Team5 Assignment2 Report1

107062313 黃寶萱　　107062117 李采蓉　　107062317 陳怡汝

◈ Part1 : Implement vanillabench property

A. vanillabench.properties

```
# Read READ_WRITE_TX_RATE
org.vanilladb.bench.benchmarks.as2.As2BenchConstants.READ_WRITE_TX_RATE=0.5
# Update count
org.vanilladb.bench.benchmarks.as2.rte.As2UpdateItemPriceParamGen.TOTAL_UPDATE_COUNT=10
```

設定需要的參數：

1. READ_WRITE_TX_RATE
2. TOTAL_UPDATE_COUNT

B. org.vanilladb.bench.benchmarks.as2.As2BenchConstants.java

```
static {
    NUM_ITEMS = BenchProperties.getLoader().getPropertyAsInteger(
            As2BenchConstants.class.getName() + ".NUM_ITEMS", 100000);

    READ_WRITE_TX_RATE = BenchProperties.getLoader().getPropertyAsDouble(
        As2BenchConstants.class.getName() + ".READ_WRITE_TX_RATE", 0.5);
}
```

按照 NUM_ITEMS 的方法讀取 READ_WRITE_TX_RATE。

C. org.vanilladb.bench.benchmarks.as2.As2BenchTransactionType.java

```
// Update item price procedures
UPDATE_ITEM_PRICE(true);
```

新增一個 transaction type UPDATE_ITEM_PRICE，並設為 true，代表這個 transaction type 是用來做 update item price。

◈ Part2 : Implement RTE

A. org.vanilladb.bench.benchmarks.as2.rte.As2BenchmarkRte.java

```
protected As2BenchTransactionType getNextTxType() {
    RandomValueGenerator random = new RandomValueGenerator();
    int RandomSeed = 10;
    if(random.number(0, RandomSeed-1) < As2BenchConstants.READ_WRITE_TX_RATE * RandomSeed) {
        return As2BenchTransactionType.READ_ITEM;
    }
    else {
        return As2BenchTransactionType.UPDATE_ITEM_PRICE;
    }
}

protected As2BenchmarkTxExecutor getTxExeutor(As2BenchTransactionType type) {
    switch(type){
    case UPDATE_ITEM_PRICE:
        executor = new As2BenchmarkTxExecutor(new As2UpdateItemPriceParamGen());
        break;
    case READ_ITEM:
        executor = new As2BenchmarkTxExecutor(new As2ReadItemParamGen());
        break;
    default:
        executor = new As2BenchmarkTxExecutor(new As2UpdateItemPriceParamGen());
        break;
    }
    return executor;
}
```

這個 class 繼承自(extends) class RemoteTerminalEmulator，因此需要在這裡藉由 function getNextTxType()以及 function getTxExeutor()定義兩個不同的 transaction type 以及 executor。

1. getNextTxType()

   利用 RandomValueGenerator().number() 產生出一個 random 的數值，以先前自行定義的機率 READ_WRITE_TX_RATE 作為判斷的依據，使系統隨機 return 下一個要執行的 transaction type 為 READ_ITEM 或 UPDATE_ITEM_PRICE。

2. getTxExeutor()

   根據要執行的 transaction type 利用 case condition 產生對應的 executor，若要產生負責執行 UPDATE_ITEM_PRICE transaction 的 executor，便會呼叫 As2UpdateItemPriceParamGen()來定義相關的參數設定。

B. org.vanilladb.bench.benchmarks.as2.rte.As2UpdateItemPriceParamGen.java

```java
@Override
public As2BenchTransactionType getTxnType() {
    return As2BenchTransactionType.UPDATE_ITEM_PRICE;
}

@Override
public Object[] generateParameter() {
    RandomValueGenerator rvg = new RandomValueGenerator();
    ArrayList<Object> paramList = new ArrayList<Object>();

    // Set read count
    paramList.add(TOTAL_UPDATE_COUNT);
    for (int i = 0; i < TOTAL_UPDATE_COUNT; i++) {
        int itemid = rvg.number(1, As2BenchConstants.NUM_ITEMS);
        double priceraise =  ((double)rvg.number(0, 50)) / 10.0;
        paramList.add(itemid);
        paramList.add(priceraise);
    }
    return paramList.toArray(new Object[0]);
}
```

這個 class 的主要功能為定義執行 UPDATE_ITEM_PRICE transaction 時所需的參數。

1. getTxnType()

   直接回傳 transaction type，亦即 UPDATE_ITEM_PRICE。

2. generateParameter()

   這個 function 負責產生並記錄每個 update 的 itemid 與 priceraise，首先先將 TOTAL_UPDATE_COUNT=10 加在 paramList 的最前面，再利用 for loop 分別將每個 iteration 隨機產生的 itemid 以及 priceraise 依序加進 paramList，隨機產生的 priceraise 會介在 0.0~5.0 之間，亦即跑完整個 for loop 後，paramList 會有 1+2*10 = 21 項 data。

◈ Part3 : Implement JDBC

A. org.vanilladb.bench.benchmarks.as2.rte.jdbc.As2BenchJdbcExecutor.java

```
case UPDATE_ITEM_PRICE:
    return new UpdateItemPriceTxnJdbcJob().execute(conn, pars);
```

在原本的 function 中新增 transaction type 為 UPDATE_ITEM_PRICE 的情況。

B. org.vanilladb.bench.server.param.as2.UpdateItemPriceProcParamHelper.java

```
@Override
public void prepareParameters(Object... pars) {
    // Show the contents of paramters
    // System.out.println("Params: " + Arrays.toString(pars));
    int indexCnt = 0;
    updateCount = (Integer) pars[indexCnt++];
    updateItemId = new int[updateCount];
    updateItemRaise= new double[updateCount];
    itemName = new String[updateCount];
    itemPrice = new double[updateCount];

    for (int i = 0; i < updateCount; i++) {
        updateItemId[i] = (Integer) pars[indexCnt++];
        updateItemRaise[i] = (Double) pars[indexCnt++];
    }

}
```

這個 function 會在開始執行 UpdateItemPriceTxnJdbcJob.executor()時被呼叫到，設定好相關的參數資訊後才開始執行 transaction，先將之前存在 paramList 中的第一個資料拿出來，即 TOTAL_UPDATE_COUNT=10，並利用 for loop 將所有的 itemid 與 priceraise 存在兩個 array 中：updateItemId[ ]、updateItemRaise[ ]。

```java
public int getUpdateCount() {
    return updateCount;
}

public int getUpdateItemId(int index) {
    return updateItemId[index];
}

public double getUpdateItemRaise(int index) {
    return updateItemRaise[index];
}

public double getUpdateItemPrice(int index) {
    double newPrice = itemPrice[index] + updateItemRaise[index];
    if(newPrice > As2BenchConstants.MAX_PRICE) newPrice = As2BenchConstants.MIN_PRICE;
    return newPrice;
}
```

```java
public void setItemName(String s, int idx) {
    itemName[idx] = s;
}

public void setItemPrice(double d, int idx) {
    itemPrice[idx] = d;
}

public void setUpdateItemPrice(double price, int idx) {
    // double newPrice = itemPrice[idx] + updateItemRaise[idx];
    // if(newPrice > As2BenchConstants.MAX_PRICE) newPrice = As2BenchConstants.MIN_PRICE;
    // itemPrice[idx] = newPrice;
    itemPrice[idx] = price;
}
```

同時也會在這個 class 中設定一些 function，方便後續讀取 item id、raise、price，以及設定 item name、price、updated price，其中 getUpdateItemPrice() 根據 assignment spec 要求，若 updated price 超過 MAX_PRICE 便將他更新為 MIN_PRICE。

◈ Part4 : Implement Stored Procedure

A. org.vanilladb.bench.benchmarks.as2.rte.jdbc.UpdateItemPriceTxnJdbcJob.java

```
Statement statement = conn.createStatement();
ResultSet rs = null;
int returnvalue = 0;

// SELECT and UPDATE
for (int i = 0; i < paramHelper.getUpdateCount(); i++) {
    // SELECT
    int iid = paramHelper.getUpdateItemId(i);
    String sql = "SELECT i_name, i_price FROM item WHERE i_id = " + iid;
    rs = statement.executeQuery(sql);
    rs.beforeFirst();
    if (rs.next()) {
        outputMsg.append(String.format("'%s', %d, ", rs.getString("i_name"), rs.getDouble("i_price")));
    } else
        throw new RuntimeException("cannot find the record with i_id = " + iid);
    rs.close();

    // UPDATE
    String update = "UPDATE item SET i_price = " + paramHelper.getUpdateItemPrice(i) + "WHERE i_id = " + iid;
    returnvalue = statement.executeUpdate(update);
    if (returnvalue > 0) {
        outputMsg.append(String.format("'%s', ", rs.getString("i_name")));
        logger.info("update i_id = " + iid + "well in JDBC");
    } else
        throw new RuntimeException("cannot update the record with i_id = " + iid);
}

conn.commit();
```

在這個 function 中會先 create 一個 statement，利用 for loop 跑 10 個 iteration (由 paramHelper.getUpdateCount()得到 TOTAL_UPDATE_COUNT)，在每個 iteration 中會執行：

1. 呼叫 paramHelper.getUpdateItemId(i)得到 item id
2. 下 SQL SELECT 指令，並由 statement.executeQuery()得到 ResultSet
3. outputMsg 顯示 SELECT 後的 i_name、i_price 的結果
4. 下 SQL UPDATE 指令，呼叫 paramHelper.getUpdateItemPrice(i) 得到 update 後的 price，將 SQL 指令傳送到 statement.executeUpdate()執行，回傳一個受影響的資料列數目的 int value

最後 commit 這個 transaction。

B. org.vanilladb.bench.server.procedure.as2.As2BenchStoredProcFactory.java

```
case UPDATE_ITEM_PRICE:
    sp = new UpdateItemPriceTxnProc();
    break;
```

在原 class 中新增 transaction type 為 UPDATE_ITEM_PRICE 的 case，並 new 一個負責處理 update transaction 的 procedure。

C. org.vanilladb.bench.server.procedure.as2.UpdateItemPriceTxnProc.java

```
UpdateItemPriceProcParamHelper paramHelper = getParamHelper();
Transaction tx = getTransaction();

for (int idx = 0; idx < paramHelper.getUpdateCount(); idx++) {
    // logger.info("getUpdateItemId = " + paramHelper.getUpdateItemId(idx));
    // logger.info("getUpdateItemRaise = " + paramHelper.getUpdateItemRaise(idx));
    // SELECT
    int iid = paramHelper.getUpdateItemId(idx);
    Scan s = StoredProcedureHelper.executeQuery(
        "SELECT i_name, i_price FROM item WHERE i_id = " + iid,
        tx
    );
    s.beforeFirst();
    if (s.next()) {
        String name = (String) s.getVal("i_name").asJavaVal();
        double price = (Double) s.getVal("i_price").asJavaVal();
        paramHelper.setItemName(name, idx);
        paramHelper.setItemPrice(price, idx);
    } else
        throw new RuntimeException("Could not find item record with i_id = " + iid);
    s.close();
    // UPDATE
    int value = StoredProcedureHelper.executeUpdate(
            "UPDATE item SET i_price = " + paramHelper.getUpdateItemPrice(idx) + "WHERE i_id = " + iid,
            tx
    );
    if(value > 0) {
        double price = paramHelper.getUpdateItemPrice(idx);
        // logger.info("setUpdateItemPrice idx = " + idx);
        paramHelper.setUpdateItemPrice(price, idx);
    } else
        throw new RuntimeException("Could not update item record with i_id = " + iid);
}
```

同樣先呼叫 getParamHelper()得到 UpdateItemPriceProcParamHelper paramHelper 來取得接下來會使用到的參數資料，因為這裡是 stored procedure 的部分，因此需要使用到 getTransaction()，在每次的 for loop iteration 中：

1. 呼叫 paramHelper.getUpdateItemId(idx)取得 item id
2. 藉由 StoredProcedureHelper.executeQuery()下 SELECT 指令
3. 呼叫 paramHelper.setItemName()、paramHelper.setItemPrice()設定該 iterm id 的資料
4. 藉由 StoredProcedureHelper.executeUpdate()下 UPDATE 指令，同時呼叫 paramHelper.getUpdateItemPrice(idx)取得 updated price

◈ Part5: CSV result

time(sec)	throughput(txs)	avg_latency(ms)	min(ms)	max(ms)	25th_lat(ms)	median_lat(ms)	75th_lat(ms)
5	7400	0	0	18	0	1	1
10	7959	0	0	11	0	1	1
15	7935	0	0	22	0	1	1
20	7992	0	0	19	0	1	1
25	8056	0	0	21	0	1	1
30	8057	0	0	15	0	1	1
35	7994	0	0	38	0	1	1
40	8188	0	0	19	0	1	1
45	8095	0	0	19	0	1	1
50	8044	0	0	22	0	1	1
55	7983	0	0	18	0	1	1

◈ Part6 : Experiment

A. Environment setting

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

B. Experiment results

1. **JDBC** : READ_WRITE_TX_RATE = 0.25

READ_ITEM - committed: 5015	aborted: 0	avg latency: 5 ms
UPDATE_ITEM_PRICE - committed: 11966	aborted: 0	avg latency: 7 ms
TOTAL - committed: 16981	aborted: 0	avg latency: 7 ms

2. **JDBC** : READ_WRITE_TX_RATE = 0.5

READ_ITEM - committed: 8100	aborted: 0	avg latency: 6 ms
UPDATE_ITEM_PRICE - committed: 8058	aborted: 0	avg latency: 8 ms
TOTAL - committed: 16158	aborted: 0	avg latency: 7 ms

3. **JDBC** : READ_WRITE_TX_RATE = 1.0

READ_ITEM - committed: 20164	aborted: 0	avg latency: 5 ms
UPDATE_ITEM_PRICE - committed: 0	aborted: 0	avg latency: 0 ms
TOTAL - committed: 20164	aborted: 0	avg latency: 6 ms

4. **Stored Procedure** : READ_WRITE_TX_RATE = 0.25

READ_ITEM - committed: 20962	aborted: 0	avg latency: 0 ms
UPDATE_ITEM_PRICE - committed: 48504	aborted: 0	avg latency: 2 ms
TOTAL - committed: 69466	aborted: 0	avg latency: 2 ms

5. **Stored Procedure** : READ_WRITE_TX_RATE = 0.5

READ_ITEM - committed: 38314	aborted: 0	avg latency: 0 ms
UPDATE_ITEM_PRICE - committed: 37543	aborted: 0	avg latency: 2 ms
TOTAL - committed: 75857	aborted: 0	avg latency: 2 ms

6. **Stored Procedure** : READ_WRITE_TX_RATE = 1.0

READ_ITEM - committed: 311981	aborted: 0	avg latency: 0 ms
UPDATE_ITEM_PRICE - committed: 0	aborted: 0	avg latency: 0 ms
TOTAL - committed: 311981	aborted: 0	avg latency: 0 ms

C. Analysis and explanation

1. JDBC and Stored Procedure

   從我們測試的結果中可以發現 JDBC 的 throughput 比 Stored Procedure 低很多，因為 JDBC 每次建立一個 statement 執行 SQL 指令時都需要重新建立 query plan tree，而 Stored Procedure 在執行 SQL 指令時可以透過 planner 選出一個較好的 query plan tree，因此能有較高的 throughput。

2. Different ratio of ReadItemTxn and UpdateItemPriceTxn

   當我們提高 UpdateItemPriceTxn 的比率，throughput 降低，推測原因為在 UpdateItemPriceTxn 中需要執行 SELECT 及 UPDATE 兩個 SQL，而在 ReadItemTxn 只需要執行 SELECT 一個 SQL，因此 ReadItemTxn 執行速度比 UpdateItemPriceTxn 更快，可得到更高的 throughput。

3. Other parameters

   a. RTE_SLEEP_TIME = 100

READ_ITEM - committed: 550	aborted: 0	avg latency: 3 ms
UPDATE_ITEM_PRICE - committed: 556	aborted: 0	avg latency: 6 ms
TOTAL - committed: 1106	aborted: 0	avg latency: 5 ms

   根據註解嘗試把 RTE_SLEEP_TIME 改成 100，發現整體的 latency 增加以及 commit 數量降低，我們推論這是合理的結果，因為 RTE 從原本不會 sleep 變成會進入 sleep 狀態。雖然在目前測試的 case 上因為沒有發生 abort 而無法明顯看出更改 RTE_SLEEP_TIME 的效果，但我們認為更改 RTE_SLEEP_TIME 應該可以降低 RTE 之間因為資源爭搶而造成 deadlock 的機率。

b. NUM_RTE

NUM_RTE = 5

READ_ITEM - committed: 54506	aborted: 0	avg latency: 0 ms
UPDATE_ITEM_PRICE - committed: 54312	aborted: 0	avg latency: 4 ms
TOTAL - committed: 108818	aborted: 0	avg latency: 3 ms

NUM_RTE = 10

READ_ITEM - committed: 48416	aborted: 0	avg latency: 1 ms
UPDATE_ITEM_PRICE - committed: 47939	aborted: 2	avg latency: 10 ms
TOTAL - committed: 96355	aborted: 2	avg latency: 6 ms

NUM_RTE = 20

READ_ITEM - committed: 60023	aborted: 2	avg latency: 1 ms
UPDATE_ITEM_PRICE - committed: 60144	aborted: 1	avg latency: 18 ms
TOTAL - committed: 120167	aborted: 3	avg latency: 10 ms

NUM_RTE = 40

READ_ITEM - committed: 61137	aborted: 5	avg latency: 1 ms
UPDATE_ITEM_PRICE - committed: 61226	aborted: 3	avg latency: 37 ms
TOTAL - committed: 122363	aborted: 8	avg latency: 20 ms

從 NUM_RTE=5 到 NUM_RTE=40 我們發現開越多 RTE，可以讓 commit 數量增加，也導致 abort 產生，以及所需要的 latency 增加，可能是因為雖然有更多 RTE 可以執行工作，但由於執行資源有限，有些 RTE 還是需要等待資源釋放後才可以執行，因而使 latency 增加，且由於 RTE 增加造成更多資源爭搶，所以 abort 的狀況也相對增加，因此 RTE 對提升 commit 數量有所上限，在資源有限的情況下無法藉由無限增加 RTE 數量達到更多的 commit。