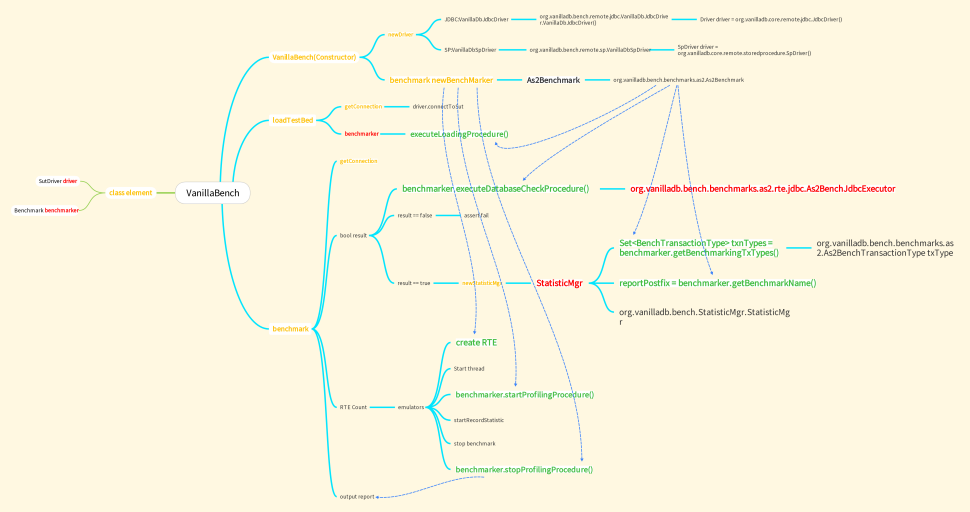
team2_assignment2_report1

members:

- 109062233 蘇裕恆
- 109062320 朱季葳
- 109062143 林聖哲



implement for UpdateItemPriceTxn

add class

since we are going to implement new transaction we nee to make it's `parameter generator` `JDBC parameter Helper` `Stored Procedure` so we add the follow class below

1. UpdateItemPriceTxnProc:

   For this class, we implement the Stored Procedure for *UpdateItemPriceTxn*.

   The class is very similar to `ReadItemTxnProc` . However , since the class is going to chack if the price is more than the `MAX_PRICE` and then update the price , we add a

varible `raise_price` to store the **increasing value** for the item when **taking item's name and price** , **check and update the price**, and call `StoredProcedureHelper` and `paramHelper` to **update the item** according the new price we gain.

2. UpdateItemPriceTxnJdbcJob

   For this class, we implement the JDBC for *UpdateItemPriceTxn*.

   We use ReadItemTxnJdbcJob as reference, to create UpdateItemPriceTxnJdbcJob since the basic structure is very similar.The operation logic is shown below:

   1. create the state for connection
   2. using for loop to run every item which we need to update
   3. gain the item id from `paramHelper.getUpdateItemId()` and call `statement.executeQuery()` to gain the name and price of item
   4. call `paramHelper.getRaisePrice()` to gain the increasing number of money.
   5. check if price is more then the MAX_PRICE and update the price
   6. call `statement.executeUpdate()` to update the sql data and store the corresponding message.

3. UpdateItemPriceProcParamHelper

   For this class, we implement the parameter Helper for *UpdateItemPriceTxn*.

   The function of this class is very simaliar to `ReadItemProcParamHelper` since both of them is going to help tx job to gain the veriable from this class

   The class has five veriable `UpdateCount` `UpdateItemId` `itemName` `itemPrice` `raise_price` . The function in this class is almost the same to the corresponding

function in `ReadItemProcParamHelper` . The tiny different is we use random to generate `raise_price` when calling `prepareParameters`

4. As2UpdateItemPriceParamGen

   For this class, we implement the parameter generator for *UpdateItemPriceTxn*.

   Since the basic logic construction is almost the same as `As2ReadItemParamGen` ,The function in this class is almost the same to the corresponding function in `As2ReadItemParamGen` .

modified class

1. As2BenchTransactionType
   - add new type `UPDATE_ITEM_PRICE` for UpdateItemPriceTxn
2. As2BenchmarkRte
   - add READ_WRITE_TX_RATE to read the probability of UpdateItemPriceTxn show up
   - add random generator to generate.**As2BenchTransactionType.READ_ITEM** for read item or **As2BenchTransactionType.UPDATE_ITEM_PRICE** for update price.
3. As2BenchJdbcExecutor
   - add new case **UPDATE_ITEM_PRICE** in `execute` to execute `UpdateItemPriceTxnJdbcJob()`
4. As2BenchStoredProcFactory
   - add new case **UPDATE_ITEM_PRICE** in `getStroredProcedure` to create `UpdateItemPriceTxnProc()`

MODIFIED STATISTICMGR

To implement the require report, we add new function `outputDetailReportCSV` to output the `.csv` report we need . The function will be called after the `outputDetailReport()` in `outputReport()`

operation logic:

1. initial the object we need(e.g. `BufferedWriter`) as same as `outputDetailReport()`
2. using for loops to convert the info store in resultSet according the formula `"time(sec), throughput(txs), avg_latency(ms), min(ms), max(ms), 25th_lat(ms), median_lat(ms), 75th_lat(ms)"` and then put into `BufferedWriter` if there is no TLE and buffer has enuogh space or it will store the abort.
3. Initialize the variable to prepare for the next info

Experiments

ENVIRONMENT

Intel® Core™ i7-1065G7 CPU 4-Core @ 1.30GHz 1.50 GHz

RAM:8.00 GB

windows x64

DB store position : 1TB HDD (st1000lm035-1Rk173)

-> has already closed Disk Write Cache

JDBC against Stored Procedures

- READ_WRITE_TX_RATE = 0.5

CONNECTION_MODE	Stored Procedures	JDBC
Txs / min	126782	1293

As we can see the Txs / min rate for SP is much faster than the JDBC mode(almost 100x).
We think the reason is that:

1. Query Optimization: Stored Procedures (SP) are typically compiled and optimized by the database engine, which can lead to better query

2. The communication for the client-host base : In the jdbc , we need to communicate beteen the host and the client , since our disk is HDD the more communication cost more time.
3. Implement of jdbc : In the implmentation of VanillaDB, which use some java api to manipulate data ,this is convenient but adds abstraction layers, which acquired internal codes to interact with the DB backend.

SP : different READ_WRITE_TX_RATE rate result in different throughput

- CONNECTION_MODE : Stored Procedures

READ_WRITE_TX_RATE	0	0.25	0.5	0.75	1
Txs / min	58725	121570	126782	147568	313228

As we can see , upon the increaing of Read rate , we have more transaction per minutes.Since the the update price not only requires querying the prices, but also updates the prices , while in ReadItem Txn only queries the prices, so the ReadItem Txn is faster than UpdateItemPrice Txn.
As for the reason above , it's not hard to see that when increasing the percentage of ReadItem, it will cause higher the tx/min value.

open Windows Disk Write Cache or not

- READ_WRITE_TX_RATE = 0.5
- CONNECTION_MODE : Stored Procedures

Cache open?	Yes	No
Txs / min	191847	126782

For this experiment , we can simply find that the Window cache actually affect the behavior alots (1.5 * without cache)

Compare with different environment : Running experiment on MacOS

Intel® Core™ i5 CPU 4-Core @ 1.4 GHz

RAM : 8 GB 2133 MHz LPDDR3

MacOS Ventura 13.2.1

time(sec)	throughput(txs)	avg_latency(ms)	min(ms)	max(ms)	25th_lat(ms)	median_lat(ms)	75th_lat(ms)
5	56974	0.08775933	0.0	20.0	0.0	0.0	0.0
UPDATE_ITEM_PRICE: ABORTED							
10	89720	0.055728935	0.0	100.0	0.0	0.0	0.0
15	20688	0.15023202	0.0	84.0	0.0	0.0	0.0

- READ_WRITE_TX_RATE = 0.5
- CONNECTION_MODE : Stored Procedures
- Txn/min : 669528

time(sec)	throughput(txs)	avg_latency(ms)	min(ms)	max(ms)	25th_lat(ms)	median_lat(ms)	75th_lat(ms)
UPDATE_ITEM_PRICE: ABORTED							
5	552	9.061594	5.0	106.0	6.0	8.0	10.0
10	696	7.1939654	5.0	16.0	6.0	7.0	8.0
15	761	6.5742445	5.0	19.0	5.0	7.0	7.0
20	724	6.910221	4.0	15.0	6.0	7.0	8.0
25	699	7.1616597	5.0	49.0	6.0	7.0	8.0
30	761	6.5755587	5.0	15.0	5.0	6.0	7.0
35	767	6.5267277	4.0	17.0	5.0	6.0	7.0
40	770	6.498701	5.0	15.0	5.0	6.0	7.0
45	744	6.725806	5.0	48.0	5.0	6.0	7.0
50	688	7.270349	5.0	18.0	6.0	7.0	8.0
55	676	7.4008875	5.0	49.0	6.0	7.0	8.0
60	569	8.796134	5.0	141.0	6.0	7.0	9.0
65	584	8.568493	5.0	31.0	6.0	8.0	10.0
70	563	8.889875	5.0	49.0	7.0	8.0	10.0
75	614	8.154723	5.0	19.0	7.0	8.0	9.0
80	541	9.255083	5.0	22.0	7.0	9.0	11.0
85	583	8.581475	5.0	25.0	7.0	8.0	9.0
90	632	7.914557	5.0	18.0	6.0	8.0	9.0
95	576	8.682292	5.0	57.0	7.0	8.0	9.0
100	645	7.7550387	5.0	20.0	6.0	8.0	9.0
105	685	7.3007298	5.0	18.0	6.0	7.0	8.0
110	684	7.3114033	5.0	18.0	6.0	7.0	8.0
115	314	7.697452	5.0	19.0	6.0	7.0	9.0

- READ_WRITE_TX_RATE = 0.5
- CONNECTION_MODE : JDBC
- Txn/min : 7736

From the above two images, we can also find out that when the connection mode is SP, it runs faster than the one in JDBC.

What's more, we can see that the performance is better on MacOS with core i5 CPU 4-Core @ 1.4 GHz

But the thing doesn't change is that the performance in SP mode is about 100 times better than the one in JDBC mode.

As a result, no matter what OS and Hardware resource we're running on, we find that the performance of SP is 100 times better than JDBC is hold in both of mac os and Windows os.