效能就會增加。

另外，在read和write的部分，原本的code(上述no opt)，一次只能有一個thread做read，但read其實是可以平行處理的，因此我們使用read / write lock，使得read可以讓多個thread同時進行，而保持write時，只有writer thread可以進行。而這項優化會在read transaction較多時，比較顯著。

第二個是防止不必要的重複計算。

如上Never Do It Again所述，一個BlockID的hashcode是不變的。

而我們使用eclipse中的call hierarchy的功能發現，在許多地方都有呼叫到BlockID的hashcode() function，若相較原本的code(上述no opt)，藉由我們的改良方法，對於同一BlockID的object，其hashcode計算，從原本的數次，降到只需一次，去除了多餘的計算hashcode時間，效能自然也會有所提升。