Report 2

組員: 109062274, 109062314, 109062315

`UpdateItemPriceTxnParamGen`

TA的作法是透過一個可序列化的class `UpdateItemPriceTxnParam`來打包生成的參數，將id和priceRaise視為一個物件，然後依序放入`Linklist`中

```
for (int i = 0; i < WRITE_COUNT; i++) {
    int itemId = rvg.number(1,As2BenchConstants.NUM_ITEMS);
    double raise = ((double) rvg.number(0, MAX_RAISE)) / 10;

    paramList.add(new UpdateItemPriceTxnParam(itemId, raise));
}
```

而我們的做法就是純粹將生成的參數按照id,priceRaise依序放入`ArrayList`中

```
for (int i = 0; i < TOTAL_UPDATE_COUNT; i++){
    paramList.add(rvg.number(1, As2BenchConstants.NUM_ITEMS)); // Randomly pick an item id
    paramList.add(rvg.randomDoubleIncrRange(0.0, 5.0, 1000.0)); // Randomly generate a number 0.0 - 5.0 for price raise
}
```

比較
TA的寫法相較來說較好，把相關的內容打包成一個物件比較合乎邏輯，也使得進行迭代時比較不容易犯錯
e.g. 不需一個index記錄當前為第幾個更新，另一個index紀錄當前在容器中的位置
TA's `UpdateItemPriceProcParamHelper`

```
for (int i = 0; i < readCount; i++) {
    // indexCnt只是為了能從index=1開始，因為index=0的位置放的是更新次數
    itemIds[i] = (Integer) (((UpdateItemPriceTxnParam)
pars[indexCnt]).itemId);
    raises[i] = (Double) (((UpdateItemPriceTxnParam)
pars[indexCnt]).raise);
    indexCnt++;
}
```

我們的 `UpdatePriceProcParamHelper`

```
for (int i = 0; i < updateCount; i++) {
    updateItemId[i] = (Integer) args[indexCnt++];
    updatePriceRaise[i] = (Double) args[indexCnt++];
}
```

`UpdateItemPriceTxnJdbcJob::executesql()`

TA與我們的差別只有以下兩點，邏輯上基本相同

- 事前先把參數全部讀出來

```
for (int i = 0; i < readCount; i++) {
    itemIds[i] = (Integer) (((UpdateItemPriceTxnParam) pars[i +
1]).itemId);
    raises[i] = (Double) (((UpdateItemPriceTxnParam) pars[i +
1]).raise);
}
```

- 額外寫一個函式決定價錢最後的更新值

```
private Double updatePrice(double originalPrice, double raise) {
    return (Double) (originalPrice > As2BenchConstants.MAX_PRICE ?
As2BenchConstants.MIN_PRICE : originalPrice + raise);
}
```

比較
助教這樣的寫法，好處就是在code會寫的比較乾淨

`UpdateItemPriceTxnProc::executesql()`

TA's solution

```
Plan p = VanillaDb.newPlanner().createQueryPlan("SELECT i_name, i_price FROM item WHERE i_id = " + iid, tx);
Scan s = p.open();
...
int result = VanillaDb.newPlanner().executeUpdate("UPDATE item SET i_price = " + paramHelper.getUpdatedItemPrice(idx) + " WHERE i_id = " + iid, tx);
if (result == 0) {
    throw new RuntimeException("Could not update item record with i_id = " + iid);
}
```

Our solution

```
Scan s = StoredProcedureHelper.executeQuery(
"SELECT i_name, i_price FROM item WHERE i_id = " + iid,tx);
...
double newPrice = priceRaise + price;
if (newPrice > As2BenchConstants.MAX_PRICE){
    newPrice = As2BenchConstants.MIN_PRICE;
}
StoredProcedureHelper.executeUpdate("UPDATE item SET i_price = " + newPrice + "WHERE i_id = " + iid,tx);
paramHelper.setItemPrice(newPrice, idx);
```

我們寫的方法基本上邏輯上是跟助教一樣，但在執行query的地方，像是`executeQuery()`和`executeUpdate()`已經有`StoredProcedureHelper`API可以呼叫，就不用再呼叫`VanillaDb.newPlanner().executeQueryPlan()`，所以我覺得我們寫得較簡潔。而我們在這裡決定新的 price 的，助教的是額外在`UpdateItemPriceProcParamHelper`額外寫一個

method決定新的price。我覺得助教的寫法比較簡潔，這裡我們的code就較凌亂。

`UpdateItemPriceProcParamHelper`

TA's solution

```
1 public double getUpdatedItemPrice(int idx) {
2     double originalPrice = itemPrices[idx] ;
3     return (Double) (originalPrice > As2BenchConstants.MAX_PRICE ?
  As2BenchConstants.MIN_PRICE : originalPrice + raises[idx]);
4 }
```

在這裡TA多一個`getUpdatedItemPrice()` method，我們是沒有寫這個method的。我們判斷新的price的邏輯是寫在`UpdateItemPriceTxnProc::executesql()`中。我覺得助教這樣寫較好，這樣可以把code比較不凌亂而且debug時更有條理。

`StatisticMgr` 產生 .csv 檔的效能比較

雖然計算統計資料與資料庫運行時的效能沒有直接關係，不過結論上來說，我們的實作效能明顯較助教的快很多，分析如下。

> *定義 n: 總資料數，$slots$: 共有幾個時間區間，$slotSize$: 某時間區間的資料數 (粗略)。*

助教的流程為:

- `outputReport` 呼叫 `outputDetailReport` 後呼叫 `outputAs2Report`
- 於 `outputDetailReport` 迴圈遍歷 `resultSets` 中，將當前 `resultSet` 一併加入 `TreeMap` 內的 `ArrayList` 為後續準備，複雜度 $O(n \times slotSize \times \log(slots))$，空間複雜度為 $O(n)$，後續沒有使用額外空間。

  > *順帶一提，我們本來以為不能修改 `outputDetailReport`，不然也會將利用此次遍歷來節省複雜度。*

- 於 `outputAs2Report` 遍歷 `TreeMap`，對例外進行檢查後呼叫 `makeStatString`
  - 取得 `TreeMap` 值，複雜度加總 $O(slots \log(slots))$

- `makeStatString` 取得統計結果的字串後，寫入檔案。
    - 呼叫 `Collection::sort` 複雜度 $O(slotSize \log(slotSize))$
    - `min`, `max`, `median`, `mean` 複雜度分別為 $O(slotSize)$
    - `lowerQ` (25th), `upperQ` (75th) 複雜度分別為 $O(\frac{slotSize}{2})$

我們的流程為:

- `outputReport` 呼叫 `outputStatisticReport`
- 遍歷 `resultSets` 蒐集一個 time slot 的資料

    > *由於以為不能改 `outputDetailReport`，故不計算遍歷 `resultSets` 的時間複雜度。*

    - 蒐集資料使用 `ArrayList`，時間複雜度共 $O(n)$，空間複雜度 $O(slotSize)$。
- 呼叫 `calculateStatistic` 計算統計資料為字串
    - 比起助教的程式碼，我們直接略過資料筆數小於 4 筆的情況，這裡應該是助教的考量比較周到。
    - 遍歷 `ArrayList` 同時取得 `min`, `max`，複雜度 $O(slotSize)$
    - 以 `ArrayList` 建立 `PriorityQueue`，時間複雜度 $O(slotSize \log(slotSize))$，空間複雜度 $O(slotSize)$
    - 不斷資料以獲得 25th, median, 75th，複雜度 $O(\frac{3}{4} slotSize \log(slotSize))$
    - 相當於進行部分的 Heap Sort，可視複雜度約小於 `Collection::sort`

可見我們的複雜度比較大致如下表

	Time	Space
助教	$n \times slotSize \times \log(slots) + slots \times \left(\log(slots) + slotSize \log(slotSize) + 8slotSize\right)$	n
我們	$n + 2slotSize + slots \times slotSize \log(slotSize)$	$2slotSize$

總結而言，助教的程式碼善用 Java 的 `Collection` API，滿優美的，不過為此在效能上付出的代價頗高；我們的實作則善用 Heap 資料結構的特性並重複利用 `ArrayList`，減少時間與空間複雜度。

`READ_WRITE_TX_RATE` 相關比較

我們將此參數加在 `VanillaBenchParameter`，不過助教加在 `As2BenchConstants`，應該是更為合理的作法。

在 `As2BenchmarkRte` 選擇 executor 中，我們在建構 `As2BenchmarkRte` 時便準備好 read 和 write 的 executor，降低 `getTxExcecutor` (注意這裡助教有 typo，我們對此進行了修正) 之 branching 和 instantiation 的性能損耗。考慮資料庫針對 R/W 的請求次數之多，我們的作法好取得更好的性能優勢。

> *簡潔有力的 `getTxExcecutor`*

```
// in As2BenchmarkRte.java
protected As2BenchmarkTxExecutor
getTxExecutor(As2BenchTransactionType type) {
    return executors[current];
}
```

對於 `getNextTxType`，由於我們實際上多一個 `TASK_TX_DIST` 參數，方便未來擴展各種 benching 的佔比:

```
// in VanillaBenchParameters.java
/**
 * Percentage of benchmark tasks of
 * [ ReadItemTxn, UpdateItemPriceTxn ] distribution
 */
public static final double[] TASK_TX_DIST;
```

所以 random 時為呼叫 `randomChooseFromDistribution` 方法，並處理 edge case，這裡效能上是助教的較好，不過若考慮擴展性，我們的更能輕鬆勝任。