Assignment2 report

Group members

1. 107062321 王劭元
2. 107062214 陳伯瑾
3. 107062228 陳劭愷

Implementation

Same part between JDBC and SP

1. `org.vanilladb.bench.benchmarks.as2.rte.As2UpdateItemPriceParamGen.java`

   為了要給update item price這個 transaction 一些參數，所以我們新增的這個檔案，準備一些 parameters。在這個檔案裡總共有兩個 function：

   - getTxnType()

     這個 function 就是很簡單的把這個 transaction 的 type 回傳回去。

   - generateParameter()

     這個 function 就是負責準備 parameters。首先我們先把 TOTAL_READ_COUNT（由於每次要read 以及要 update 的數量是一樣的，所以我們就共用同一個變數）add 在 paramList 的最前面。接下來總共會add TOTAL_READ_COUNT*2個東西進paramList，每次都先 add 一個隨機的id，並選擇一個在 0.0 ~ 5.0 之間隨機的數字，代表要增加的 price。

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

  // Set read count
  paramList.add(TOTAL_READ_COUNT);
  // Set read id and update price
  for (int i = 0; i < TOTAL_READ_COUNT; i++){
    paramList.add(rvg.number(1, As2BenchConstants.NUM_ITEMS));
    double d = Math.round(rvg.randomDoubleIncrRange(0.0, 5.01, 0.1) * 10) / 10.0;
    paramList.add(d);
  }

  return paramList.toArray(new Object[0]);
}
```

2. `org.vanilladb.bench.server.param.as2.UpdateItemPriceProcParamHelper.java`

   在這個檔案裡面，最重要的部分就是我們要去prepare parameters，也就是把parameter generator產生的parameters parse出來，之後可以讓我們在執行transaction job的時候可以跟parameter helper拿到變數

   - prepareParameters(Object... pars)
     1. 在這邊我們先拿出我們的updateCount（共要update幾個item）
     2. allocate空間給之後會用的的data structure
     3. 接著就輪流取出要update 的 item 的 id，以及他要被update多少price。

```java
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];
  itemName = new String[updateCount];
  itemPrice = new double[updateCount];
  updatePrice = new double[updateCount];

  for (int i = 0; i < updateCount; i++){
    updateItemId[i] = (Integer) pars[indexCnt++];
    updatePrice[i] = (double) pars[indexCnt++];
  }
}
```

JDBC

1. `org.vanilladb.bench.benchmarks.as2.As2BenchTransactionType.java`

   在 Class `As2BenchTransactionType` 中新增一個新的 transaction type，叫做 `UPDATE_ITEM_PRICE`。`true` 代表這個新的 transaction type 是用來做 benchmark 的。

```
// Benchmarking procedures
UPDATE_ITEM_PRICE(true);
```

2. `org.vanilladb.bench.benchmarks.as2.rte.jdbc.As2BenchJdbcExecutor.java`

   在這邊，由於我們新增了一個 transaction type，所以在這邊switch case的時候，也必須多補我們新增的 tx type上去。

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

3. `org.vanilladb.bench.benchmarks.as2.rte.jdbc.UpdateItemPriceTxnJdbcJob.java`

   在這個檔案中，大致上的邏輯與ReadItemTxnJdbcJob差不多，只是下的sql不一樣，以及需要去update。

   - Execute Logic：
     1. 先create一個statement
     2. 跑一個for迴圈，次數是我們要更新的item個數
     3. 去跟parameter helper拿我們這次要更新的item id，以及要update的price
     4. 下一個sql `SELECT i_price FROM item WHERE i_id = " + iid` 去拿到這個 id 目前的price
     5. 檢查old price加上要update的price後，會不會超過上限，會的話就把new price設為min，不會的話new price就等於 old price + update price
     6. 下一個sql `UPDATE item SET i_price = " + itemNewPrice + " WHERE i_id = " + iid` 把這個id的price更新
     7. 最後commit這個transaction

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

// SELECT
for (int i = 0; i < paramHelper.getUpdateCount(); i++) {
  int iid = paramHelper.getUpdateItemId(i);
  double addPrice = paramHelper.getUpdatePrice(i);
  String sql = "SELECT i_price FROM item WHERE i_id = " + iid;
  rs = statement.executeQuery(sql);
  rs.beforeFirst();
  if (rs.next()) {
    double price = rs.getDouble("i_price");
    double itemNewPrice = price + addPrice;
    if (itemNewPrice > As2BenchConstants.MAX_PRICE)
      itemNewPrice = As2BenchConstants.MIN_PRICE;
    String sql_update = "UPDATE item SET i_price = " + itemNewPrice + " WHERE i_id = " + iid;
    statement.executeUpdate(sql_update);
    outputMsg.append(String.format("item %d's price change from %f to %f ! ", iid, price, itemNewPrice));
  } else
    throw new RuntimeException("cannot find the record with i_id = " + iid);
  rs.close();
}

conn.commit();
```

Stored Procedure

1. `org.vanilladb.bench.benchmarks.as2.As2BenchTransactionType.java`

在 Class `As2BenchTransactionType` 中新增一個新的 transaction type，叫做 `UPDATE_ITEM_PRICE`。`true` 代表這個新的 transaction type 是用來做 benchmark 的。

2. `org.vanilladb.bench.server.procedure.as2.As2BenchStoredProcFactory.java`

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

在 Class `As2BenchStoredProcFactory` 中定義了不同的 procedure id 要呼叫哪一個 transaction 執行，因此我們新增一個 `UPDATE_ITEM_PRICE` 的 case，使其呼叫一個新的 transaction `UpdateItemPriceTxnProc`。

3. `org.vanilladb.bench.server.procedure.as2.UpdateItemPriceTxnProc.java`

   在 `UpdateItemPriceTxnProc` 中，首先用外面傳進來的參數建立一個 Class `UpdateItemPriceProcParamHelper` 叫做 `paramHelper`，接著從 `paramHelper` 中每次拿出一個 `id` 進行更新，因為我們需要原本的 price，因此先用 `SELECT i_price FROM item where id=...` 拿到原本的 price，接著就隨機的把 price 增加 0.0 ~ 5.0，若 price 超過 MAX_PRICE=100，則將其設為 MIN_PRICE=0，最後執行 SQL `UPDATE item SET i_price = itemNewPrice WHERE id=...`。

   因為是 Stored Procedure，並且我們想把這兩句 SQL 看成一個 transaction，因此利用 Class `StoredProceduerHelper` 的 `executeQuery` 以及 `executeUpdate` 來執行上面的 SQL。

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

// SELECT
for (int idx = 0; idx < paramHelper.getUpdateCount(); idx++) {
  int iid = paramHelper.getUpdateItemId(idx);
  double addPrice = paramHelper.getUpdatePrice(idx);
  Scan s = StoredProcedureHelper.executeQuery(
    "SELECT i_price FROM item WHERE i_id = " + iid,
    tx
  );
  s.beforeFirst();
  if (s.next()) {
    double price = (Double) s.getVal("i_price").asJavaVal();
    double itemNewPrice = price + addPrice;
    if (itemNewPrice > As2BenchConstants.MAX_PRICE)
      itemNewPrice = As2BenchConstants.MIN_PRICE;
    StoredProcedureHelper.executeUpdate(
      "UPDATE item SET i_price = " + itemNewPrice + " WHERE i_id = " + iid,
      tx
    );
  } else {
    throw new RuntimeException("Cloud not find item record with i_id = " + iid);
  }
  s.close();
}
```

StatisticMgr

- 這裡我們特別提一下，因為我們在實作時，我們選擇把transaction歸類在第幾個5秒interval內的方法是，等他commit後，才去看他屬於哪個interval。也就是說，以他commit的時間為主。

  所以如果有一些transaction在很靠近60秒時開始執行，但在60秒後結束，這樣會導致他被歸類在65秒的這個區間。因此若我們的report有看到65秒的時候有commit少量transaction，這種情況是正常的!

Screenshot of CSV Report

- output of stored procedure with `READ_WRITE_TX_RATE=0.5`

time(sec)	throughput(txs)	avg_latency(ms)	min(ms)	max(ms)	25th_lat(ms)	median_lat(ms)	75th_lat(ms)
5	13802	0	0	18	0	0	0
10	14005	0	0	4	0	0	0
UPDATE_ITEM_PRICE: ABORTED							
15	13974	0	0	9	0	0	0
20	14051	0	0	7	0	0	0
UPDATE_ITEM_PRICE: ABORTED							
25	13985	0	0	2	0	0	0
30	14023	0	0	3	0	0	0
35	13863	0	0	24	0	0	0
40	13982	0	0	2	0	0	0
45	13983	0	0	2	0	0	0
UPDATE_ITEM_PRICE: ABORTED							
50	13915	0	0	8	0	0	0
55	13928	0	0	6	0	0	0
60	13892	0	0	21	0	0	0
READ_ITEM - committed: 83581	aborted: 0	avg latency: 0 ms					
UPDATE_ITEM_PRICE - committed: 83822	aborted: 3	avg latency: 0 ms					
TOTAL - committed: 167403	aborted: 3	avg latency: 1 ms					

Experiments

Experiment Environment

Intel i5-8365 @ 2.4GHz, 16GB RAM, 512GB SSD, macOS Big Sur 11.2.3

Performance comparison

- JDBC `READ_WRITE_TX_RATE=0.25`

READ_ITEM - committed: 5721	aborted: 0	avg latency: 4 ms
UPDATE_ITEM_PRICE - committed: 17030	aborted: 1	avg latency: 5 ms
TOTAL - committed: 22751	aborted: 1	avg latency: 5 ms

- JDBC `READ_WRITE_TX_RATE=0.5`

READ_ITEM - committed: 11804	aborted: 0	avg latency: 4 ms
UPDATE_ITEM_PRICE - committed: 11734	aborted: 1	avg latency: 5 ms
TOTAL - committed: 23538	aborted: 1	avg latency: 5 ms

- JDBC `READ_WRITE_TX_RATE=1.0`

READ_ITEM - committed: 26171	aborted: 0	avg latency: 4 ms
UPDATE_ITEM_PRICE - committed: 0	aborted: 0	avg latency: 0 ms
TOTAL - committed: 26171	aborted: 0	avg latency: 5 ms

- Stored Procedure `READ_WRITE_TX_RATE=0.25`

READ_ITEM - committed: 36937	aborted: 0	avg latency: 0 ms
UPDATE_ITEM_PRICE - committed: 111540	aborted: 3	avg latency: 0 ms
TOTAL - committed: 148477	aborted: 3	avg latency: 1 ms

- Stored Procedure `READ_WRITE_TX_RATE=0.5`

READ_ITEM - committed: 83581	aborted: 0	avg latency: 0 ms
UPDATE_ITEM_PRICE - committed: 83822	aborted: 3	avg latency: 0 ms
TOTAL - committed: 167403	aborted: 3	avg latency: 1 ms

- Stored Procedure `READ_WRITE_TX_RATE=1`

READ_ITEM - committed: 231714	aborted: 0	avg latency: 0 ms
UPDATE_ITEM_PRICE - committed: 0	aborted: 0	avg latency: 0 ms
TOTAL - committed: 231714	aborted: 0	avg latency: 0 ms

1. JDBC and Stored Procedure

   可以發現，JDBC 的 throughtput 比 Stored Procedure 低很多。推測 Stored Procedure 比 JDBC Query 還要快的原因如下：

   - Pre-parsed SQL：Stored procedure 可以預先 parsed SQL statement，但是我們認為以現在的 CPU 速度這個部分應該不是影響的主要原因。
   - Pre-generated Query Execution Plan：Stored procedure 可以先預先把 query plan tree 建好，可是 JDBC 每一個 statement 都需要跑一次建立 Query plan tree。
   - Reduce Network Traffic：JDBC 需要把 SQL 拿到的 result 全部回傳給 client 端，Stored procedure 可以只將處理後的資料回傳。因此傳輸的速度較快。

2. Different ratio of `ReadItemTxn` and `UpdateItemPriceTxn`

   可以發現，`UpdateItemTxn` 的比率越高，則 throughputs 越低。因為 `UpdateItemTxn` 需要執行 `SELECT` 以及 `UPDATE` 兩個 SQL statements，而 `ReadItemTxn` 只需要執行 `SELECT` 的而已。因此當今天我們benchmark的時間都只有60秒，但執行 `ReadItemTxn` 的比例越高的話，就可以commit越多次。

3. other adjustable parameters

   `RTE_SLEEP_TIME` ： 因為我們第一次在執行的時候，有很多LockAbortException，因此我們那時候有試著把這個地方的時間，根據上面註解的建議，調成100試試看。發現這樣，

LockAbortException減少了，但是總共執行的commit數量也變少了。我們認為應該是這樣可以降低目前兩個RTE，彼此之間對於資源的競爭。所以就比較不會所以就比較不會Deadlock。

`NUM_RTES`：我們也有試著玩玩看，分別把這個RTE的數量調整成10和20。

我們是固定跑 Stored Procedure `READ_WRITE_TX_RATE=0.25` 的情況下，調整RTE的數量

1. RTE = 2

READ_ITEM - committed: 36937	aborted: 0	avg latency: 0 ms
UPDATE_ITEM_PRICE - committed: 111540	aborted: 3	avg latency: 0 ms
TOTAL - committed: 148477	aborted: 3	avg latency: 1 ms

2. RTE = 10

READ_ITEM - committed: 46462	aborted: 0	avg latency: 1 ms
UPDATE_ITEM_PRICE - committed: 140958	aborted: 37	avg latency: 3 ms
TOTAL - committed: 187420	aborted: 37	avg latency: 3 ms

3. RTE = 20

READ_ITEM - committed: 41847	aborted: 1	avg latency: 2 ms
UPDATE_ITEM_PRICE - committed: 124754	aborted: 63	avg latency: 8 ms
TOTAL - committed: 166601	aborted: 64	avg latency: 7 ms

- 可以看到當RTE = 2的時候，總共執行了14萬8千多個 transaction，並且只abort了3個，avg latency也是很小的1ms。

- 再看到RTE = 10的時候，我們總共執行的transaction數量來到了18萬7千個，但也因為有更多RTE會搶資源，所以也abort了更多的transaction，來到了37個。而且我們猜想也因為有更多競爭者的關係，也讓avg latency也跟著上升到了3ms。

- 最後我們用了RTE = 20來做實驗，可以發現在這時，總共commit的transaction數下降了，只剩16萬6千多個，abort的transaction也上升到了64個，連 `READ_ITEM` 都會被abort了。avg latency也變成7ms。

- 所以我們可以看出來，RTE的上升可以讓我們能執行的transaction數上升，但開太多也沒有意義，只是會導致很多RTE都想去做transaction，但資源又沒有那麼多，只會導致

abort越來越多transaction，但總commit的transaction數也不會變多。我們之後甚至還有開到40、80，但就是abort數量跟著上升，大概會到120~140個transaction，還有avg latency也分別上升到14ms以及32ms。至於commit的總數大概都是在15萬左右。我們認為這就跟 thrashing的概念有點類似！