Team5 Final Project Report

107062313 黃寶萱　107062117 李采蓉　107062317 陳怡汝

Collect features

共 5 個 features : Read Count、Write Count、Field Size、And Size、Insert Size、TotalWaitTime

- Feature1 : Read Count

  因為使用 Conservative ConcurrencyMgr 在 Tx 執行前會將需要拿的 read lock 分別存在 `readSet` 中，因此我們可以在根據 readSet 的大小得知該 Tx read count 數量，若 read count 數量越多代表該 Tx 需要讀取較多的資料，因此會影響 Tx 的 latency。

```
readCount = readSet.size();
writeCount = writeSet.size();
```

- Feature2 : Write Count

  同理，該 Tx 需要拿的 write lock 都記錄在 writeSet 中，因此可以藉由 `writeSet` 大小取得 write count，若 write count 數量越多代表該 Tx 需要執行較多的 write operation，因此會影響 Tx 的 latency。

- Feature3 : Field Size

  在 NewOrderProc 與 PaymentProc 的 `prepareKeys()` 中我們利用變數 `fieldSize` 紀錄該 Tx 執行所有 query 總共會 SELECT 幾個 column/field，因為我們認為保留的 column 數量多寡可能會影響後續運算時需要查看的 table size 大小以及資料傳遞的 loading，因而影響到 Tx 的 latency。

```
// SELECT ... FROM warehouse WHERE w_id = wid
keyEntryMap = new HashMap<String, Constant>();
keyEntryMap.put("w_id", widCon);
warehouseKey = new PrimaryKey("warehouse", keyEntryMap);
readSet.add(warehouseKey);
fieldSize += 1;
andSize += 0;

// SELECT ... FROM district WHERE d_w_id = wid AND d_id = did
keyEntryMap = new HashMap<String, Constant>();
keyEntryMap.put("d_w_id", widCon);
keyEntryMap.put("d_id", didCon);
districtKey = new PrimaryKey("district", keyEntryMap);
readSet.add(districtKey);
fieldSize += 2;
andSize += 1;
```

- Feature4 : And Size

  同樣在 `prepareKeys()` 中我們利用變數 `andSize` 紀錄該 Tx 執行 query 中的所有 WHERE 條件數量，因為 query 中 WHERE 設立的條件越多，Scan 在 go through table 時很有可能會根據條件需要 access 較多 column/field 的 value，我們認為這樣也有可能使 Tx 花費較多時間在處理條件判斷，因而影響 Tx 的 latency。

```
// SELECT ... FROM warehouse WHERE w_id = wid
keyEntryMap = new HashMap<String, Constant>();
keyEntryMap.put("w_id", widCon);
warehouseKey = new PrimaryKey("warehouse", keyEntryMap);
readSet.add(warehouseKey);
fieldSize += 1;
andSize += 0;

// SELECT ... FROM district WHERE d_w_id = wid AND d_id = did
keyEntryMap = new HashMap<String, Constant>();
keyEntryMap.put("d_w_id", widCon);
keyEntryMap.put("d_id", didCon);
districtKey = new PrimaryKey("district", keyEntryMap);
readSet.add(districtKey);
fieldSize += 2;
andSize += 1;
```

- Feature5 : Insert Size

  此外，我們也利用變數 `insertSize` 記錄該 Tx 共執行多少次 INSERT SQL 指令，因為我們認為 INSERT query 是加入一整個 tuple，會影響到 table 中的所有 column，因此會花費較多的 cost，若該 Tx 執行較多 INSERT query 理論上它的 latency 也會較長。

```
// INSERT INTO order_line (ol_o_id, ol_w_id, ol_d_id, ol_number, ...)
// VALUES (nextOId, wid, did, i, ...)
keyEntryMap = new HashMap<String, Constant>();
keyEntryMap.put("ol_o_id", oidCon);
keyEntryMap.put("ol_d_id", didCon);
keyEntryMap.put("ol_w_id", widCon);
keyEntryMap.put("ol_number", olNumCon);
orderLineKeys[i][2] = new PrimaryKey("order_line", keyEntryMap);
writeSet.add(orderLineKeys[i][2]);
fieldSize += 10;
insertSize += 1;
```

- Feature6 : TotalWaitTime

  我們覺得 Tx 在拿取 read/write lock 需要等待的時間長短也會影響 latency，因此我們在 `bookReadKeys(readSet)`與 `bookWriteKeys(writeKey)`之前用變數 `BeforeWaitTime` 紀錄當時的時間，拿完 key 之後也用變數 `AfterWaitTime` 記錄時間，便可以得到該 Tx 再拿取 read/write lock 所花的時間，即 `TotalWaitTime`。

```
BeforeWaitTime = System.nanoTime();
ccMgr.bookReadKeys(readSet);
ccMgr.bookWriteKeys(writeSet);
AfterWaitTime = System.nanoTime();
TotalWaitTime = AfterWaitTime - BeforeWaitTime;
```

Output Collected Features

- **StoredProcedure**

  Stored Procedure 進到 `execute()`執行完 SQL 指令後，在 Tx commit 時呼叫上述每個 feature 的 function 取得該 Tx 對應的 5 個 feature data，並包在一個 `SpResultSet` 中 return 回去。

```
try {
    ConservativeConcurrencyMgr ccMgr = (ConservativeConcurrencyMgr) tx.concurrencyMgr();

    // Acquire locks before execution
    ccMgr.acquireBookedLocks();

    executeSql();

    // The transaction finishes normally
    tx.commit();
    //ActiveTxs -= 1;

    isCommitted = true;
    readCount = readSet.size();
    writeCount = writeSet.size();
    txNum = tx.getTransactionNumber();
    myBlks = paramHelper.getBlks();
    myRecs += paramHelper.getRecords();
    fieldSize += paramHelper.getFieldSize();
    andSize += paramHelper.getAndSize();
    insertSize += paramHelper.getInsertSize();
    myLog = VanillaDb.logMgr().getLog();
    maxBufferSize = tx.bufferMgr().getMaxBufferSize();
    hitRate = tx.bufferMgr().getHitRate();
    TotalWaitTime = AfterWaitTime - BeforeWaitTime;
```

```
return new SpResultSet(
    isCommitted,
    txNum,
    readCount,
    writeCount,
    myRecs,
    myLog,
    maxBufferSize,
    hitRate,
    fieldSize,
    andSize,
    insertSize,
    ActiveTxs,
    TotalWaitTime,
    paramHelper.getResultSetSchema(),
    paramHelper.newResultSetRecord()
);
```

- **SpResultSet**

  在 SpResultSet 裡面我們新增 feature 相關的變數，以及取得 feature data 的 function，例如：`getReadCount()`、`getWriteCount()`等。

```
public SpResultSet(boolean isCommitted, long txNum, long readCount, long writeCount, long myRecs, long myLog,
                   long maxBufferSize, double hitRate, long fieldSize, long andSize, long insertSize, long activeTxs,
                   long TotalWaitTime, Schema schema, Record... records) {
    this.isCommitted = isCommitted;
    this.records = records;
    this.schema = schema;
    this.readCount = readCount;
    this.writeCount = writeCount;
    this.txNum = txNum;
    this.myRecs = myRecs;
    this.myLog = myLog;
    this.maxBufferSize = maxBufferSize;
    this.hitRate = hitRate;
    this.fieldSize = fieldSize;
    this.andSize = andSize;
    this.insertSize = insertSize;
    this.activeTxs = activeTxs;
    this.TotalWaitTime = TotalWaitTime;
}
```

```
public long getReadCount() {
    return readCount;
}

public long getWriteCount() {
    return writeCount;
}

public long getTxNum() {
    return txNum;
}
```

- **StatisticMgr**

  我們用一個 Class `MyStaticResult` 來存 Tx 的 feature，並新增一個 Hashmap `myMap`，當 Tx commit 時將該 Tx 的 `txNum` 以及對應的 feature data 存到 `myMap` 中。

```
class MyStaticResult{
    long readCount = 0;
    long writeCount = 0;
    long startTime = 0;
    long responseTime = 0;
    long fieldSize = 0;
    long andSize = 0;
    long insertSize = 0;
    long TotalWaitTime = 0;
    MyStaticResult(long readCount, long writeCount, long startTime, long responseTime,
                   long fieldSize, long andSize, long insertSize,long TotalWaitTime)
    {
        this.readCount = readCount;
        this.writeCount = writeCount;
        this.startTime = startTime;
        this.responseTime = responseTime;
        this.fieldSize = fieldSize;
        this.andSize = andSize;
        this.insertSize = insertSize;
        this.TotalWaitTime = TotalWaitTime;
    }
}
```

在 `analyzeResultSet()` 中將 Tx 的 features 加入 `myMap`。

```
if (rs.isTxnIsCommited()) {
    Long startTime = rs.getTxnEndTime() - rs.getTxnResponseTime();
    Long responseTime = rs.getTxnResponseTime();
    myMap.put(rs.getTxNum(), new MyStaticResult(rs.getReadCount(), rs.getWriteCount(), startTime, responseTime,
            rs.getRecs() ,rs.getLog(), rs.getMaxBufferSize(), rs.getHitRate(), rs.getFieldSize(),
            rs.getAndSize(), rs.getInsertSize(), rs.getActiveTxs(), rs.getTotalWaitTime()));
```

並在 `outputFeatureReport()` 產生 output feature.csv file。

```
private void outputFeatureReport(String fileName) throws IOException {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(new File(outputDir, "feature_test.csv")))) {
        writer.write(
                "Transaction ID,Start Time,Read Count,Write Count,Field Size,And Size,Insert Size,Total Wait Time");
        writer.newLine();
        // logger.info("=== " + myMap.size() + " ===");
        for(Map.Entry<Long, MyStaticResult> item : myMap.entrySet()) {
            long id = item.getKey();
            MyStaticResult value = item.getValue();
            long read = value.readCount;
            long write = value.writeCount;
            long start = value.startTime / 1000000;
            long fieldSize = value.fieldSize;
            long andSize = value.andSize;
            long insertSize = value.insertSize;
            long TotalWaitTime = value.TotalWaitTime;
            String str = id + "," + start + "," + read + "," + write + "," + fieldSize + ","
                    + andSize + "," + insertSize + "," + TotalWaitTime;

            writer.write(str);
            writer.newLine();
        }
    }
}
```

在 `outputLatencyReport()` 產生 output latency.csv file。

```
private void outputLatencyReport(String fileName) throws IOException {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(new File(outputDir, "latency_test.csv")))) {
        writer.write(
                "Transaction ID,Latency");
        writer.newLine();

        for(Map.Entry<Long, MyStaticResult> item : myMap.entrySet()) {
            long id = item.getKey();
            MyStaticResult value = item.getValue();
            long response = value.responseTime / 1000000;
            String str = id + "," + response;

            writer.write(str);
            writer.newLine();
        }
    }
}
```

- **Other details**

在 bench 端我們對 `SutResultSet` 新增對應 SpResultSet 的 function 以取得 SpResultSet 的資料，並用 `VanillaDbSpResultSet` implement SutResultSet 中的 function，最終 feature 在 `TpccTxExecutor` 中透過 `TxResultSet` 包回給 StatisticMgr 進行輸出。

Collect Latency

- TpccTxExecutor

  在 TpccTxExecutor 中我們計算 Tx 從開始執行到結束的時間作為 Tx 的 latency，並且包成 `TxResultSet` 回傳給 StatisticMgr 進行輸出。

```
// send txn request and start measure txn response time
long txnRT = System.nanoTime();

SutResultSet result = executeTxn(conn, params);

// measure txn Sresponse time
long txnEndTime = System.nanoTime();
txnRT = txnEndTime - txnRT;
```

Model

- Model type

  我們嘗試了 Random Forest 及 Linear Regression 這兩種模型，並且比較在這兩種模型上預測的結果。

- Data preprocess

  因為我們搜集的資料在數值上跨度極大（如下表），我們認為這有可能會影響到模型預測的結果，因此我們對每種資料分別進行的 normalize 的動作，將資料 normalize 到 0 到 1 之間，並比較做與不做 normalize 的預測結果差異。

Features	Data range
Read/Write Count	(0~33)
Field Size	(0~333)
And Size	(0~34)
Insert Size	(0~15)
Total Wait Time	(0~519618)

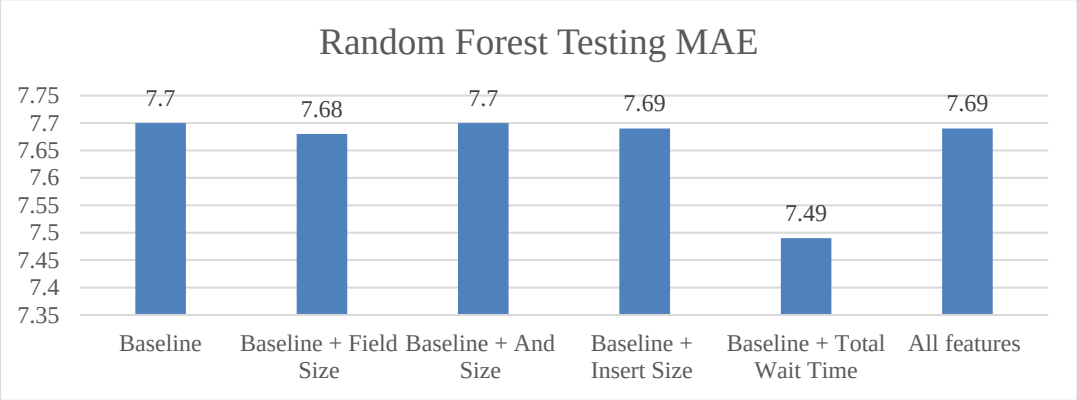
Experiments

- Environment setting : Intel Core i7-8565U CPU @ 1.80GHz 1.99 GHz, 16.0 GB RAM
- Parameter setting : BUFFER_POOL_SIZE=400000
- Benchmark : TPC-C
- Experiment result

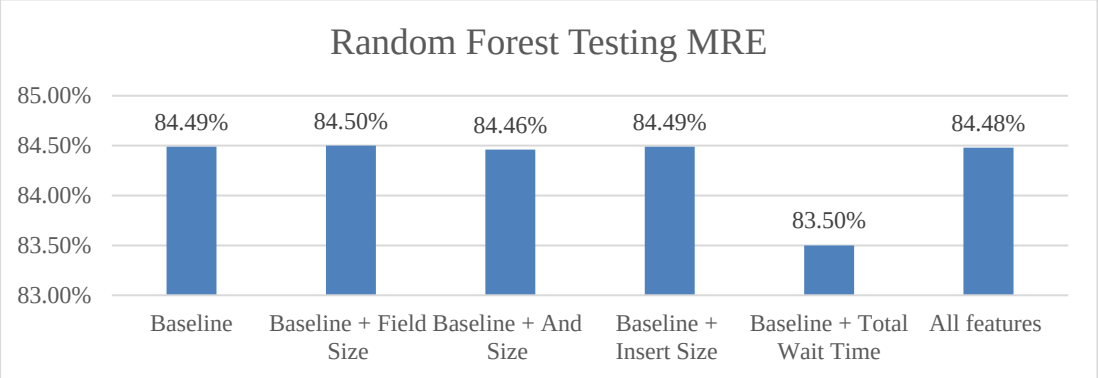
我們用各項新增加的 feature 加上原來的 baseline 對兩個 model 的結果做比較。

A. Random Forest

1. MAE



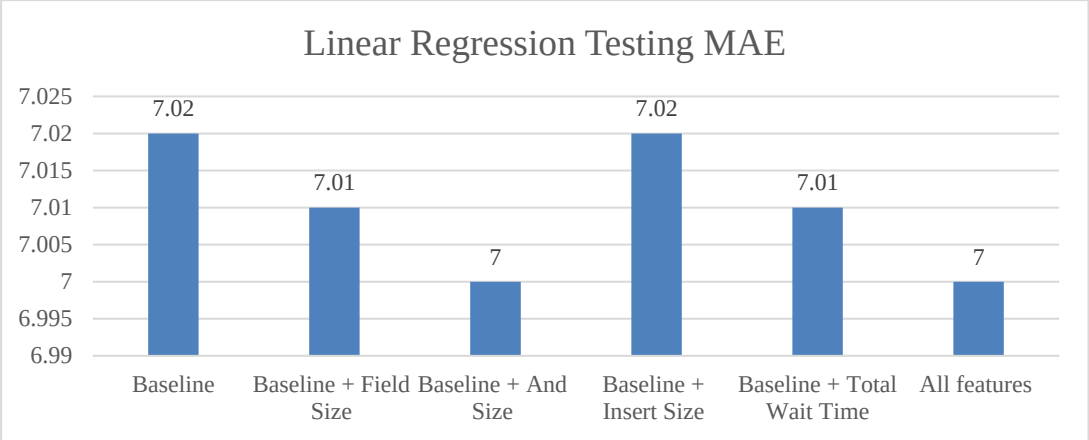
2. MRE



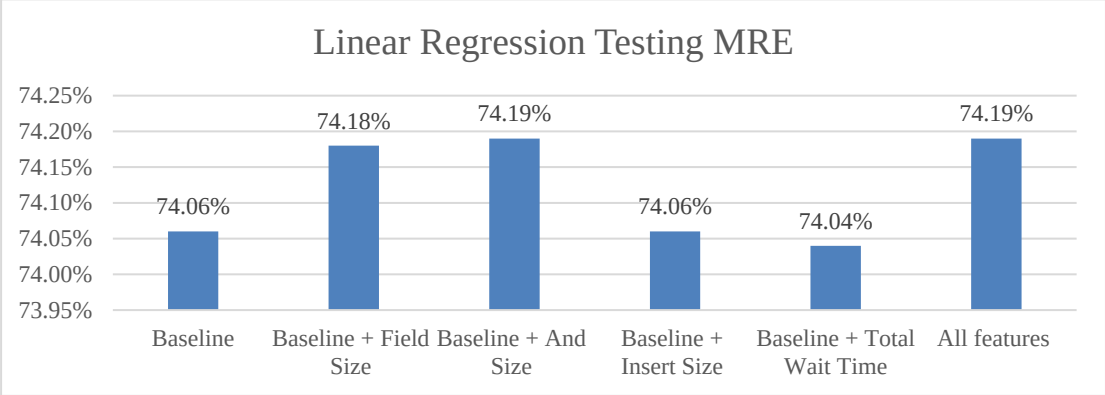
上圖分別是我們分兩個 model 中 Random forest 的 MAE 和 MRE 結果呈現。可以發現其中無論 MAE 或 MRE 皆是 Baseline 加上 Total Wait Time 表現較好，但最大最小值的差距並沒有相差太多。

B. Linear Regression

1. MAE



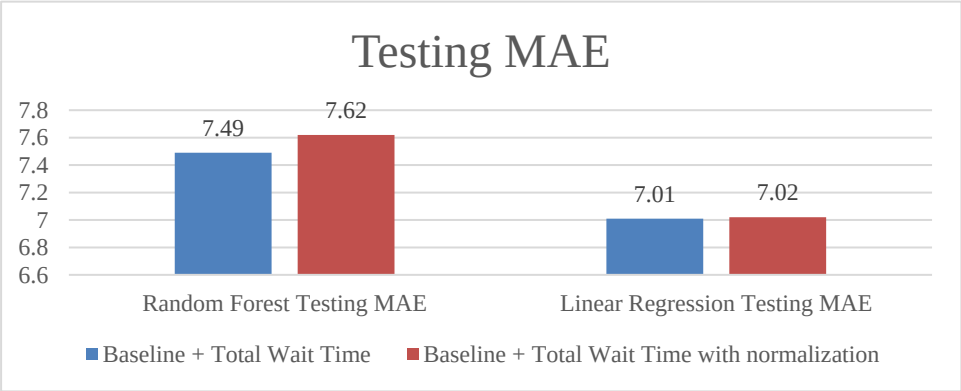
2. MRE



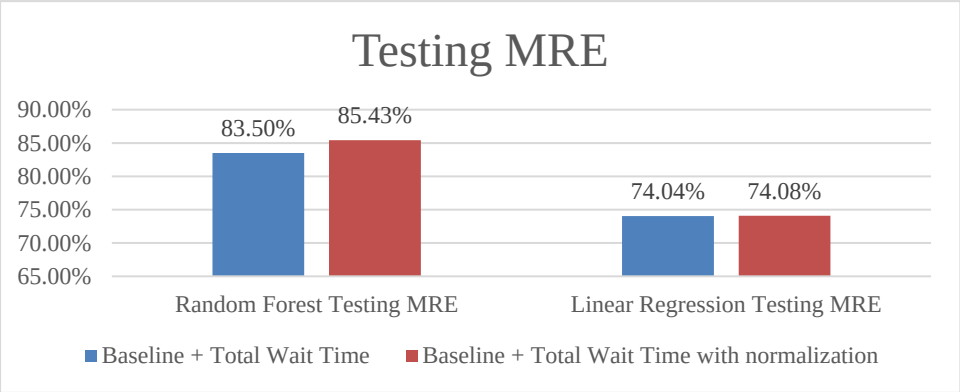
上圖分別是我們分兩個 model 中 Linear Regression 的 MAE 和 MRE 結果呈現。可以發現其中 MAE 的部分差距不大，皆落在 7.00~7.02 之間，而 MRE 的部分仍是 Baseline 加上 Total Wait Time 表現較好。

C. All features with normalization and without normalization

1. MAE



2. MRE



從前兩項實驗結果可以得出 Baseline 加上 Total Wait Time 表現相對較好，因此最後我們再比較有無使用 normalization 以及兩個 model 的結果。可以發現有沒有做 normalization 相差不大，而兩個 model 中則是 Linear Regression 的差異較小。

- Summary of experiments

  綜合上方實驗結果，最後我們把 Random Forest 換成 **Linear Regression** 配上 feature 為 **Read/write count 加上 Total Wait Time** 的組合，這樣 MAE 從 Baseline 的 **7.7** 降為 **7.01**，而 MRE 從 Baseline 的 **84.49%** 變成 **74.04%**，MAE 提升 **0.69**，MRE 提升 **10.45%**，整體的表現沒有非常大幅的提升，推測可能是我們拿的 feature 不夠全面，且 feature 之間的類型較接近。

- Branch ***final*** 為我們最後採用的 features (read/write count、total wait time)與 model (linear regression)，另一條在 107062313/db22-final-project 的 branch ***baseline*** 是我們用來測試 baseline 組合(read/write count + random forest)。

Other features we have tried but failed

- CPU load & Free physical memory size

  我們認為 Tx 開始時可用的 CPU 和記憶體大小可能會影響到 Tx 執行的效能，因此我們也嘗試拿取 Tx 執行前的 CPU Load 及 Free Physical Memory Size，然而，在嘗試實驗的過程中 Tx 執行的，拿取這兩個 Feature 會讓 Latency 變得很大（大約是原本的 10 倍），進而導致 MRE 變得太小而無法正常評估實驗結果，因此決定不使用。

```java
private void printUsage() {
  OperatingSystemMXBean operatingSystemMXBean = ManagementFactory.getOperatingSystemMXBean();

  for (Method method : operatingSystemMXBean.getClass().getDeclaredMethods()) {
    method.setAccessible(true);
    if (method.getName()=="getFreePhysicalMemorySize") {
        Object value;
        try {
            memorySize = (long)method.invoke(operatingSystemMXBean);
        }
        catch (Exception e) {
            value = e;
        }
    }
    else if (method.getName()=="getProcessCpuLoad") {
        Object value;
        try {
            cpuLoad = (double)method.invoke(operatingSystemMXBean);
        }
        catch (Exception e) {
            value = e;
        }
    }
  }
}
```

```java
private Transaction scheduleTransactionSerially(boolean isReadOnly,
        Set<PrimaryKey> readSet, Set<PrimaryKey> writeSet) {
    SERIAL_CONTROL_LOCK.lock();
    try {
        Transaction tx = VanillaDb.txMgr().newTransaction(
                Connection.TRANSACTION_SERIALIZABLE, isReadOnly);

        ConservativeConcurrencyMgr ccMgr = (ConservativeConcurrencyMgr) tx.concurrencyMgr();

        // Reserve lock so that deterministic ordering is ensured

        BeforeWaitTime = System.nanoTime();
        ccMgr.bookReadKeys(readSet);
        ccMgr.bookWriteKeys(writeSet);
        AfterWaitTime = System.nanoTime();
        printUsage();

        return tx;
    } finally {
        SERIAL_CONTROL_LOCK.unlock();
    }
}
```

- Log Count

  當 Tx commit 時會將 lsn 之前的 log 紀錄 flush 回 disk 上，在 LogMgr 中新增 `myLog` 變數紀錄要 flush 回的 disk 的 byte 數量，以及 `getLog()`回傳 `myLog` 值，若該 Tx 執行過程多寫下較多的 log，需要 flush 回 disk 的 log 也會較多，因此我們認為可以作為預測 latency 的依據。

```
public void flush(LogSeqNum lsn) {
    logMgrLock.lock();
    try {
        if (lsn.compareTo(lastFlushedLsn) >= 0) {
            if(lsn.offset() < lastFlushedLsn.offset()) myLog = lsn.offset() + 4096 - lastFlushedLsn.offset();
            else myLog = lsn.offset() - lastFlushedLsn.offset();

            flush();
        }

    } finally {
        logMgrLock.unlock();
    }
}
```

```
public long getLog() {
    return myLog;
}
```

- Max buffer size

  在 bufferMgr 裡面有紀錄一個 `bufferToFlush` 的 set，主要記錄在 Tx 執行期間用到的所有 buffer，我們在 Tx commit 時回傳這個 `bufferToFlush` 的大小作為 feature max buffer size。因為當 Tx commit 時須將 dirty buffer flush 回 disk 上，若 bufferToFlush 越大代表要 flush 的較多，也就需要做較多的 I/O 影響 Tx latency。

```
public long getMaxBufferSize()
{
    return buffersToFlush.size();
}
```

- Hit rate

  1. bufferMgr

     我們在 bufferMgr 裡面記錄兩個新變數，`totalPinCount` 以及 `hitCount`，分別紀錄呼叫 buffer pin 的次數以及 buffer hit 的次數。

     ```
     private long hitCount = 0;
     private long totalPinCount = 0;
     ```

     當呼叫 pin 的時候，計算 `totalPinCount` 次數，並且當我們要 pin 的 block 已經在 buffer 裡的時候增加 `hitCount`。

     這裡主要分成兩種情況：

     a. 第一種情況是該 block 已經在 buffer 中且 block 所在的 buffer 已經被 pin 住了，這種情況下我們直接增加 `hitCount`。

```
public Buffer pin(BlockId blk) {

    totalPinCount++;
    // Try to find out if this block has been pinned by this transaction
    PinningBuffer pinnedBuff = pinningBuffers.get(blk);
    if (pinnedBuff != null) {
        pinnedBuff.pinCount++;
        hitCount++;
        return pinnedBuff.buffer;
    }
```

b. 第二種情況是 buffer 還沒 pin，在這種情況我們對 bufferPoolMgr 呼叫 pin，bufferPoolMgr 對該 block 查詢是否已經在 buffer 中，若是該 block 已經在 buffer 內，會回傳 isHit = true，我們再判斷當 `isHit == true` 時增加 `hitCount` 數量。

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

```
if (isHit)
    hitCount++;
```

最後當 Tx commit 時，呼叫 `getHitRate()`得到 Tx 的 hit rate。

```
public double getHitRate()
{
    return (double)hitCount/(double)totalPinCount;
}
```

我們認為若要 pin 的 block 已經存在 buffer 中代表不需要再從 disk 取得 data，可以省去做 I/O 的時間，因此 hit rate 的高低會影響 Tx latency。

2. bufferPoolMgr

在前面提到我們透過 bufferPoolMgr 對沒有 pin 的 buffer 查找是否有 block 在 buffer 中，我們修改的 `pin()`的回傳結果，除了原本的 buffer 外多包一個 isHit 變數用來記錄是否找到 hit 的 buffer。

```
class MyResult
{
    Buffer buffer;
    boolean isHit = false;
    MyResult(Buffer buffer, boolean isHit) {
        this.buffer = buffer;
        this.isHit = isHit;
    }
}
```

在 pin()中，當我們找到 hit 的 buffer，設定 `isHit=true` 並回傳結果，其餘則回傳 `isHit=flase`。

```
} else {
    // Get the lock of buffer
    buff.getSwapLock().lock();

    // Optimization
    // Early release the blockLock
    // because the following txs, which need the same block, will get the same non-null buffer
    blockLock.unlock();

    try {
        // Check its block id before pinning since it might be swapped
        if (buff.block().equals(blk)) {
            if (!buff.isPinned())
                numAvailable.decrementAndGet();
            buff.pin();
            return new MyResult(buff, true);
        }
        return new MyResult(buff, true);

    } finally {
        // Release the lock of buffer
        buff.getSwapLock().unlock();
    }
}
```

```
if (buff.getSwapLock().tryLock()) {
    try {
        // Check if there is no one use it
        if (!buff.isPinned() && !buff.checkRecentlyPinnedAndReset()) {
            this.lastReplacedBuff = currBlk;

            // Swap
            BlockId oldBlk = buff.block();
            if (oldBlk != null)
                blockMap.remove(oldBlk);
            buff.assignToBlock(blk);
            blockMap.put(blk, buff);
            if (!buff.isPinned())
                numAvailable.decrementAndGet();

            // Pin this buffer
            buff.pin();
            return new MyResult(buff, false);
        }
    } finally {
        // Release the lock of buffer
        buff.getSwapLock().unlock();
    }
}
```