Team 17

107060008 張承勛
107062123 鄭宇軒
107062315 黃偉哲

Implementation

To accomplish this task, we traced the implementations of **ReadItem** among the two connection methods first and took them as references respectively. Precisely speaking, for any file with the key word **ReadItem**, add a new counterpart to implement **UpdateItemPrice**.

- **UpdateItemPriceTxn** with JDBC
  Add a class called `bench.benchmarks.as2.rte.UpdatePriceParamGen`, which works as a parameter generator to generate the prices to add. Next, add a class called `bench/server/param/as2/UpdatePriceProcParamHelper` to provide the help of parsing the parameters in the server side. Finally, we are now able to implement **UpdateItemPriceTxn** with JDBC by adding a new class: `bench.benchmarks.as2.rte.jdbc.UpdatePriceJdbcJob`, which follows the specifications of this task that select the name and the price of the item first, and then inspect whether the price has exceeded the maximum price. If so, adjust the price to the minimum price; otherwise, adjust the price with the number generated by parameter generator.

  Then, modify corresponding files such as `As2BenchTransactionType` or `As2BenchmarkRte` to make this works.

- **UpdateItemPriceTxn** with stored procedures
  Modify `bench/server/procedure/as2/As2BenchStoredProcFactory` first so that the benchmarker can recognize the procedure **UpdateItemPriceTxn**. Then, implement **UpdateItemPriceTxn** with stored procedures by add a new class: `bench/server/procedure/as2/UpdatePriceTxnProc`. Again, since the implementation of **ReadItem** is a good reference to take, we did not encounter any specific difficulty.

- The ratio between **ReadItemTxn** and **UpdateItemPriceTxn**
  First, specify the value of READ_WRITE_TX_RATE in `vanillabench.properities` so that we can easily control the rate in the

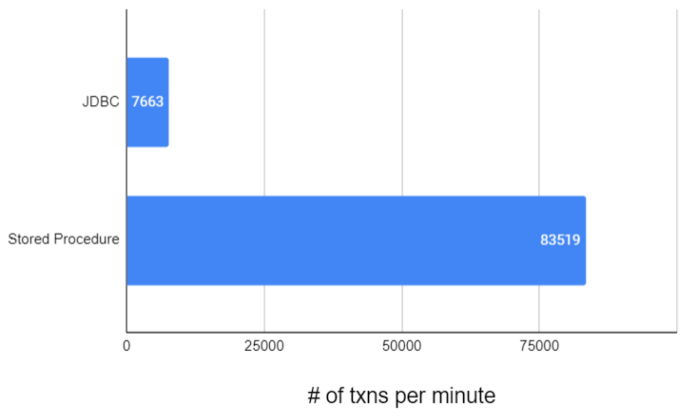
following experiments. And according to the hint, the rate is actually a probability, so we implement this function by generating a random number between [0, 1), and let the next transaction to be **UpdateItemPriceTxn** if the random number is less than or equal to the READ_WRITE_TX_RATE we set.

Experiment

- Environment

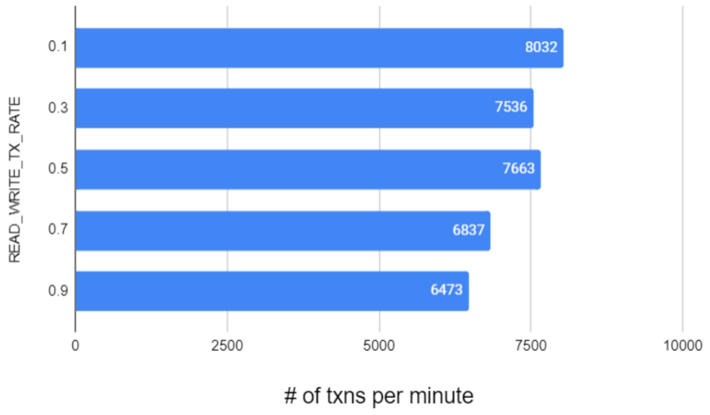
Local machine	MacBook Air 11-inch, Early 2015
CPU	Intel Core i5-5250U 1.6GHz
Memory	8 GB 1600 MHz DDR3
Disk	256 GB SSD
OS	macOS Big Sur

- JDBC vs stored procedures
  This experiment is done with exactly the same configuration except the connection mode, and the result is illustrated in the following figure. Specifically, we set READ_WRITE_TX_RATE to 0.5 in this experiment.



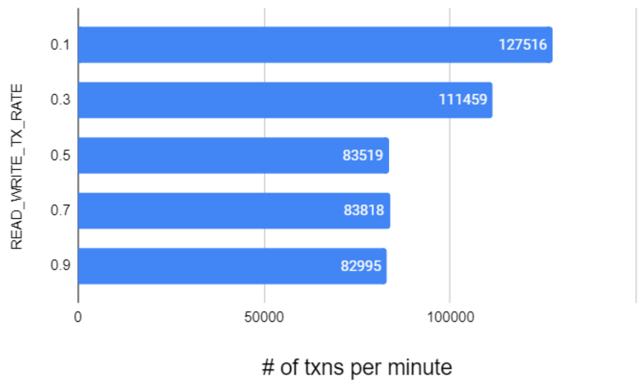
In the figure above, we can observe that the performance of stored procedures is much better than JDBC, and we believe this phenomenon is caused by the monotonic tasks. Under such a circumstance, connecting with stored procedures can execute the benchmark much more efficiently.

- Different ratio of **ReadItemTxn** and **UpdateItemPriceTxn** with JDBC



We adjusted the ratio 0.2 per time, and the result is shown in the figure above. Despite the inevitable deviation, we can still observe that the performance is in a negative correlation toward the rate of **UpdateItemPriceTxn**. We believe the relation is caused by the property of JDBC and the complexity of **UpdateItemPriceTxn**. JDBC deals with the query one at a time, and since **UpdateItemPriceTxn** is more complex than **ReadItemTxn**, it is not surprising that the performance drop if there are more **UpdateItemPriceTxn** contained in the benchmark.

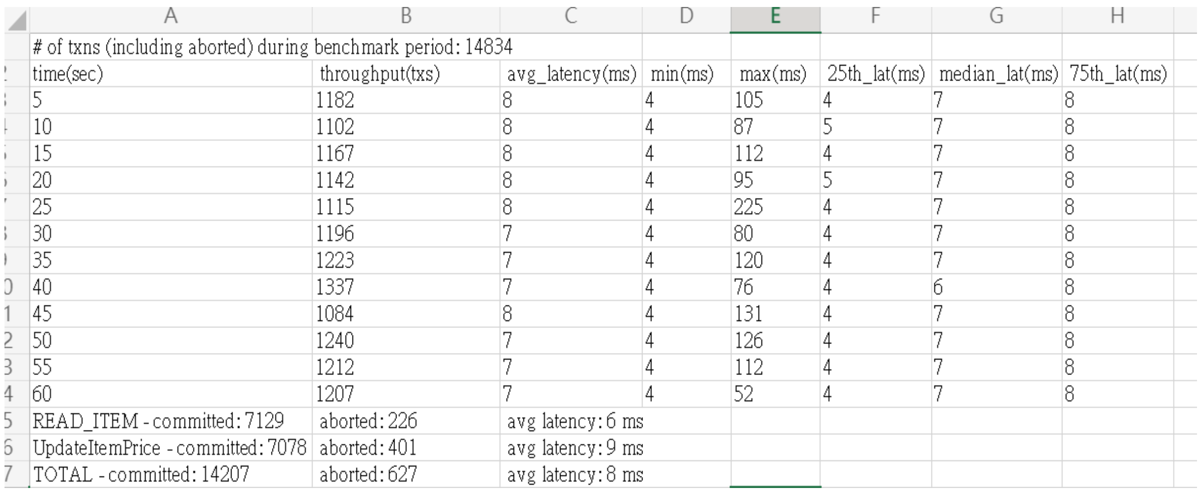
- Different ratio of **ReadItemTxn** and **UpdateItemPriceTxn** with stored procedures



Again, we adjusted the ratio 0.2 per time. Though the performance has little difference after the rate exceeds 0.5, it is still in a negative correlation toward the rate of **UpdateItemPriceTxn**. Before doing this experiment, we expect the performance would maintain the same among different READ_WRITE_TX_RATE since it saves complex queries that require executions of several SQL statements into stored procedure.

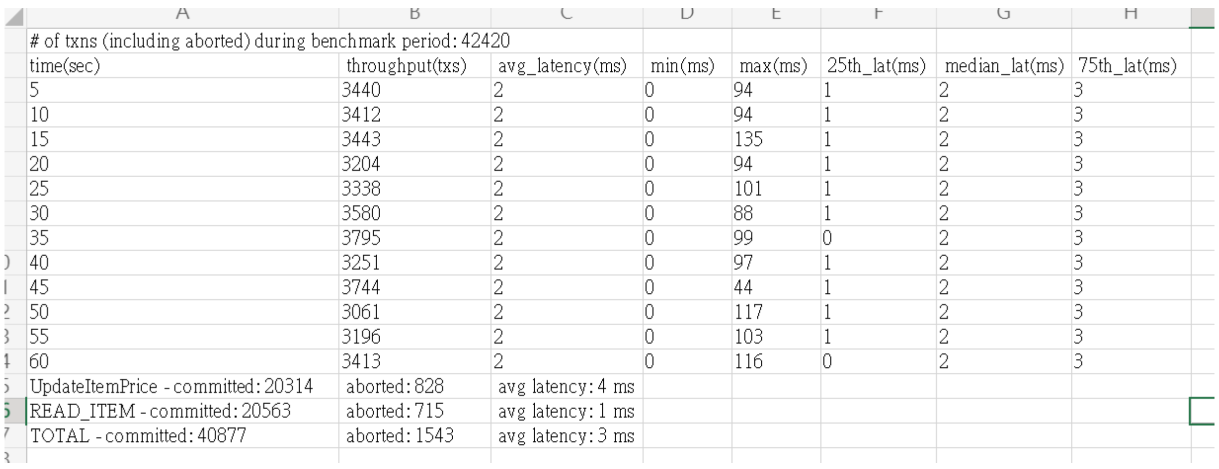
Screenshot of CSV report

- JDBC mode with READ_WRITE_TX_RATE=0.5



A	B	C	D	E	F	G	H
# of txns (including aborted) during benchmark period: 14834							
time(sec)	throughput(txs)	avg_latency(ms)	min(ms)	max(ms)	25th_lat(ms)	median_lat(ms)	75th_lat(ms)
5	1182	8	4	105	4	7	8
10	1102	8	4	87	5	7	8
15	1167	8	4	112	4	7	8
20	1142	8	4	95	5	7	8
25	1115	8	4	225	4	7	8
30	1196	7	4	80	4	7	8
35	1223	7	4	120	4	7	8
40	1337	7	4	76	4	6	8
45	1084	8	4	131	4	7	8
50	1240	7	4	126	4	7	8
55	1212	7	4	112	4	7	8
60	1207	7	4	52	4	7	8
READ_ITEM - committed: 7129	aborted: 226	avg latency: 6 ms					
UpdateItemPrice - committed: 7078	aborted: 401	avg latency: 9 ms					
TOTAL - committed: 14207	aborted: 627	avg latency: 8 ms					

- SP mode with READ_WRITE_TX_RATE=0.5



A	B	C	D	E	F	G	H
# of txns (including aborted) during benchmark period: 42420							
time(sec)	throughput(txs)	avg_latency(ms)	min(ms)	max(ms)	25th_lat(ms)	median_lat(ms)	75th_lat(ms)
5	3440	2	0	94	1	2	3
10	3412	2	0	94	1	2	3
15	3443	2	0	135	1	2	3
20	3204	2	0	94	1	2	3
25	3338	2	0	101	1	2	3
30	3580	2	0	88	1	2	3
35	3795	2	0	99	0	2	3
40	3251	2	0	97	1	2	3
45	3744	2	0	44	1	2	3
50	3061	2	0	117	1	2	3
55	3196	2	0	103	1	2	3
60	3413	2	0	116	0	2	3
UpdateItemPrice - committed: 20314	aborted: 828	avg latency: 4 ms					
READ_ITEM - committed: 20563	aborted: 715	avg latency: 1 ms					
TOTAL - committed: 40877	aborted: 1543	avg latency: 3 ms					