Exchange Server Scalability Report

Tiffany Lin/tl330
Che-Jui Nien/cn154

Introduction

In this homework, we implemented an exchange matching server using C++ that supports multi-threading, concurrency control, and core control. The server receives client requests in XML format and performs various stock exchange operations:

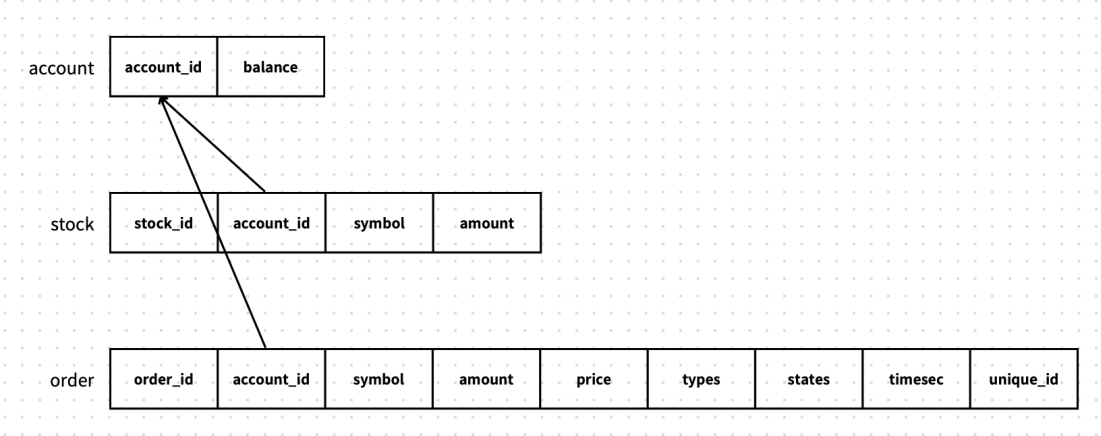
1. create user account
2. creating stock holdings for users
3. matching and executing transaction orders (buy or sell)
4. querying transactions
5. canceling transactions

Additionally, we built a test infrastructure for functionality testing and scalability testing on the client side.

1. Server Implementation

1.1 Create a PostgreSQL database

We first cleaned up existing tables in the database, then created a PostgreSQL with the following 3 tables, with the ACCOUNT table storing the information of each user, the STOCK table storing the stocks that each user holds, and the ORDER table storing all transactions(opened, executed, canceled).



We wrote two function to conduct this action:

```
void createTable(string fileName, connection *C);
void deleteTable(connection *C, string tableName);
```

1.2 SQL functions

For this part, we wrote 5 functions, each corresponding to an action in the upper 5 operations that we listed

1. create user account
   `string add_account(connection *C, int account_id, float balance);`
2. creating stock holdings for users
   `string add_stock(connection *C, int account_id, string symbol, int amount);`
3. matching and executing transaction orders (buy or sell)
   `string add_order(connection *C, int account_id, string symbol, int amount, float price);`
4. querying transactions
   `string query(connection *C, int order_id);`
5. canceling transactions
   `string cancel(connection *C, int account_id, int order_id)`

1.3 Server thread

In our implementation, the server's thread continuously runs and accepts new TCP connections from clients while receiving their requests. We adopted the per-request threading strategy. By utilizing this approach, we ensured that each client request is processed by a separate thread, which significantly enhances the performance and responsiveness of the server.

1.4 Concurrency control

To prevent "read-modify-write" that may cause data race conditions, we took precautions to maintain concurrency control to avoid conflicts between multiple threads reading and writing to the same data. We used the "Row level locking" between SELECT and UPDATE so that the row is locked after it is selected and is unlocked when it is committed. This helps ensure that the server operates smoothly, even under heavy workloads.

1.5 Request queue & thread limit

```
postgres=# SHOW max_connections;
 max_connections
-----------------
 100
(1 row)
```

The PostgreSQL database has a maximum connection of 100. So we have to manage the total number of threads created. We set our connection limit to 90. We place a lock on codes modifying the number of connections, to prevent race conditions. Each thread for connection will

wait until the total number of threads are less than 90 to make new connections. We avoid the creation of too many connections, which can lead to decreased server performance and potential resource contention. Overall, this design allows for the efficient processing of client requests while maintaining control over thread usage.

2 Client Testing

We conducted both functionality testing and scalability analysis.

2.1 Functionality Testing

In Functionality testing, we make sure that each request is processed properly and the information in the database is correct. To test the functions, we can either run `./test filename` or run the bash script `./run.sh` that iterates through all the files. We then check the output of the code and the database to see if our implementation is correct.

order_id	account_id	symbol	amount	price	types	states	timesec	unique_id
4	1	h	3	102	sell	open	1680655282	4
8	3	e	4	100	buy	execute	1680655282	9
1	1	e	4	100	sell	execute	1680655282	1
2	1	f	1	100	sell	open	1680655282	2
2	1	f	3	100	sell	execute	1680655282	11
9	3	f	3	100	buy	execute	1680655282	10
11	3	h	4	101	buy	open	1680655282	13
13	3	j	1	101	buy	open	1680655282	15
13	3	j	3	100	buy	execute	1680655282	16
5	1	j	3	100	sell	execute	1680655282	5
15	3	l	4	101	buy	open	1680655282	18
16	4	g	2	99	buy	execute	1680655282	20
6	2	g	2	99	sell	execute	1680655282	6
16	4	g	2	99	buy	execute	1680655282	21
10	3	g	2	99	sell	execute	1680655282	12
16	4	g	2	101	buy	open	1680655282	19
16	4	g	1	100	buy	execute	1680655282	22
3	1	g	1	100	sell	execute	1680655282	3
17	4	k	2	99	sell	execute	1680655282	24
7	2	k	2	99	buy	execute	1680655282	7
17	4	k	3	98	sell	open	1680655282	23
17	4	k	2	99	sell	execute	1680655282	25
14	3	k	2	99	buy	execute	1680655282	17
18	1	i	4	101	sell	execute	1680655282	27
12	3	i	4	101	buy	execute	1680655282	14
19	1	l	3	102	sell	open	1680655282	28

(26 rows)

2.2 Scalability Analysis

In Scalability testing, we make sure that our program can run on a large scale of data. To test the functions, we can either run `./test filename` or run the bash script `./run.sh` that iterates through all the files. We then check the time that is required to handle multiple client requests.

We can change the 2 variables in the bash script to generate multiple client connections, fileNum can be 1~10, maxThread can be larger than 1. The total number of requests are fileNum*maxThread.

2.2.1 Different Request

For testing different requests, we run the bash script `./run1.sh` to test the latency of our 10 test cases. We modify this line to run our program on a different number of cores. We test our program with maxThread=10. Though the time here may differ using different testing files. For our program, our testing files contain multiple orders so it may take longer to process.

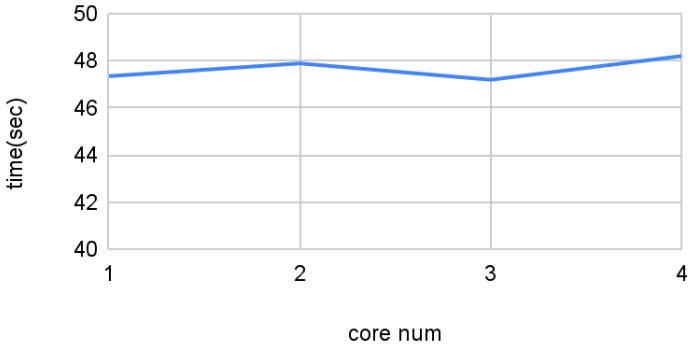
test case		Time(sec)	Latency (sec)
1	create	0.209008787	0.0209008787
2	create	0.242387147	0.0242387147
3	transaction	0.135019143	0.0135019143
4	transaction	0.147469295	0.0147469295
5	transaction	0.150876955	0.0150876955
6	transaction	0.353486113	0.0353486113
7	transaction	0.208325594	0.0208325594
8	transaction	0.246987493	0.0246987493
9	transaction	0.203235502	0.0203235502
10	transaction	0.23890316	0.023890316

2.2.2 Different number of cores

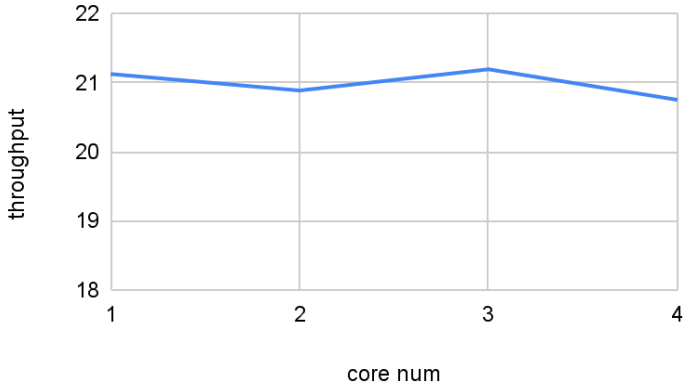
For testing different cores, we modified the bash script for the server and used `taskset -c 0,1,2,3 ./test` to make it run on 4 cores. We modify this line to run our program on a different number of cores. We test our program with fileNum=10 and maxThread=1,000, meaning 10,000 requests.

core num	time(sec)	throughput
1	47.3393694	21.12406677
2	47.88085199	20.88517557
3	47.1873917	21.19210162
4	48.19248383	20.75012368

- Time



- Throughput



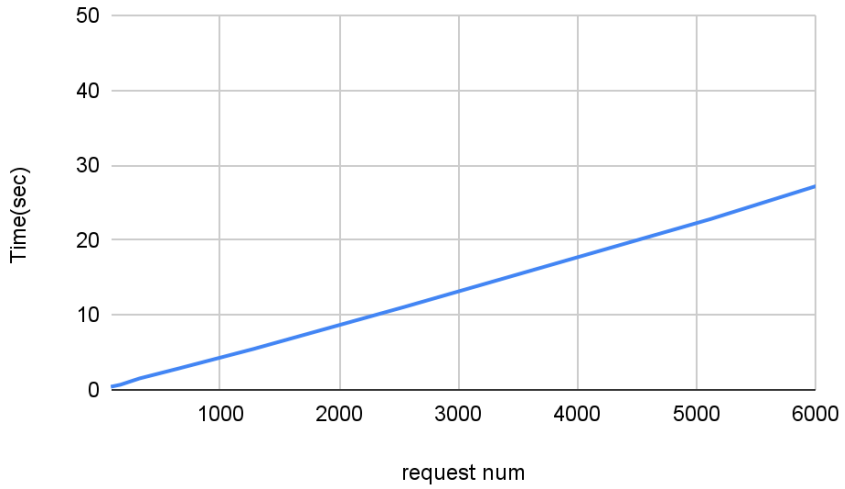
In our case, the core number does not make a big difference on the performance. We believe that the reason for the lack of significant performance improvement could be due to uneven workload distribution between the cores. Furthermore, since the workload may not be intense enough to exceed the processing capacity of a single core, the workload is concentrated on one core.

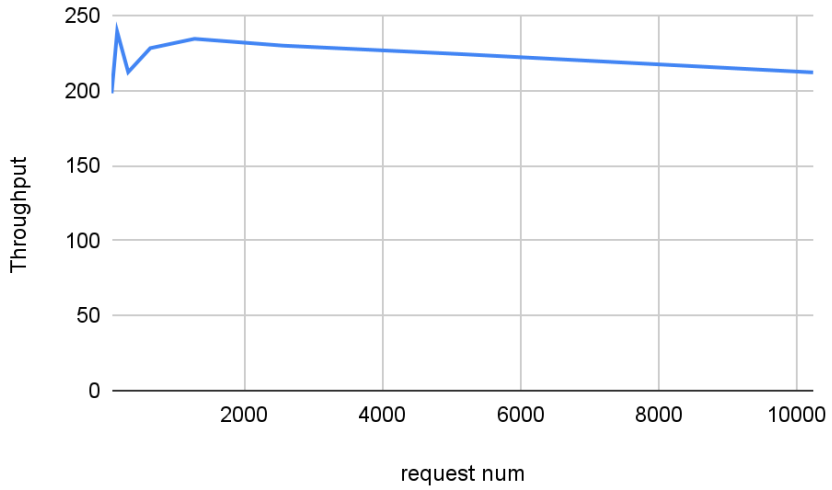
2.2.3 Different number of requests

For this part, we are testing different the latency and throughput on number of requests so we modify `maxThread` in `run.sh` to 2, 4, 8, 16, 32, 64, 128, 256 to generate 20, 40, 80, 160, 320, 640, 1280, 2560 requests.

request num	Time(sec)	Throughput
80	0.403479116	198.2754418
160	0.668416199	239.3718169
320	1.506660891	212.3901947
640	2.80244337	228.3721437
1280	5.456416622	234.5861925
2560	11.12799528	230.0504211
5120	22.81299476	224.4334886

10240	48.25917083	212.187649





In our case, the number of requests have a linear relationship with the time that is needed, and the throughput remains between 198 and 239. We believe that the throughput will decrease accordingly if the number of requests increases.

3 Reference

1. row level locking
2. core control