Stock Exchange Matching Engine Report

Kaixin Lu(kl461) & Xinyi Xie(xx98)

April 2024

1 Introduction

For this assignment, we finish an exchange matching engine which can simulate stock markets to buy or sell orders. We use C++03 and PostgreSQL to support this system and this server can handle the request in XML format and respond to the clients in XML format. In addition, the server uses multi-threading to improve the efficiency and capitalizes on the character of database to handle concurrency.

2 Implementation

2.1 Database

The system's architecture is built around four primary tables: **ACCOUNT**, **ORDERS**, **POSITION**, and **STOCK**, which are designed to record user accounts, transactions, stock positions, and stock symbols respectively.

ACCOUNT Table **Purpose:** Stores user account information.
Primary Key: ACCOUNT_ID
Fields:

- ACCOUNT_ID: Unique identifier for the account.
- BALANCE: Balance of the account.

ORDERS Table **Purpose:** Records buy and sell orders submitted by users.
Primary Key: ORDER_ID
Foreign Keys: ACCOUNT_ID, STOCK_ID
Fields:

- ORDER_ID: Unique identifier for the order.
- TRANS_ID: Unique identifier for the transaction.
- ACCOUNT_ID: ID of the account submitting the order.
- STOCK_ID: ID of the stock being traded.
- AMOUNT: Number of stocks to buy (positive) or sell (negative).
- PRICE: Price of the order.
- STATUSS: Status of the order (e.g., OPEN, CANCELED, EXECUTED).
- ORDER_TIME: Time the order was submitted.

POSITION Table **Purpose:** Tracks the quantity of each stock owned by an account.
Primary Key: ID
Foreign Keys: ACCOUNT_ID, STOCK_ID
Fields:

- ID: Unique identifier for the position.
- STOCK_ID: ID of the stock.
- ACCOUNT_ID: ID of the account owning the stock.
- AMOUNT: Quantity of the stock owned.

STOCK Table **Purpose:** Lists the stocks available in the market.
Primary Key: STOCK_ID
Fields:

- STOCK_ID: Unique identifier for the stock.
- SYMBOL: Symbol of the stock.

Operations include creating tables, adding accounts and positions, executing and canceling orders. These operations are crucial for maintaining the integrity and accuracy of the exchange system, leveraging PostgreSQL features to ensure atomic and isolated transactions.

Creating Tables and Adding Accounts/Positions SQL statements are used for table creation, while functions handle account additions and position updates, ensuring data consistency across the database.

Executing Orders This operation includes verifying account balances, updating positions, and processing buy or sell orders. It is designed to reflect accurate market transactions, adjusting account balances and stock positions as needed.

Canceling Orders Handles the cancellation of orders, updating account balances or stock positions depending on the order type.

Transaction Management Utilizes PostgreSQL's transaction processing capabilities to ensure that all database operations are atomic and isolated, preventing data inconsistencies.

2.2 Server

The server component is central to the exchange matching engine project. It begins with no predefined symbols, accounts, or orders. It listens on port 12345 for incoming connections, processing requests formatted as XML messages. These messages dictate actions regarding account creation, symbol creation, and transaction handling (buy/sell orders, queries, cancellations).

- **Account and Symbol Management:** It creates and manages accounts with unique IDs and balances, introduces new trading symbols, and allocates shares to accounts.
- **Order Processing:** The server matches buy and sell orders based on criteria like symbol, quantity, and price limits. It supports executing and canceling orders with strict validation to ensure transaction integrity.

- **Transaction Handling:** Actions such as trade execution, order status queries, and order cancellations are managed, with detailed success or error responses for each transaction.
- **Scalability and Performance:** The server's design emphasizes scalability, particularly in how it handles concurrent transactions and manages resources under varying loads.

Key to its operation is ensuring atomicity in transactions, maintaining data integrity, and providing clear communication through XML messages.

3 Scalability Experiments

3.1 Methodology

The goal of this scalability testing was to evaluate the performance of our program across various CPU core counts, especially focusing on its throughput and execution time in scenarios with different numbers of clients.

We came up with two testing strategies, single client with multiple cores, and multiple clients with multiple cores. For the load generation, each client issued 3,000 requests, encompassing both the creation of new orders and the execution of transactions. Tests were conducted 10 times, and average values were taken to mitigate the effects of outliers and enhance the reliability of the results. All of the variances are under ±5%.

3.2 Analysis on Scalability

Single Client with Multiple Cores We wrote bash scripts to simulate a single client sending requests to the server, engaging in operations like creating and executing orders by sending XML requests in loops, thus imposing a workload on the server. We used **taskset -c** command to make the client running on multiple cores. For the different number of cores on the client side, we also tested different number of cores on the server side.

Table 1: Throughput (request/second) of Multi-core Clients with Multi-core Servers

Core Num of Server \ Core Num of Client	1	2	4	8
1	146.68	149.49	151.48	151.87
2	141.51	147.51	146.54	148.28
4	143.09	146.34	145.52	149.02
8	148.55	147.83	150.59	153.14

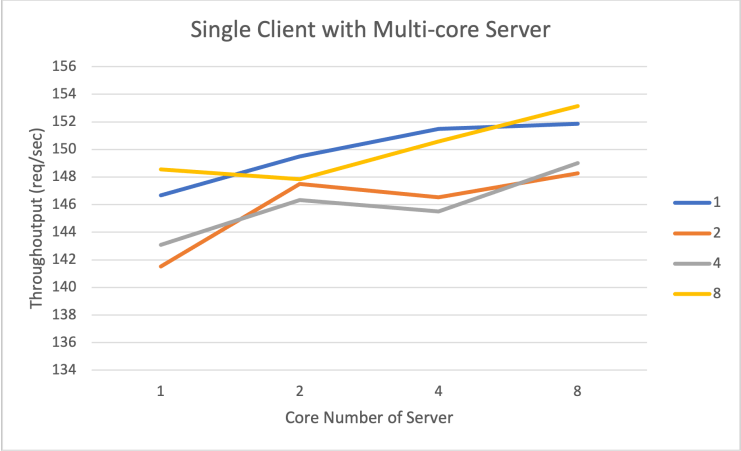

Figure 1: Throughput of Single clients with Multi-core Servers

First, we tested single clients with multiple cores sending requests to the server with multiple cores. From Table 1 and Figure 1, we can see that the number of cores on the server side doesn't have significant impact on the throughput.

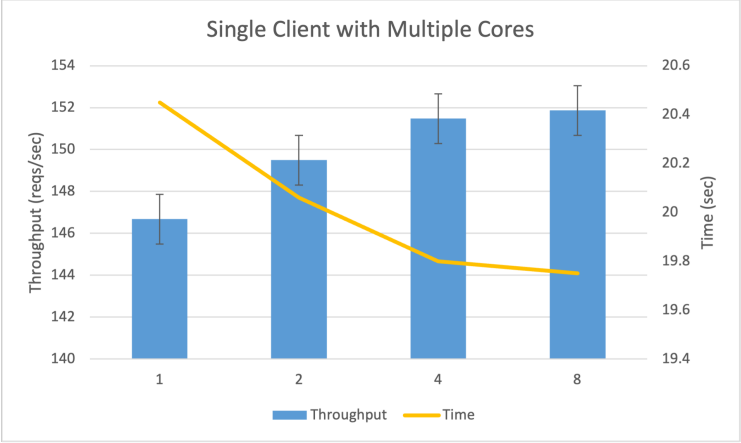

Figure 2: Throughput and Time of Single client with Multiple Cores

When the server only runs with one core, we can learn from Figure 2 that as the number of cores goes up, the throughput improves, and the execution time reduces. Generally, as the number of CPU cores increased, the server's capacity to process requests improved, reflected by increased throughput and decreased execution time. Through this 'htop' table3, we can see that the usage of each cores are most even. This is under the situation that server and client are all running on 8 cores.

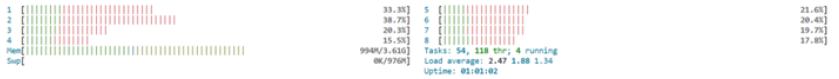

Figure 3: Usage of Each Core-htop

If you would like to re-run the test, please change your directory to "/testing/sacleTest/", and run "chmod +x *.sh" and "./test1.sh". The results can be seen at the terminal.

Multiple Clients with Multiple Cores We also conducted tests on the throughput and execution time for multiple clients running on multiple cores. Consistent with our previous methodology, each client sent 3,000 requests. Additionally, we experimented with manually enforced allocations (8 clients on 8 cores, with each core exclusively allocated to a single client). For the final results, we calculated the average value per client to avoid random outcomes.

Table 2: Throughput (request/second) and Time of Multiple Clients with Multiple Cores

Number of Cores	Throughput (req/sec)	Time (second)
1	39.37875	76.53
2	34.30875	86.92
4	31.30375	96.18
8	28.68625	104.56
8 (one client per core)	35.23375	85.13125

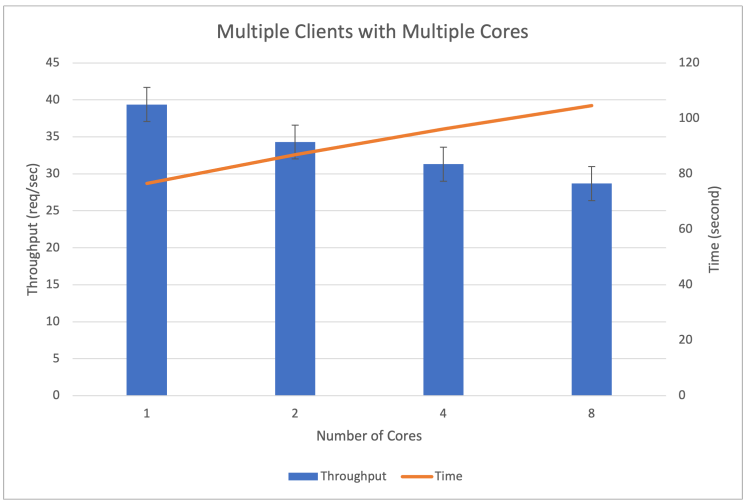

Figure 4: Throughput and Time of Multiple Clients with Multiple Cores

As shown in Table 2 and Figure 4, it can be observed that as the number of cores increases, the throughput per client decreases, and the time increases. We believe this is due to the CPU spending a certain amount of time on resource allocation, hence the more cores there are, the lower the throughput and the longer the time becomes. This might be due to the excessive usage of capacity. If the number of requests is larger than the capacity of the server, the trend would be that the higher the throughput and the shorter the response time. When we limit the configuration to 8 cores with only one client per core, the throughput and time closely resemble the results of 8 clients on 2 cores, which corroborates our hypothesis.

If you would like to re-run the test, please change your directory to "/testing/sacleTest/", and run "chmod +x *.sh", "./test2.sh" and "./test3.sh". In these tests, we distribute each client with one individual terminal screen to simulate different clients in different places. The results can be seen at "/testing/sacleTest/output2" and "/testing/sacleTest/output3".

3.3 Conclusion

- By comparing the throughput and execution time across different test scenarios, we observed the impact of CPU core counts on server performance.

- Generally, when a single client running on multiple cores, as the number of CPU cores increased, the server's capacity to process requests improved, reflected by increased throughput and decreased execution time.
- In the multiple clients, multiple cores scenario, the server may not have significant improvement in scalability as the number of cores increases.