Introduction to Database System - Assignment 2 Report

Team 22 - 108020017 黃珮綺, 108020021 賴柏翰

Phase 1 Report

1. Implementations

As2BenchTransactionType

- Add a new `As2BenchTransactionType` for update price transaction.

Vanillanbench Properties

- Add a new `READ_WRITE_TX_RATE` properties.

As2BenchmarkRte

- **Choose the type of the next transaction.** We extract the value `READ_WRITE_RATE` defined in `BenchProperties.java` to be the ratio of read and write transaction and increase the precision by `PRECISION` (=100). The function will randomly generate a number between 0 and `PRECISION`. If the number is greater than `READ_WRITE_RATE * PRECISION`, then the next transaction type will be updated. Otherwise, it will be the default transaction type of read.
- **Get the transaction executor.** Return the executor for current transaction type.

As2StoredProcFactory

- **Handle update transaction in stored procedure.** This function calls the next stored procedure depending on the current `As2BenchTransactionType`. We add a case for updating price transaction.

As2BenchJdbcExecutor

- **Handle update transaction in JDBC.** This function calls the next JDBC job. We also add a case to do the updates.

UpdatePriceParamGen

- **Generate a parameter list.** The first item indicates the number of items to be updated (`TOTAL_WRITE_COUNT`). Then, generates a number of `TOTAL_WRITE_COUNT` pairs of the index of the item we want to update later each followed by a randomly generated value of a raise.

UpdatePriceProcParamHelper

- **Get the indexes of items that will be updated and their price raise.** It will help extract the list of values generated by the `UpdatePriceParamGen`.
- **Get updated price of the item with the given index.** The function returns the exact price for the item of the given index after update. The price will be adjusted to `As2BenchConstants.MIN_PRICE` if the original price exceeds `As2BenchConstants.MAX_PRICE`. Otherwise, the price will be the addition of the original one and the raise.

UpdatePriceJdbcJob

- **Execute** `SELECT`. First, select the name and the price of item with the index. The result will be recorded in the result set and will be used to do the update.
- **Execute** `UPDATE`. Get the updated price of the current item and execute the update query. The return value of the executor will be the number of lines updated. Check if this number is not zero to make sure the execution is done.

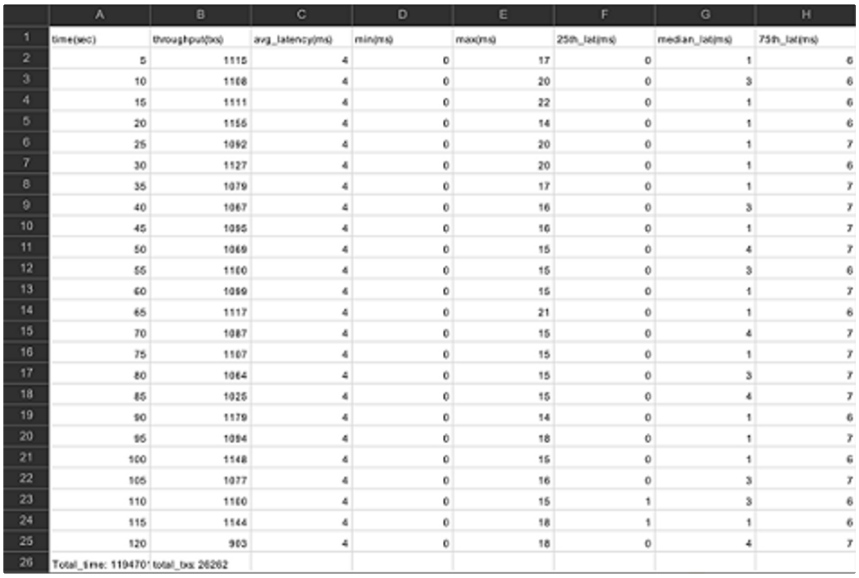
UpdatePriceProc

- **Execute** `SELECT`. Same as in the JDBC job, select the name and the price of item with the index. The result will be recorded in the result set and will be used to do the update.
- **Execute** `UPDATE`. Get the updated price of the current item and execute the update query. The return value of the executor will be the number of lines updated. Check if this number is not zero to make sure the execution is done.

2. CSV Report

StatisticMgr

- **Output report.** We record the average, minimum, maximum, 25th, median, 75th latency along with throughput in every 5 seconds. Also, we output the total time (in ns) spent on execution and the total number of transactions.
- **Example CSV file.** Here is an example output with 0.5 read-write rate on stored procedure.



	A	B	C	D	E	F	G	H
1	time(sec)	throughput(txs)	avg_latency(ms)	min(ms)	max(ms)	25th_lat(ms)	median_lat(ms)	75th_lat(ms)
2	5	1115	4	0	17	0	1	6
3	10	1108	4	0	20	0	3	6
4	15	1111	4	0	22	0	1	6
5	20	1156	4	0	14	0	1	6
6	25	1092	4	0	20	0	1	7
7	30	1127	4	0	20	0	1	6
8	35	1079	4	0	17	0	1	7
9	40	1067	4	0	16	0	3	7
10	45	1095	4	0	16	0	1	7
11	50	1069	4	0	15	0	4	7
12	55	1100	4	0	15	0	3	6
13	60	1089	4	0	15	0	1	7
14	65	1117	4	0	21	0	1	6
15	70	1087	4	0	15	0	4	7
16	75	1107	4	0	15	0	1	7
17	80	1064	4	0	15	0	3	7
18	85	1025	4	0	15	0	4	7
19	90	1179	4	0	14	0	1	6
20	95	1094	4	0	18	0	1	7
21	100	1148	4	0	15	0	1	6
22	105	1077	4	0	16	0	3	7
23	110	1100	4	0	15	1	3	6
24	115	1144	4	0	18	1	1	6
25	120	903	4	0	18	0	4	7
26	Total_time: 119470	total_txs: 26262						

3. Experiment

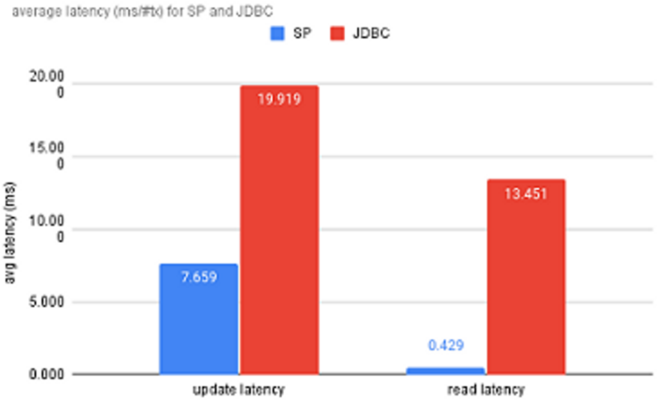
- **Environment**

  `Intel Core i7-8565U @1.8GHz, 16 GB RAM, 512 GB SSD, Windows 10`

- Performance comparison

1. Latencies of `ReadItemTxn` and `UpdatePriceTxn`

	# of tx (ratio)	total latency (ms) (ratio)	average latency (ms/#tx)
SP update latency	13010 (49.5%)	99638 (94.6%)	7.658570331
SP read latency	13252 (50.5%)	5680 (5.4%)	0.4286145487
JDBC update latency	3406 (48.5%)	67844 (58.2%)	19.91896653
JDBC read latency	3614 (51.5%)	48612 (41.7%)	13.4510238



> Here is a simple comparison of the latencies of `UpdatePriceTxn` and `ReadItemTxn`. We set `READ_WRITE_TX_RATE` to be 0.5, and run on both procedures (SP and JDBC).
>
> We can observe a great difference between the latencies of `UpdatePriceTxn` and `ReadItemTxn`, respectively, for each procedure. The average latencies of `UpdatePriceTxn` are higher than that of `ReadItemTxn`, which indicates that the system operate `ReadItemTxn` faster than `UpdatePriceTxn`.
>
> In addition, at our first glance, the performance of stored procedure is better than JDBC job.

2. Different ratio of `UpdatePriceTxn` with respect to `ReadItemTxn`

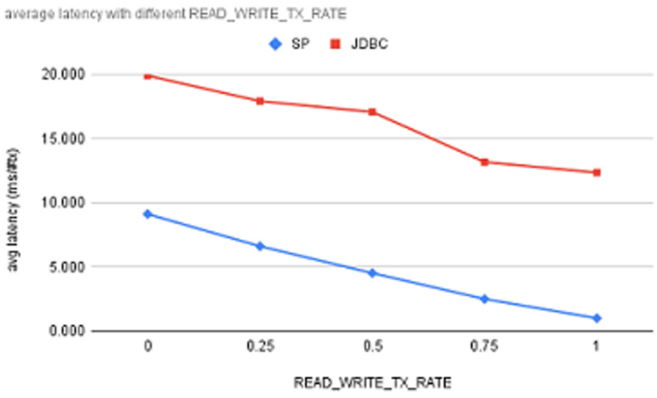
   - SP

ratio (#read/#update)	# of tx	total latency (ms)	average latency (ms/#tx)	performance (#tx/ms)
0.00 (100/0)	114075	118,123	1.035	0.966
0.25 (75/25)	47061	119,122	2.531	0.395
0.50 (50/50)	26262	119,470	4.549	0.220
0.75 (25/75)	18019	119,607	6.638	0.151
1.00 (0/100)	13099	119,709	9.139	0.109

- JDBC

ratio (#read/#update)	# of tx	total latency (ms)	average latency (ms/#tx)	performance (#tx/ms)
0.00 (100/0)	9703	119,977	12.365	0.081
0.25 (75/25)	9097	119,953	13.186	0.076
0.50 (50/50)	7020	119,950	17.087	0.059
0.75 (25/75)	6692	119,969	17.927	0.056
1.00 (0/100)	6024	119,933	19.909	0.050

- Compare JDBC and SP



> Overall, the average latency decreases when `READ_WRITE_TX_RATE` approaches to 1. That is, the higher proportion of `UpdatePriceTxn` with respect to `ReadItemTxn`, the better performance on execution. This agrees with the previous discussion on the latencies of `ReadItemTxn` and `UpdatePriceTxn`, respectively, that, on average, the system can execute `ReadItemTxn` faster than `UpdatePriceTxn`.
>
> Furthermore, all the numbers of transaction executed on stored procedure are larger than those on JDBC even though the `READ_WRITE_TX_RATE` changes. The average latencies on stored procedure are under 10 ms/#tx, while the average latencies on JDBC are all above. We also calculate the number of executions per mini-second (#tx/ms) for each procedure, and come to a conclusion that the performance of stored procedure is better than that of JDBC job.

Reference:

Source Code