Cloud Databases Assignment 1

Group 2
102061105 Yun-Sheng, Chang
102061212 Jian-Hao, Huang

March 15, 2017

1 Implementation

1.1 Update Transaction

First, we define a new transaction *UPDATE_TXN*.

```
1  public enum TransactionType {
2    SCHEMA_BUILDER, TESTBED_LOADER, SAMPLE_TXN, UPDATE_TXN
3  }
```

TransactionType.java

For the new item prices, we generate additional numbers (on the client side) with *RandomValueGenerator* and append them to the *paramList*.

```
1  public Object[] generateParameter() {
2    RandomValueGenerator rvg = new RandomValueGenerator();
3    LinkedList<Object> paramList = new LinkedList<Object>();
4    for (int i = 0; i < 10; i++)
5      paramList.add(rvg.number(1, TpccConstants.NUM_ITEMS));
6    for (int i = 0; i < 10; i++)
7      paramList.add(rvg.nextDouble());
8    return paramList.toArray();
9  }
```

UpdateTxnParamGen.java

1.1.1 JDBC

A new method *getJdbcExecutor* is provided for the *JdbcRte* to randomly choose a JDBC executor. The value, *probOfUpdateJdbc*, determines the probability of choosing a *JdbcUpdateTxnExecutor*.

```
1  private JdbcTxnExecutor getJdbcExecutor() {
2    RandomValueGenerator rvg = new RandomValueGenerator();
3    JdbcTxnExecutor txnExecutor = null;
4    int probOfUpdateJdbc = 50;
5    int random = rvg.number(0, 99);
6
7    if (random < probOfUpdateJdbc)
8      txnExecutor = new JdbcUpdateTxnExecutor();
9    else
10     txnExecutor = new JdbcSampleTxnExecutor();
11
12   return txnExecutor;
13 }
14
15 public TxnResultSet executeTxnCycle() {
16   JdbcTxnExecutor txnExecutor = null;
17   txnExecutor = getJdbcExecutor();
18   return txnExecutor.execute();
19 }
```

JdbcRte.java

In order to implement the new transaction (select 10 items and update their prices), we create a new class *JdbcUpdateTxnExecutor*. Like every JDBC transaction executor, it first prepares the parameters which will be used to form a complete query statement later.

```
protected void prepareParams() {
  TxParamGenerator pg = new UpdateTxnParamGen();
  for (int i = 0; i < 10; i++) {}
    itemId[i] = (Integer) pg.generateParameter()[i];
    itemPrice[i] = (Double) pg.generateParameter()[i + 10];
  }
}
```

JdbcUpdateTxnExecutor.java

The main task is defined in the method *executeSql*. While the *SELECT* query part is identical to *JdbcSampleTxnExecutor*, two more lines are added to update the items' price. We invoke *executeQuery* to do *SELECT* query and invoke *executeUpdateQuery* to do *UPDATE* query.
One problem we met here is that if we do not transform double to string with *String.format*, in some cases, the number will be expressed in scientific notation (e.g., 4E-4) and incurs a *BadSyntaxException*. Thus, we manually do the transformation.

```
public void executeSql() {
  for (int i = 0; i < 10; i++) {
    // do SELECT query

    sql = "UPDATE item SET i_price = " +
        String.format("%.8f", itemPrice[i]) +
        " WHERE i_id = " + itemId[i];
    JdbcService.executeUpdateQuery(sql, stm);
  }
}
```

JdbcUpdateTxnExecutor.java

1.1.2 Stored Procedure

We define a new RTE *StoredProcRte* to randomly do *SampleTxn* and *UpdateTxn* using stored procedure. Like JDBC, the value *probOfUpdateTxn* determines the probability of choosing a *UpdateTxnExecutor*.

```
public TransactionExecutor getTxnExecutor() {
  RandomValueGenerator rvg = new RandomValueGenerator();
  int random = rvg.number(0, 99);
  int probOfUpdateTxn = 50;

  if (random < probOfUpdateTxn)
    return new UpdateTxnExecutor(updateParamGen);
  else
    return new SampleTxnExecutor(sampleParamGen);
}

protected TxnResultSet executeTxnCycle() {
  //TransactionExecutor tx = new SampleTxnExecutor(paramGem);
  TransactionExecutor tx = getTxnExecutor();
  return tx.execute(conn);
}
```

StoredProcRte.java

A new class *UpdateTxnProc* is defined to do 10 *SELECT* queries and 10 *UPDATE* queries. Unlike JDBC, this class belongs to the server side and naturally its method is executed by the server rather than the client.

```
protected void executeSql() {
  // do SELECT query

  sql = "UPDATE item SET i_price = " +
      String.format("%.8f", paramHelper.getItemPrice(i)) +
      " WHERE i_id = " + paramHelper.getItemId(i);
  VanillaDb.newPlanner().executeUpdate(sql, tx);
}
```

UpdateTxnProc.java

1.2 Statistic Manager

Previous implementation of *StatisticMgr* maintains a list of *TxnResultSet* which consists of the transaction results generated by every RTEs. However, since we want to know the result within each time interval, we instead maintain a list of list of *TxnResultSet*. Every list of *TxnResultSet* now stores the transaction results generated by every RTEs within a time interval.

```
//private List<TxnResultSet> resultSets;
private List<List<TxnResultSet>> resultSetsList;
```

StatisticMgr.java

The *Benchmarker* has a new task: add a new list of *TxnResultSet* to *resultSetsList* every 5 seconds.

```
public void run() {
  long timeLeft = TestingParameters.BENCHMARK_INTERVAL;
  for (int i = 0; i < TestingParameters.BENCHMARK_INTERVAL / 5000; i++){
    timeLeft -= 5000;
    Thread.sleep(5000);
    statMgr.addResultSetList();
  }
  if(timeLeft > 0)
    Thread.sleep(timeLeft);

  // benchmark finished
  for (int i = 0; i < emulators.length; i++)
    emulators[i].stopBenchmark();
}
```

Benchmarker.java

Besides adding a new list, we must also record which list is the newest. Thus, we add an attribute *setID* to *StatisticMgr* so that every transaction results can be put into the newest list. The add-to-list task must be done carefully or some synchronization issues may occur. We make the method *addResultSetList synchronized* to guarantee the atomicity of the two operations: adding a list and increase *setID*.

```
private int setID;

public synchronized void addResultSetList(){
  List<TxnResultSet> resultSets = new LinkedList<TxnResultSet>();
  resultSetsList.add(resultSets);
  setID++;
}
```

StatisticMgr.java

Since only the committed transactions should be considered, we add two more lists to store the results of committed transactions. The first list, *committedTxnsList*, is similar to the *resultSetsList* except that all the results of uncommitted transactions are removed. The second list, *allCommittedTxns*, includes all the results of committed transactions during the entire benchmarking period.

```
public synchronized void outputReport() {
  HashMap<TransactionType, TxnStatistic> txnStatistics = new HashMap<TransactionType,
    TxnStatistic>();
  txnStatistics.put(TransactionType.SAMPLE_TXN, new TxnStatistic(
      TransactionType.SAMPLE_TXN));
  txnStatistics.put(TransactionType.UPDATE_TXN, new TxnStatistic(
      TransactionType.UPDATE_TXN));
  try {
    // only considering the committed transactions
    List<List<TxnResultSet>> committedTxnsList = new ArrayList<List<TxnResultSet>>();
    List<TxnResultSet> allCommittedTxns = new LinkedList<TxnResultSet>();
    for(List<TxnResultSet> resultSets : resultSetsList){
      List<TxnResultSet> tempSets = new LinkedList<TxnResultSet>();

      for (TxnResultSet resultSet : resultSets) {
        if (resultSet.isTxnIsCommited()){
          tempSets.add(resultSet);
          allCommittedTxns.add(resultSet);
        }
      }
      committedTxnsList.add(tempSets);
```

```
    }
  }
  // evaluate the transaction results and show the final result
  ...
}
```

StatisticMgr.java

With the committed transaction lists, we're now able to iterate through the lists and calculate the final statistic result.

```
public synchronized void outputReport() {
  try {
    // form a committed lists
    ...

    // write total transaction count
    newBwrFile.write("time(sec), throughput(txs), avg_latency(ms), min(ms), max(ms),"
        + " 25th_lat(ms), median_lat(ms), 75th_lat(ms)");
    newBwrFile.newLine();
    / read all txn resultset
    int timePerTxn, numOfTxn = 0, totalTime;
    List<Integer> txnList = new ArrayList<>();
    int i = 1;
    for(List<TxnResultSet> committedTxns : committedTxnsList){
      if (!committedTxns.isEmpty()) {
        totalTime = 0;
        numOfTxn = 0;
        txnList.clear();
        for (TxnResultSet resultSet : committedTxns){
          timePerTxn = (int) TimeUnit.NANOSECONDS.toMillis(resultSet.getTxnResponseTime());
          txnList.add(timePerTxn);
          totalTime += timePerTxn;
          numOfTxn++;
        }
        Collections.sort(txnList);
        newBwrFile.write(i * 5 + 60 + ", " + numOfTxn + ", "
            + totalTime / numOfTxn + ", "
            + txnList.get(0) + ", "
            + txnList.get(txnList.size()-1) + ", "
            + txnList.get(txnList.size()/4) + ", "
            + txnList.get(txnList.size()/2) + ", "
            + txnList.get(txnList.size()*3/4));
        newBwrFile.newLine();
        i++;
      }
    }
    newBwrFile.close();
  }
}
```

StatisticMgr.java

2 Experiment

2.1 Environment

Intel Core i7-4790 CPU @ 3.60 GHz
32 GB RAM
1TB HDD
Ubuntu Linux 16.04

2.2 Result and Analysis

All the following experiments are run within a 1-minute warm up followed by a 3-minute benchmarking.

2.2.1 JDBC versus Stored Procedure

We first compare the performance of JDBC and stored procedure.
We fix the transaction ratio at 50% *SampleTxn* and 50% *UpdateTxn*. We obtain the minumum, maximum and

quartile of transaction latency with our new *StatisticMgr*.

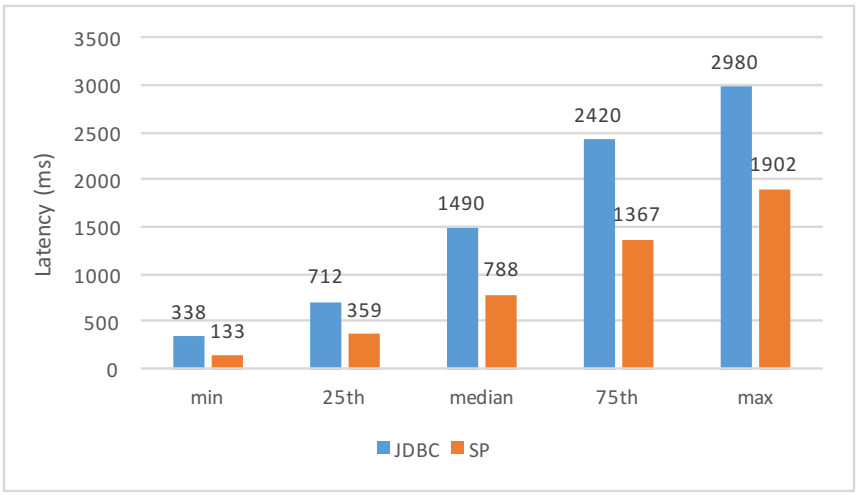

Figure 1: Latency comparison between JDBC and stored procedure

The throughput is defined as the total number of both transactions during the benchmarking period.

	JDBC	stored procedure
Throughput (txns)	2307	4156

Table 1: Throughput comparison between JDBC and stored procedure

It's clear that JDBC has a higher transaction latency and lower throughput compared to stored procedure. We give two possible reasons:

1. JDBC has more communication overhead;
2. JDBC has an additional layer overhead while stored procedure directly talk to the native interface.

2.2.2 Transaction Ratio

Next, we want to know the performance impacts with different transaction ratio. But before that, we compare the average latency of *SampleTxn* and *UpdateTxn*.
This experiment is conducted at the 50/50 ratio using both JDBC and stored procedure.

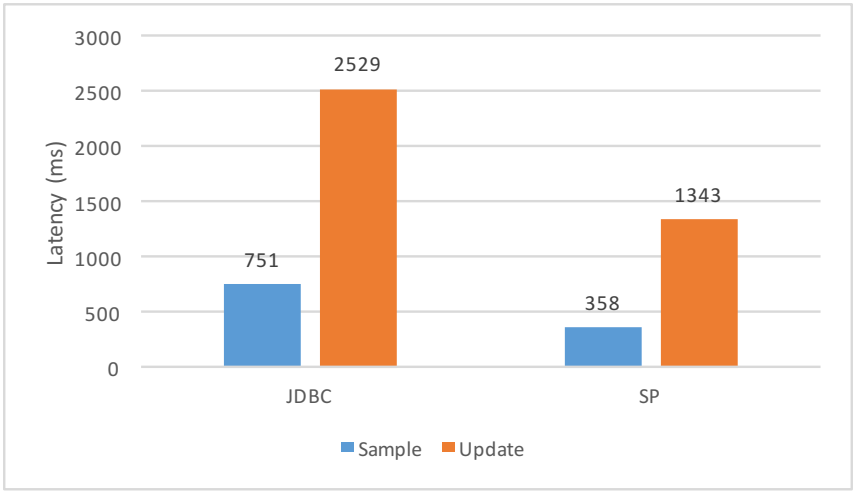

Figure 2: Latency comparison between different transaction

Regardless of which interface we use, *UpdateTxn* has much higher latency compared to *SampleTxn*.
Further, we run both the JDBC and the stored procedure with three different transaction ratios (0/100, 50/50 and 100/0). Numbers in the bottom of the figure (0, 50, 100) denote the proportion of *SampleTxn*. The throughput is defined as the total number of both transactions during the benchmarking period.

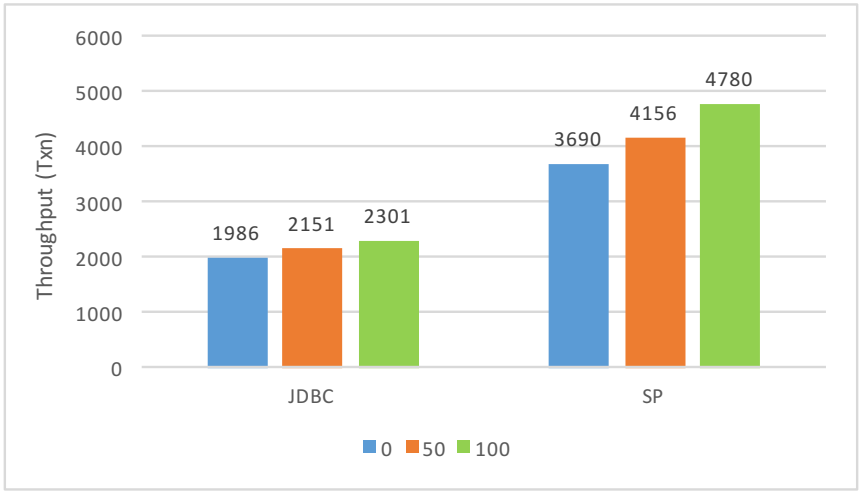

Figure 3: Throughput comparison between different transaction ratios

The throughput does increase as the proportion of *SampleTxn* increases, however, the relation is less obvious compared to the large difference in latency.

2.2.3 Number of RTEs

We tune the number of RTEs and observe the performance difference.
The experiment is conducted using JDBC and only the *UpdateTxn*s are invoked. We set the number of RTEs to 5, 20 and 40 and plot the throughput and the latency versus number of RTEs in Figure 4.

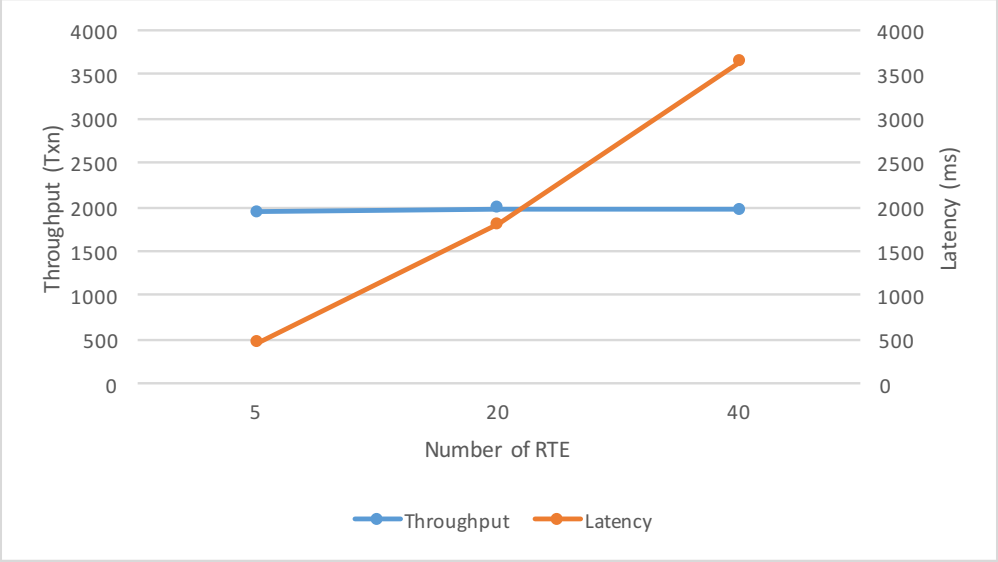

Figure 4: Throughput and average latency with different number of RTEs

The latency almost linearly increase with the number of RTEs while the throughput reamins the same. We think the reasons behind this might be:

1. the throughput remains the same because the database system's utilization is too high to serve more requests concurrently;

2. since the database is not able to serve more requests concurrently, increasing the number of RTEs simply increase the number of queued requests and thus increase the latency of each request.

2.2.4 Bufferpool Size

We also tune the size of the bufferpool.
The experiment is conducted using JDBC and only the *UpdateTxn*s are invoked. The bufferpool size is set to

3200, 6400, 12800, 51200 and 102400 KB (actually we're not sure about the unit since it's not described in the comment).

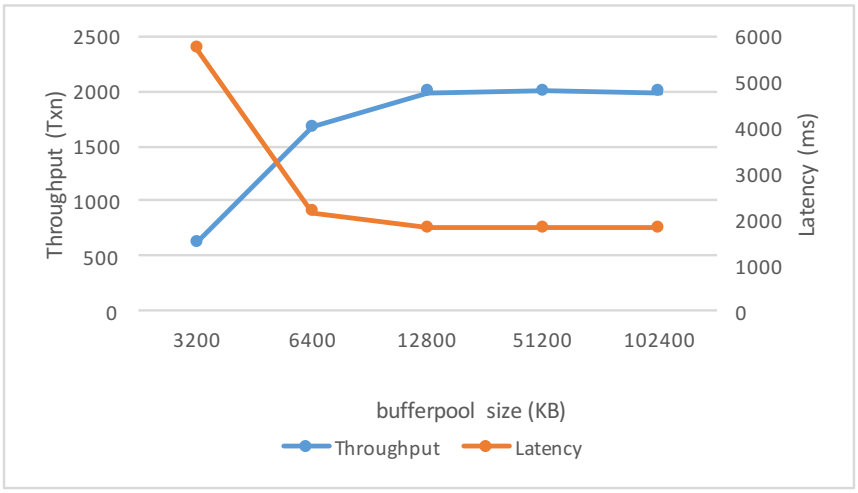

Figure 5: Throughput and average latency with different bufferpool size

Even if we decrease the bufferpool size to 12800 KB, there's still no performance penalty. However, further decreasing the bufferpool size to 6400 KB or even 3200 KB, the performance significantly drops. We think it's because the dataset is fully fitted in 12800 KB but is not fully fitted in a smaller bufferpool.

2.2.5 CSV Screenshot

	A	B	C	D	E	F	G	H
1	time(sec)	throughput(tx	avg_latency(1	min(ms)	max(ms)	25th_lat(ms)	median_lat(m	75th_lat(ms)
2	65	114	875	233	1930	367	500	1487
3	70	116	861	142	1929	375	534	1394
4	75	116	862	99	1739	317	1085	1286
5	80	110	924	133	1804	376	1077	1419
6	85	117	845	133	1846	359	960	1269
7	90	118	864	158	1913	359	852	1394
8	95	118	828	125	1754	342	1003	1236
9	100	118	860	217	1787	368	518	1386
10	105	115	857	183	1679	333	1011	1336
11	110	117	861	108	2104	400	551	1395
12	115	118	828	150	1862	350	843	1287
13	120	116	884	175	1787	350	1028	1411
14	125	108	886	100	1846	359	944	1461
15	130	113	916	133	2112	359	893	1394
16	135	122	821	133	1695	359	777	1311
17	140	122	821	83	1696	284	952	1221
18	145	113	876	134	2005	392	869	1311
19	150	120	851	142	1987	359	919	1295

sp-50-50-20-102400

Figure 6: CSV file generated by new StatisticMgr

3 Code Tracing

The code tracing part is for our own reference.

3.1 Create a RTE

Entry point of the benchmark client program.

```java
public static void main(String[] args) {
  Benchmarker b = new Benchmarker();

  switch (action) {
  case 2:
    // Run benchmarks
    b.run();
    b.report();
    break;
  }
}
```

App.java

Get RTEs of either JDBC (*JdbcRte*) or stored procedure (*SampleTxnRte*).
RemoteTerminalEmulator extends *Thread*. Each *emulators[i]* will run as an individual thread.

```java
public void run() {
  // Initialize RTEs
  RemoteTerminalEmulator[] emulators = new RemoteTerminalEmulator[TestingParameters.NUM_RTES];
  for (int i = 0; i < emulators.length; i++) {
    emulators[i] = getRte(i);
  }
}

private RemoteTerminalEmulator getRte(Object... args) {
  RemoteTerminalEmulator rte;
  if(TestingParameters.CONNECTION_MODE == TestingParameters.ConnectionMode.JDBC)
    rte = new JdbcRte(args);
  else
    rte = new SampleTxnRte(args);
  return rte;
}

public void run() {
  // start each RTEs
  for (int i = 0; i < emulators.length; i++) {
    emulators[i].start();
  }
}
```

Benchmarker.java

executeTxnCycle() is an *abstract* method defined by *JdbcRte* and *SampleTxnRte*.
The benchmark results are stored in *rs*.

```java
public void run() {
  while (!stopBenchmark) {
    TxnResultSet rs = executeTxnCycle();
    if (!isWarmingUp)
      statMgr.processTxnResult(rs);
  }
}
```

RemoteTerminalEmulator.java

3.2 JDBC

```java
public TxnResultSet executeTxnCycle() {
  txnExecutor = new JdbcSampleTxnExecutor();
  return txnExecutor.execute();
}
```

JdbcRte.java

JdbcTxnExecutor is a template for the JDBC executor.
One should implement *executeSql()* as a basic task unit. The basic task unit will be repeatedly performed several times.

```java
public TxnResultSet execute() {
  // send txn request and start measure txn response time
  long txnRT = System.nanoTime();
  executeSql();
  SutResultSet result = new VanillaDbResultSet(createResultSet());
  // measure txn response time
  txnRT = System.nanoTime() - txnRT;
}

protected abstract void executeSql();
```

JdbcTxnExecutor.java

An implementation of ten *SELECT* queries.
The auto-committing property is disabled since we want to perform the ten queries in one transaction.
TODO: what is statement?

```java
public void executeSql() {
  conn = JdbcService.connect();
  try {
    conn.setAutoCommit(false);
    Statement stm = JdbcService.createStatement(conn);
    ResultSet rs = null;
    for (int i = 0; i < 10; i++) {
      String sql = "SELECT i_name FROM item WHERE i_id = "
          + itemId[i];
      rs = JdbcService.executeQuery(sql, stm);
      rs.beforeFirst();
      if (rs.next()) {
        itemName[i] = rs.getString("i_name");
      }
      rs.close();
    }
    conn.commit();
    isCommitted = true;
  } finally {
    JdbcService.disconnect(conn);
  }
}
```

JdbcSampleTxnExecutor.java

3.3 Stored Procedure

```java
protected TxnResultSet executeTxnCycle() {
  TransactionExecutor tx = new SampleTxnExecutor(paramGem);
  return tx.execute(conn);
}
```

SampleTxnRte.java

Similar to JDBC, stored procedure also has a template, *TransactionExecutor*.

```java
public abstract TxnResultSet execute(SutConnection conn);
```

TranctionExecutor.java

An implementation of the stored procedure, which does the following things:

1. prepare the parameters to be sent to the server;
2. invoke the stored procedure;
3. form the result.

There's not much to be done on the client side for stored procedure as the acutal tasks are defined on the server side.

```java
public TxnResultSet execute(SutConnection conn) {
  try {
    TxnResultSet rs = new TxnResultSet();
```

```
    rs.setTxnType(pg.getTxnType());
    // generate parameters
    Object[] params = pg.generateParameter();

    // send txn request and start measure txn response time
    long txnRT = System.nanoTime();
    SutResultSet result = callStoredProc(conn, params);
    // measure txn response time
    txnRT = System.nanoTime() - txnRT;

    // display output
    rs.setTxnIsCommited(result.isCommitted());
    rs.setOutMsg(result.outputMsg());
    rs.setTxnResponseTimeNs(txnRT);
    return rs;
  }
}
```

SampleTxnExecutor.java

Unlike JDBC, stored procedure only obtain two informations: whether the transaction is committed and output message formed by the server.

```
protected SutResultSet callStoredProc(SutConnection spc, Object[] pars) {
  try {
    SutResultSet result = spc.callStoredProc(pg.getTxnType().ordinal(),
        pars);
    return result;
  }
}
```

TranctionExecutor.java