As2 Report

105062206蔡哲維
105062306胡世昕

前置作業：修改As2BenchTxnType、As2BenchRte，新增UpdateItemPriceParamGen

由於此次作業要新增update的transaction，首先需要做的是修改As2BenchTxnType，新增一個名為UPDATE_ITEM的enum。

```
public enum As2BenchTxnType implements BenchTransactionType {
    // Loading procedures
    TESTBED_LOADER(false),

    // Database checking procedures
    CHECK_DATABASE(false),

    // Benchmarking procedures
    READ_ITEM(true),

    // Benchmarking procedures for update items (assignment2)
    UPDATE_ITEM(true);
```

接著，必須去新增一個UpdateItemPriceParamGen來產生要去修改的item　id以及price參數。

```
public class As2UpdateItemParamGen implements TxParamGenerator<As2BenchTxnType> {

    private static final int UPDATE_COUNT = 10;

    //Override
    public As2BenchTxnType getTxnType() {
        return As2BenchTxnType.UPDATE_ITEM;
    }
    //Override
    public Object[] generateParameter() {
        RandomValueGenerator rvg = new RandomValueGenerator();
        LinkedList<Object> paramList = new LinkedList<Object>();

        paramList.add(UPDATE_COUNT);
        for (int i = 0; i < UPDATE_COUNT; i++)
            paramList.add(rvg.number(1, As2BenchConstants.NUM_ITEMS));
        for (int i = 0; i < UPDATE_COUNT; i++)
            paramList.add(rvg.randomDoubleIncrRange(0.0,5.0,0.1));
        return paramList.toArray();
    }
}
```

此物件所實作的兩個method，getTxnType是用來讓TxnExecutor了解要執行的txn是什麼type，故這裡回傳UPDATE_ITEM。

而在generateParameter中，首先在paramList中插入update item之數量，接著隨機生成要update的item id，最後在0~5之間生成要raise的價錢。此method中生成的paramList會回傳給As2BenchTxnExecutor，接著透過JDBC或SP去執行update動作。

實作完parameter generator後，去修改As2BenchRte。
原先在As2BenchRte中，只有一個As2BenchTxnExecutor，是去專門執行read item工作的。故首先要做的，是再新增一個As2BenchTxnExecutor，其專門去執行update item的工作（所以其建構子傳入的是UpdateItemParamGen的物件）。

```
public As2BenchRte(SutConnection conn, StatisticMgr statMgr) {
    super(conn, statMgr);
    executor = new As2BenchTxExecutor(new As2ReadItemParamGen());
    executor_for_update = new As2BenchTxExecutor(new As2UpdateItemParamGen());
    counter = 0;
}
```

又由於spec要求，必須要可以從property file中讀取read write的比例，來決定read item跟update item的比例，故新增兩個field，READ_RATIO和WRITE_RATIO，其值從讀取property file中的設定來決定。（必須也在benchproperty file中新增這兩個設定值）

```
//Record read/write ratio
public static final double READ_RATIO = BenchProperties.getLoader().getPropertyAsDouble(As2BenchRte.class.getName() + ".READ_RATIO", 1.0);
public static final double WRITE_RATIO = BenchProperties.getLoader().getPropertyAsDouble(As2BenchRte.class.getName() + ".WRITE_RATIO", 0.0);
```

有了ratio，在getNextTxnType這個method中，就可以根據ratio來決定read item跟update item的執行比例。

```
protected As2BenchTxnType getNextTxType() {
    int read_times = (int)(READ_RATIO*10);
    /*int write_times = (int)(WRITE_RATIO*10);*/
    if(READ_RATIO == 1.0) {
        return As2BenchTxnType.READ_ITEM;
    }
    else if(WRITE_RATIO == 1.0) {
        return As2BenchTxnType.UPDATE_ITEM;
    }
    else if(counter < read_times) {
        counter++;
        return As2BenchTxnType.READ_ITEM;
    }
    else {
        //for write type transactions
        counter++;
        counter = counter % 10;
        return As2BenchTxnType.UPDATE_ITEM;
    }
    /*else if(counter-read_times < write_times-1) {
        counter++;
```

實際在執行benchmark時，會根據得到的下一個txn type為何，而決定要得到哪一個Txn executor，故最後必須去修改getExecutor這個method。

```
protected As2BenchTxExecutor getTxExeutor(As2BenchTxnType type) {

    if (type == As2BenchTxnType.READ_ITEM)
        return executor;
    else if(type == As2BenchTxnType.UPDATE_ITEM)
        return executor_for_update;
    else
        return executor;
```

在這裡判斷，若下一個txn為read item，則回傳executor（擁有ReadItemParamGen物件），若為update item，則回傳executor_for_update（擁有UpdateItemParamGen物件）。做到這裡，前置作業已完成。

JDBC實作：

Client App在呼叫執行benchmark後，根據BenchProperties中定義創建NUM_RTES個RTE。每個RTE將獲得As2BenchTxExecutor物件和一條與server的連線。

創建完成後每個RTE(thread)執行start()，在benchnmarking (60 seconds) 結束前執行executeTxnCycle。根據propertie file中的R/W比率，透過getNextTxType 切換執行ReadItemTxnJdbcJob().execute 與 UpdateItemPriceTxnJdbcJob().execute。

ReadItemTxnJdbcJob().execute用傳入的connection新增一java.sql.Statement，並根據傳入的參數透過statement.executequery執行sql: select指令，並從伺服器回應之resultset中擷取需要資料存入outputMsg，最後回應一VanillaDbJdbcResultSet後結束。

UpdateItemPriceTxnJdbcJob().execute用傳入的connection新增一java.sql.Statement，並根據傳入參數中的itemIds透過statement.executequery執行sql: select指令，並從伺服器回應之resultset中擷取需要資料存入outputMsg。判斷resultset中i_price是否超過MAX_PRICE後將update_price設成MIN_PRICE或i_price+傳入參數中的 raise_value。透過statement.executeUpdate執行sql: update指令。最後回應一VanillaDbJdbcResultSet後結束。（下圖為JDBC UpdateItemJob主要實作內容）

```
Statement statement = conn.createStatement();
ResultSet rs = null;
for (int i = 0; i < 10; i++) {
    String sql = "SELECT i_name, i_price FROM item WHERE i_id = " + itemIds[i];

    rs = statement.executeQuery(sql);
    rs.beforeFirst();
    if (rs.next()) {
        outputMsg.append(String.format("'%s', ", rs.getString("i_name")));
        origin_price = rs.getDouble("i_price");
        update_price = origin_price + raise_value[i];

        if( rs.getDouble("i_price")>As2BenchConstants.MAX_PRICE)
            sql = "UPDATE item SET i_price = " + As2BenchConstants.MIN_PRICE;
        else
            sql = "UPDATE item SET i_price = " + update_price;
        sql += " WHERE i_id = " + itemIds[i];
```

StoreProcedure實作：

在StoreProcedure的部分，我新增了兩個class：UpdateItemProcParamHelper和As2UpdataItemProc。
UpdateItemProcParamHelper是用來協助As2UpdataItemProc，將其要處理的parameters準備好，以及更新後的結果（itemName及itemPrice）包成SpResultRecord。

所以在UpdateItemProcParamHelper中，比較重要的是這兩個function的實作。

prepareParameters：

```
@Override
public void prepareParameters(Object... pars) {
    // TODO Auto-generated method stub
    int indexCnt = 0;

    updateCount = (Integer)pars[indexCnt++];
    updateItemId = new int[updateCount];
    updateItemPrice = new double[updateCount];
    itemName = new String[updateCount];
    itemPrice = new double[updateCount];

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

}
```

在這裡，我們將要更新的item數量assign給updateCount，將每個要update的itemId，assign給updateItemId，將每個要update的值，assign給updateItemPrice。此外這裡new了兩個array，itemName和itemPrice，是用來記錄更新之後item的資料。

newResultSetRecord：

```
@Override
public SpResultRecord newResultSetRecord() {
    // TODO Auto-generated method stub
    SpResultRecord rec = new SpResultRecord();
    rec.setVal("rc", new IntegerConstant(itemName.length));
    for (int i = 0; i < itemName.length; i++) {
        rec.setVal("i_name_" + i, new VarcharConstant(itemName[i], Type.VARCHAR(24)));
        rec.setVal("i_price_" + i, new DoubleConstant(itemPrice[i]));
    }
    return rec;
}
```

在這裡，我們將itemName、itemPrice（更新完之後的狀態）塞到SpResultrecord的物件中，這個物件會回傳給StoreProcedure.execute()，並在那裡被包進SpResultSet的物件中，透過connection回傳給client端。

As2UpdataItemProc則是實際用來執行Update transaction的地方。其物件透過As2BenchProcFactory來生成。

```
public class As2BenchStoredProcFactory implements StoredProcedureFactory {

    @Override
    public StoredProcedure<?> getStroredProcedure(int pid) {
        StoredProcedure<?> sp;
        switch (As2BenchTxnType.fromProcedureId(pid)) {
        case TESTBED_LOADER:
            sp = new TestbedLoaderProc();
            break;
        case CHECK_DATABASE:
            sp = new As2CheckDatabaseProc();
            break;
        case READ_ITEM:
            sp = new ReadItemTxnProc();
            break;
        case UPDATE_ITEM:
            sp = new As2UpdateItemProc();
            break;
        default:
            throw new IllegalArgumentException("Wrong procedure type");
        }
        return sp;
    }
}
```

此處可以看到，factory根據pid來判斷要生成哪一個procedure物件。（此處的pid是從client端透過connection物件，傳到這裡來的。）原本只有三個case，在此處實作時加上了第四個case UPDATE_ITEM。

As2UpdateItemProc其實作僅有executesql這個method和建構子。
建構子做的事情是生成一個UpdateItemProcParamHelper物件。

```
    public As2UpdateItemProc() {
        super(new UpdateItemProcParamHelper());
    }
```

executesql所做的主要分為兩步驟，第一步驟為根據UpdateItemProcParamHelper準備好的updateItemId，去資料庫中尋找該item之name以及price。第二個步驟則根據第一步驟讀到的price，判斷其是否比MAX_PRICE大，若是則改為MIN_PRICE，否則將其改為原本的price加上updateItemPrice。（此處的第一步驟和第二步驟，是根據每個itemId來做，換句話說，若總共要更新十個item，則會重複步驟一和步驟二十次）

步驟一：尋找該item之name以及price。

```
int iid = paramHelper.getUpdateItemId(idx);
double raise_price = paramHelper.getUpdateItemPrice(idx);
Plan p = VanillaDb.newPlanner().createQueryPlan(
        "SELECT i_name, i_price FROM item WHERE i_id = " + iid, tx);
Scan s = p.open();
```

步驟二：根據price的值，決定如何更新。

```
origin_price = (Double) s.getVal("i_price").asJavaVal();
update_price = origin_price + raise_price;
paramHelper.setItemName("updated_"+(String) s.getVal("i_name").asJavaVal(), idx);
if(origin_price > As2BenchConstants.MAX_PRICE) {
    if(VanillaDb.newPlanner().executeUpdate(
            "UPDATE item SET i_price = " + As2BenchConstants.MIN_PRICE + " WHERE i_id = " + iid, tx) <= 0) {
        throw new RuntimeException("Update Item Price Fail: Could not update item with i_id = " + iid);
    }
    paramHelper.setItemPrice(As2BenchConstants.MIN_PRICE, idx);
}
else {
    if(VanillaDb.newPlanner().executeUpdate(
            "UPDATE item SET i_price = " + update_price + " WHERE i_id = " + iid, tx) <= 0) {
        throw new RuntimeException("Update Item Price Fail: Could not update item with i_id = " + iid);
    }
    paramHelper.setItemPrice(update_price, idx);

}
```

StatisticMgr的修改：
在這個實作中，我將生成csv file的method定義為outputCSVReport。並在outputReort這個method中呼叫。

```
    public synchronized void outputReport() {
        try {
            SimpleDateFormat formatter = new SimpleDateFormat("yyyyMMdd-HHmmss"); // E.g. "20180524-200824"
            String fileName = formatter.format(Calendar.getInstance().getTime());
            if (fileNamePostfix != null && !fileNamePostfix.isEmpty())
                fileName += "-" + fileNamePostfix; // E.g. "20200524-200824-postfix"

            outputDetailReport(fileName);
            outputCSVReport(fileName);
```

csv檔要統計每五秒內，執行的transaction數量，以及之中每一筆執行時間的數據分析。
此處我是在loop中（當讀過的resultset數目小於resultSets中的數目），將resultSets中最前面，且尚未讀過的resultset抽取出來，看他代表的transaction是否有執行成功，若沒有則去resultSets中挑下一個，若有，則將這個resultset的endtime當作一個period的開頭，不斷去resultSets中往後讀取成功commited的resultset的endtime，直到讀出的resultset endtime跟period開頭的時間相減大於五十億奈秒則停止。

```
 if(resultSets.get(counted_txn).isTxnIsCommited()) {
     List<TxnResultSet> in_period_resultSets = new ArrayList<TxnResultSet>();
     in_period_resultSets.add(resultSets.get(counted_txn));
     long period_start = resultSets.get(counted_txn).getTxnEndTime();
     long period_end = resultSets.get(counted_txn).getTxnEndTime();
     counted_txn++;
     while(period_end - period_start <= period_Nanosecs && counted_txn < resultSets.size()){
         if(resultSets.get(counted_txn).isTxnIsCommited()) {
             //in_period_resultSets.add(resultSets.get(counted_txn));
             period_end = resultSets.get(counted_txn).getTxnEndTime();
             if(period_end - period_start > period_Nanosecs) {
                 break;
             }
             else {
                 in_period_resultSets.add(resultSets.get(counted_txn));
                 counted_txn++;
             }

         }
         else {
             counted_txn++;
         }
     }
```

在這段期間被讀取的resultset都會被加到一個arraylist（in_period_resultSets）中。
之後針對in_period_resultSets裡的每一個resultset，去讀取其ResTime，並存到array（restime_in_period）中。此外也將全部的Restime加總起來，assign給totalResTimeUs。

```
long [] restime_in_period = new long[in_period_resultSets.size()];
long totalResTimeUs = 0L;
long avgResTimeUs = 0L;
long minResTimeUs= 0L;
long maxResTimeUs = 0L;
long first_quar_ResTimeUs = 0L;
long median_ResTimeUs = 0L;
long third_quar_ResTimeUs = 0L;
for(int i = 0; i < in_period_resultSets.size(); i++) {
    restime_in_period[i] = in_period_resultSets.get(i).getTxnResponseTime();
    totalResTimeUs += in_period_resultSets.get(i).getTxnResponseTime();
}
Arrays.sort(restime_in_period);
```

呼叫Array.sort(restime_in_period)，將裡面的資料由小到大排列好。
之後便拿著這個sorting好的array，去計算中位數、四分位數以及min跟max。將這些資料計算好後，呼叫writer.write()將這五秒內的結果寫入CSV檔中。（此過程不斷重複直到將ResultSets中的每個resultset都跑過一遍）

```
time_sec += 5;
writer.write(time_sec + ", " + in_period_resultSets.size() + ", " + avgResTimeUs + ", "
        + minResTimeUs + ", " + maxResTimeUs + ", " + first_quar_ResTimeUs + ", " + median_ResTimeUs + ", " + third_quar_ResTimeUs);
writer.newLine();
```

實驗結果與分析（2.4 GHz 四核心Intel Core i5、8GB RAM、512GB SSD、macOS Catalina 10.15.4）

以下是以JDBC mode去執行benchmark的情形

read:100% write:0%

20200419-142710-as2bench

time(sec)	throughput(txs)	avg_latency(us)	min(us)	max(us)	25th_lat(us)	median_lat(us)	75th_lat(us)
5	1216	4112	3477	26185	3703	3877	4223
10	1255	3984	3453	24935	3716	3839	4032
15	1225	4081	3394	24383	3640	3755	3942
20	1272	3930	3487	24249	3703	3801	3980
25	1250	4000	3413	25024	3719	3827	4007
30	1216	4110	3464	23712	3703	3844	4058
35	1217	4110	3432	24477	3764	3912	4217
40	1258	3974	3498	25228	3712	3825	4019
45	1248	4004	3439	24627	3735	3854	4044
50	1243	4022	3498	24007	3733	3855	4060
55	1276	3919	3522	28315	3695	3771	3911
60	1310	3794	3420	25792	3664	3719	3775

```
UPDATE_ITEM - committed: 0, aborted: 0, avg latency: 0 us
READ_ITEM - committed: 14986, aborted: 0, avg latency: 4001 us
TOTAL - committed: 14986, aborted: 0, avg latency: 4001 us
```

read:50% write:50%

20200419-142355-as2bench

time(sec)	throughput(txs)	avg_latency(us)	min(us)	max(us)	25th_lat(us)	median_lat(us)	75th_lat(us)
5	944	5298	3607	24635	4089	5268	5796
10	958	5217	3475	25157	3931	5060	5563
15	940	5320	3474	26039	4054	5164	5747
20	914	5468	3478	28914	4108	5145	5872
25	966	5176	3442	23100	3989	5108	5654
30	968	5164	3497	21352	4031	5098	5688
35	1048	4770	3482	23946	3906	4958	5377
40	1025	4873	3482	21538	3899	4970	5416
45	1099	4552	3448	26121	3712	4829	5124
50	1052	4755	3465	25823	3803	4942	5256
55	1079	4635	3428	23679	3758	4910	5200
60	1103	4508	3409	23124	3723	4862	5119

```
UPDATE_ITEM - committed: 6049, aborted: 0, avg latency: 5758 us
READ_ITEM - committed: 6047, aborted: 0, avg latency: 4157 us
TOTAL - committed: 12096, aborted: 0, avg latency: 4957 us
```

read:0% write:100%

20200419-142047-as2bench

time(sec)	throughput(txs)	avg_latency(us)	min(us)	max(us)	25th_lat(us)	median_lat(us)	75th_lat(us)
5	885	5654	4865	16658	5114	5308	5613
10	829	6028	4867	21197	5205	5500	6352
15	819	6102	4896	19416	5272	5564	6227
20	877	5700	4854	17159	5163	5375	5846
25	832	6011	4866	16877	5268	5539	6253
30	885	5650	4825	20445	5180	5362	5633
35	865	5776	4841	22205	5158	5408	5766
40	786	6362	4854	22333	5352	5765	6706
45	877	5701	4893	15533	5161	5351	5750
50	843	5928	4746	27412	5278	5563	6045
55	869	5753	4907	22827	5233	5433	5873
60	867	5724	4923	19997	5196	5396	5818

```
UPDATE_ITEM - committed: 10234, aborted: 0, avg latency: 5858 us
READ_ITEM - committed: 0, aborted: 0, avg latency: 0 us
TOTAL - committed: 10234, aborted: 0, avg latency: 5858 us
```

分析：可以看到，當benchmark全做read的時候，在同樣時間裡所做的transaction數目最多，而當benchmark全做write時，所做的transaction數目最少。當read/write比例各為50%時，平均每個transaction的latency介於全做read跟全做write之間，且約略等於read avg latency和 write avg latency的平均。這樣的實驗結果符合我們的預期，因為在update中，不只要去讀，還要去寫，所以每個transaction所需時間必定比只做read還要多，也就造成總執行的transaction數量較少。另外，我們也預期當benchmark一半跑read一半跑write時，其執行結果的數據，throughput應在前兩者之間，因為是在同樣的時間內交錯的做read和write，平均每個transaction所需時間比只做read還要多，但比只做write少。明顯的實驗結果亦符合我們的推論。

以下是以SP mode去執行benchmark的情形：

read:100% write:0%

20200419-144157-as2bench

time(sec)	throughput(txs)	avg_latency(us)	min(us)	max(us)	25th_lat(us)	median_lat(us)	75th_lat(us)
5	10514	468	340	2123	418	448	494
10	10171	484	346	2549	434	460	505
15	10257	480	348	4893	432	457	503
20	10833	455	341	4852	399	438	480
25	10851	454	340	3506	392	436	483
30	10081	489	355	3004	436	464	512
35	10241	481	372	2399	434	459	500
40	10554	467	337	48220	407	440	485
45	12272	401	335	2646	368	388	425
50	10867	453	340	5022	424	439	461
55	10881	452	333	40944	425	439	460
60	10931	450	337	2462	426	439	463

```
UPDATE_ITEM - committed: 0, aborted: 0, avg latency: 0 us
READ_ITEM - committed: 128453, aborted: 0, avg latency: 460 us
TOTAL - committed: 128453, aborted: 0, avg latency: 460 us
```

read:50% write:50%

20200419-144610-as2bench

time(sec)	throughput(txs)	avg_latency(us)	min(us)	max(us)	25th_lat(us)	median_lat(us)	75th_lat(us)
5	8538	578	333	4346	444	584	676
10	8519	580	347	4458	445	585	678
15	8825	559	342	5851	425	568	661
20	8845	559	341	2781	438	561	670
25	8607	574	341	4623	444	571	678
30	9397	525	337	2587	395	548	625
35	8479	582	400	5466	447	631	678
40	8653	570	336	3849	442	563	672
45	8563	576	334	48397	443	562	671
50	9657	511	333	2966	385	542	600
55	8722	566	342	2889	442	561	673
60	8602	574	338	33405	444	566	676

```
UPDATE_ITEM - committed: 52704, aborted: 0, avg latency: 684 us
READ_ITEM - committed: 52703, aborted: 0, avg latency: 440 us
TOTAL - committed: 105407, aborted: 0, avg latency: 561 us
```

read:0% write:100%

20200419-144903-as2bench

time(sec)	throughput(txs)	avg_latency(us)	min(us)	max(us)	25th_lat(us)	median_lat(us)	75th_lat(us)
5	6546	755	536	8391	653	689	792
10	6735	734	541	6891	657	690	769
15	7036	703	531	5161	602	660	730
20	6935	713	539	3717	647	680	746
25	7122	694	526	21662	598	659	722
30	6724	736	548	2924	667	696	771
35	6492	762	539	7140	660	692	787
40	6746	733	541	5142	658	686	767
45	6551	755	529	7467	660	693	790
50	6118	800	541	9786	633	706	852
55	7268	680	535	5940	588	638	704
60	6338	780	547	4482	669	719	831

```
UPDATE_ITEM - committed: 80611, aborted: 0, avg latency: 736 us
READ_ITEM - committed: 0, aborted: 0, avg latency: 0 us
TOTAL - committed: 80611, aborted: 0, avg latency: 735 us
```

分析：單就SP mode，以不同read write比例去跑benchmark，其結果和分析狀況和JDBC mode雷同。

比較JDBC和SP：
透過實驗結果可以發現，SP mode的執行速率，比JDBC快了非常多，約略是十倍。會有這個差異主要在於透過JDBC來下sql指令得到結果，其途徑須通過connection、statement、resultset這些remote interface來與server溝通；反之，SP的client只需透過connection、resultset來與server溝通，實際下sql指令是依靠server端的stored procedure，這點差異造成了兩者的執行速度上有了不小的差距。