17 Assignment2 Report

210510210 詹其佐
X1086023 何配瑜

Implement the transaction using JDBC and stored procedures

a. JDBC

Step 1: 在 As2BenchTxnType 裡新增 UPDATE_ITEM
Step 2: 新增並且完成 UpdatePriceParamGen
Step 3: 在 As2BenchJdbcExecutor 裡新增 case: UPDATE_ITEM
Step 4: 實作 UpdatePriceJdbcJob
Step 5: 修改 BenchRte 使之可以利用 READ_WRITE_TX_RATE 來決定要執行 READ or UPDATE

b. Stored procedure

Step 1: 在 UPdatePriceProcParamHelper 生成 10 個 0~5.0 的隨機數字
Step 2: 在 As2BenchStoredProcFactory 新增 case: UPDATE_ITEM
Step 3: 實作 UpdatePriceProc

JDBC

Step 1: 在 As2BenchTxnType 裡新增 UPDATE_ITEM

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

    // Database checking procedures
    CHECK_DATABASE(false),

    // Benchmarking procedures
    READ_ITEM(true),

    // I want to create a new type update_item
    UPDATE_ITEM(true);

    public static As2BenchTxnType fromProcedureId(int pid) {
        return As2BenchTxnType.values()[pid];
    }
```

Step 2: 新增並且完成 UpdatePriceParamGen

```
public class UpdatePriceParamGen 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));

        return paramList.toArray();
    }
```

Step 3: 在 As2BenchJdbcExecutor 裡新增 case: UPDATE_ITEM

```
@Override
public SutResultSet execute(Connection conn, As2BenchTxnType txType, Object[] pars)
        throws SQLException {
    switch (txType) {
    case TESTBED_LOADER:
        return new TestbedLoaderJdbcJob().execute(conn, pars);
    case CHECK_DATABASE:
        return new CheckDatabaseJdbcJob().execute(conn, pars);
    case READ_ITEM:
        return new ReadItemTxnJdbcJob().execute(conn, pars);
    case UPDATE_ITEM:
        //System.out.println("1");
        return new UpdatePriceJdbcJob().execute(conn, pars);
    default:
        throw new UnsupportedOperationException(
                String.format("no JDCB implementation for '%s'", txType));
```

Step 4: 實作 UpdatePriceJdbcJob

Step 4.1: 隨機生成 10 個 0~5.0 的數字，用來看 price 要加多少

```
// I do, generate 10 random price
RandomValueGenerator rvg = new RandomValueGenerator();
double[] price_raise = new double[10];
for (int i = 0; i < 10; i++)
    price_raise[i] = rvg.fixedDecimalNumber(1, 0.0, 5.0);
```

Step 4.2: 把 item 裡面的 price 讀出來

```
// i do, get price from item.i_price
String sql = "SELECT i_price, i_name FROM item WHERE i_id = " + itemIds[i];
rs = statement.executeQuery(sql);
rs.beforeFirst();
if (rs.next()) {
    price = rs.getDouble("i_price");
    //System.out.println(itemIds[i] + ": " + price);
    //outputMsg.append(String.format("'%s', ", rs.getString("i_name")));
} else
    throw new RuntimeException("cannot find the record with i_id = " + itemIds[i]);
//rs.close();
```

Step 4.3: 判斷 price 調整之後，會不會大於 MAX_PRICE，如果大於，就將他變成 MIN_PRICE

```
// i do, check if the price exceeds As2BenchConstants.MAX_PRICE
if(price + price_raise[i] > As2BenchConstants.MAX_PRICE)
    price = As2BenchConstants.MIN_PRICE;
else price += price_raise[i];
```

Step 4.4: 將原本的 id 的 record 裡面的 price 更新成，上面新算出來的 price

```
// i do, UpdatePrice
String sql2 = "UPDATE item SET i_price = " + String.valueOf(price) + " WHERE i_id = "
update_return = statement.executeUpdate(sql2);
if (update_return > 0) {
    outputMsg.append(String.format("'%s', ", rs.getString("i_name")));
    //System.out.println(itemIds[i] + ": " + price);
} else
    throw new RuntimeException("cannot update the record with i_id = " + itemIds[i]);
rs.close();
```

Step 5: 修改 BenchRte 使之可以利用 READ_WRITE_TX_RATE 來決定要執行 READ or UPDATE

Step 5.1: 隨機生成一個數字，決定 As2BenchTxnType 要回傳哪種

```
RandomValueGenerator rvg = new RandomValueGenerator();
double rate = rvg.fixedDecimalNumber(1, 0.0,  10.0);
//System.out.println(rn);
if(rate > As2BenchConstants.READ_WRITE_TX_RATE) {
    //System.out.println(As2BenchConstants.READ_WRITE_TX_RATE);
    //executor = new As2BenchTxExecutor(new UpdatePriceParamGen());
    return As2BenchTxnType.UPDATE_ITEM;
}
else {
    //System.out.println('2');
    //executor = new As2BenchTxExecutor(new As2ReadItemParamGen());
    return As2BenchTxnType.READ_ITEM;
}
```

Step5.2: 如果 type 是 UPDATE_ITEM 那 executor 就 new 一個 UpdatePriceParamGen()

```
if(type == As2BenchTxnType.UPDATE_ITEM) executor = new As2BenchTxExecutor(new UpdatePriceParam
else executor = new As2BenchTxExecutor(new As2ReadItemParamGen());
return executor;
```

Stored procedure

Step 1: 在 UPdatePriceProcParamHelper 生成 10 個 0~5.0 的隨機數字

```
// random number
RandomValueGenerator rvg = new RandomValueGenerator();
for (int i = 0; i < 10; i++)
    price_raise[i] = rvg.fixedDecimalNumber(1, 0.0, 5.0);

for (int i = 0; i < readCount; i++)
    readItemId[i] = (Integer) pars[indexCnt++];
```

Step 2: 在 As2BenchStoredProcFactory 新增 case: UPDATE_ITEM

```
  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 UpdatePriceProc();
      break;
  default:
      throw new IllegalArgumentException("Wrong procedure type");
  }
  return sp;
```

Step 3: 實作 UpdatePriceProc

Step 3.1: 把 item 裡面的 price 讀出來

```
String name;
double price;

int iid = paramHelper.getReadItemId(idx);
Plan p = VanillaDb.newPlanner().createQueryPlan(
        "SELECT i_name, i_price FROM item WHERE i_id = " + iid, tx);
Scan s = p.open();
s.beforeFirst();
if (s.next()) {
    name = (String) s.getVal("i_name").asJavaVal();
    price = (Double) s.getVal("i_price").asJavaVal();
    //System.out.println(name + ": " + price);
} else
    throw new RuntimeException("Cloud not find item record with i_id = " + iid);
s.close();
```

Step 3.2: 判斷 price 調整之後，會不會大於 MAX_PRICE，並把原本的 price 更新

```
if(price + paramHelper.getPriceRaise(idx) > As2BenchConstants.MAX_PRICE)
    price = As2BenchConstants.MIN_PRICE;
else price += paramHelper.getPriceRaise(idx);

int p1 = VanillaDb.newPlanner().executeUpdate(
        "UPDATE item SET i_price = " + String.valueOf(price) + " WHERE i_id = " + iid, tx);
```

Experiments:

CSV Report:
以微秒為單位，以期將數據差異表現更加明顯。
下圖為 store procedures 在 50% read, 50% update 的結果。

	A	B	C	D	E	F	G	H
1	time(sec)	throughput(txs)	avg_latency(Micros)	min(Micros)	max(Micros)	25th_lat(Micros)	median_lat(Micros)	75th_lat(Micros)
2	5	998	4979	1576	15639	3708	4753	5849
3	10	869	11427	1638	26322	4719	5192	6053
4	15	924	16118	1540	27429	4460	4822	5902
5	20	1004	19780	1540	20823	3320	4737	5886
6	25	999	24848	1575	17578	3577	4728	5849
7	30	968	30767	1531	24463	3941	4771	5911
8	35	958	36270	1538	25972	4365	4772	5870
9	40	1038	38252	1445	14243	3363	4608	5748
10	45	1106	40387	1508	28191	3197	3824	5172
11	50	1029	48233	1444	14051	3582	4651	5683
12	55	1115	48967	1507	27095	3084	4106	5305

Experiment environment:
Intel® Core™ i5-8250U CPU @ 1.60GHz × 8, 8GB RAM, 140 GB HDD, Ubuntu 18.04.3

Performance Comparation:
a. JDBC and store procedures:

(1) JDBC:

```
READ_ITEM - committed: 2606, aborted: 0, avg latency: 9 ms
UPDATE_ITEM - committed: 2620, aborted: 0, avg latency: 13 ms
TOTAL - committed: 5226, aborted: 0, avg latency: 11 ms
```

(2) store procedures:

```
UPDATE_ITEM - committed: 5939, aborted: 0, avg latency: 5 ms
READ_ITEM - committed: 6074, aborted: 0, avg latency: 4 ms
TOTAL - committed: 12013, aborted: 0, avg latency: 5 ms
```

b. Different ratio:

(1) 50% ReadItemTxn, 50% UpdateItemPriceTxn:
JDBC:

```
READ_ITEM - committed: 2606, aborted: 0, avg latency: 9 ms
UPDATE_ITEM - committed: 2620, aborted: 0, avg latency: 13 ms
TOTAL - committed: 5226, aborted: 0, avg latency: 11 ms
```

Store procedures:

```
UPDATE_ITEM - committed: 5939, aborted: 0, avg latency: 5 ms
READ_ITEM - committed: 6074, aborted: 0, avg latency: 4 ms
TOTAL - committed: 12013, aborted: 0, avg latency: 5 ms
```

(2) 20% ReadItemTxn, 80% UpdateItemTxn:

JDBC:

```
READ_ITEM - committed: 980, aborted: 0, avg latency: 10 ms
UPDATE_ITEM - committed: 3785, aborted: 0, avg latency: 13 ms
TOTAL - committed: 4765, aborted: 0, avg latency: 13 ms
```

Store procedures:

```
UPDATE_ITEM - committed: 9781, aborted: 0, avg latency: 5 ms
READ_ITEM - committed: 2576, aborted: 0, avg latency: 3 ms
TOTAL - committed: 12357, aborted: 0, avg latency: 5 ms
```

(3) 1% ReadItemTxn, 99% UpdateItemTxn:
JDBC:

```
READ_ITEM - committed: 54, aborted: 0, avg latency: 11 ms
UPDATE_ITEM - committed: 4000, aborted: 0, avg latency: 14 ms
TOTAL - committed: 4054, aborted: 0, avg latency: 15 ms
```

Store procedures:

```
UPDATE_ITEM - committed: 10138, aborted: 0, avg latency: 5 ms
READ_ITEM - committed: 107, aborted: 0, avg latency: 4 ms
TOTAL - committed: 10245, aborted: 0, avg latency: 6 ms
```

Analysis and explanation:

由以上的數據對比可見，store procedures 的速度明顯較 JDBC 快上許多。執行的項目約為JDBC的兩倍，而 latency 大約少了一半。這部份我們認為是 store procedures 已經把 query 寫好預存於系統中，可以直接執行，因此可以較JDBC快上許多。而在不同作業系統上執行時也會有不同的結果，如 Windows，他在執行時的latency 幾乎不會超過 1ms，由此得知，作業系統的差異也會使速度產生不同的變化。

碰到的問題&解決的方法

1. 在剛開始的時候，因為不明原因，占用到 port: 1099 導致 server 跑不起來

Solution:
將 org.vanilladb.bench.server 的 JdbcStartUp.startUp(1099) 改成JdbcStartUp.startUp(1100)

2. 無法宣告 UPDATE_ITEM

```
// Loading procedures
TESTBED_LOADER(false),

// Database checking procedures
CHECK_DATABASE(false),

// Benchmarking procedures
READ_ITEM(true);

// I want to create a new type update_item
UPDATE_ITEM(true);
```

Solution: 因為我不知道 java 的 enumeration 語法，後來經過助教提點，查了之後才知道裡面是用逗號隔開，最後再加分號

3. 無法控制 READ_WRITE_TX_RATE

Solution: 發現沒有去改到 executor 後來利用回傳的 type 來判斷要產生哪個 executor 就可以了