HW4 Report

xs106

sm957

Introduction:

In this project, an exchange matching system is developed into a multithread server and client system with a concurrency-controlled database in *PostgreSQL*. The server initially establishes a connection with the database, following which it sets up a thread pool with certain predetermined number of threads to handle the requests send by clients in *XML* format. Each thread will process the *XML* file and then interacts with the database to create an account, share symbol and process created order by invoking the corresponding functions in the database. At the same time, database operates on these requests and throws an exception if any error exist, enabling each thread on the server to construct a corresponding response and send it back to the client. This report will describe the implementation method of server, client and database and then analyze the result of scalability experience by measure the throughput and latency of sending different large number of requests from client to server and process the request will difference number of system cores.

Implementation:

a. Server: In this project, the scalability is realized by pre-created multithreads in a thread pool with redetermined number of threads, which benefit in better load balancing and avoid the overhead of create each thread, which easier to control the number of threads. It is implemented by boost threadpool library. The main thread will be used to receive the request from the client and then put it into the thread pool to assign it to the created threads.

   In the beginning, server will be set up and connect to the database. Then, it will create a *threadpool* class object, with assigned number of threads and each of them will enter an infinite loop parallelly to wait for the task to be enqueued and execute it. Each task executed by threads is to connect to clients and receive the request message send by them. Then, it will get rid of the first line of the received message and put the task to handle received request message in the thread pool task queue.

   The workers threads will continuously check if there is new task in the queue. If it find a new task, run the *handleRequest()* function, which will parse the received the xml file into corresponding requests using the *tinyxml2 parsing library*. The requests will be in a form of children class object of *Request* class, which has several children class: *Account*, *Position*, *OpenOrder*, *Order*, *Query* and *Cancel* class. Each children class will override the *execute()* virtual function in the parent *Request* class. So that every time the thread parsed the request into a specific class, it will call the override *execute()* function to interact with database to do corresponding operation such as create account, create positions, create open order, cancel order, make a query or cancel an open class. After that, it will send back corresponding result response to client. The task queue will

be protected by a mutex queue lock to prevent thread accessing the queue when new task is added into the queue, which provide a thread-safe working mechanism to execute tasks in different threads parallelly.

b. Database

In the database, 4 tables will be implemented to store the share system information: *accounts*, which store the account id and balance of each account; *positions*, which store the account id which this share belongs to and the symbol and amount of the position; *openorder*, which will used to store the order request from the client, which including all "open", "executed", "cancel" status. It will generate a transaction id for each order and record the account id which this order belongs to. Then it stores the symbol and amount of share of this transaction order as well as the limit that buyer/seller can accept. When order in *openorder* table get matched, it will put the matched order into *orders* table, in which the order information can't be changed and only contains two statuses: "executed" or "cancel" to indicate this transaction is finished. Beside the information contained in *openorders*, the *orders* table will also store the truncation executed time and the executed price.

The implemented database in this project adopts the *RAII* technique, where all the database functions and fields are encapsulated into a class called *stock_database*. The class's destructor method includes the *disconnect()* function, allowing for the automatic disconnection of the database upon the object's destruction. So that the server can just create and initialize a *stock_database* object in the beginning, it will keep connecting during the running of the program and then automatically disconnect with database when it goes out of scope.

In this project, the database concurrency is set to repeatable read isolation level, which not only avoid the complexity and high system resource consumption of serializable isolation, but also can avoid dirty read, and nonrepeatable read, which the selected data has been edit by another transaction in another thread. If the data is changed by another thread, it will throw an error exception and rollback. In this case, the throw exception will be catch and send an error response to client to indicate the failure of transaction. In addition, to avoid the *read-modify-write cycles*, *row-level-locks* are implemented in database function. To avoid dead lock during the implementation of *row-level-lock*, the *"SELECT ..FOR UPDATE"* followed by "*UPDATE"* function are both wrapped into the same function. So that we only have lock when we need update value. In addition, in *InsertPositions()* function, *"ON CONFLICT (ACCOUNT_ID,SYM) DO UPDATE"* is used to turn the inserting operation into update operation if there is another thread has insert the position before.

There are 8 public function in the *stock_database* class to interfacing server, which include *initDB()*, *CreateAccounts()*, *CreatePositions()*, *CreateOpenOrder()*, *CancelOpenOrder()*, *getOrderQuery()* and its constructor and destructor.

In the *CreateAccounts()* function, it will verify the existence of account id and the non-negative value of balance before inserting the new account into database.

In *CreatePositions()* function, it will verify the existence of the account if and non-negative amount of share. Then decide if the position is not existed in the account

before, it will insert a position, otherwise, it will update the existed position.
In *CreateOpenOrder()* function, it will insert a new open order in the *openoder* table, and match it to corresponding open sell/buy orders that have the same share symbol and fit the limit requirement(buyer limit>sell limit). After finish matching these orders, it will process this order, by distinguish the new open order is from buyers or sellers. If the first matched order has the same amount of share, we can just update status of both order from 'open' into "executed" and insert two executed orders into *orders* table, update seller balance and buyer balance (for refund) and add share into buyer's account. If the matched order amount is larger than the new open order amount, then we may split the matched order into both open & executed orders and do a similar operation with previews operation. In the last situation, if the first matched order amount can't fit the required amount, then a loop was used to make multiple transaction for this new open order, until it satisfied the needed amount, or all matched order are executed.
In *CancelOpenOrder()*, it will firstly verify the existence of the corresponding open order and then refund for buyers(or return shares into seller account) and update the "open" status into "canceled".

Scalability test:

In this project, the scalability of this system is measured by carry out 3 experiments.

Experiment 1: Different cores with single thread: set the core number to 1,2,4 and run with different number of requests in single core server. In this experiment, the client will firstly send create account request to create a request into the database, and after receiving the response from the server, it will send another request to make transaction to create order, make query and then cancel the order. The measured time will be the total time from client send the first request to the last response it received.
Results:
Figures below shows the measurement of single thread to process various number of threads using different number of cores.

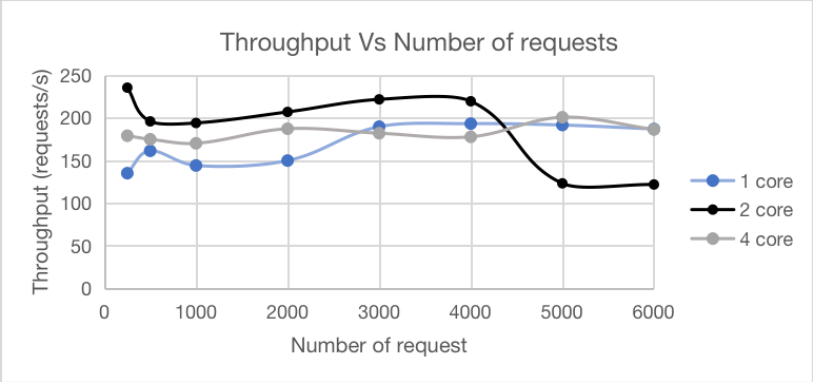

Figure 1 Calculated throughput with processing various request with different number of cores.

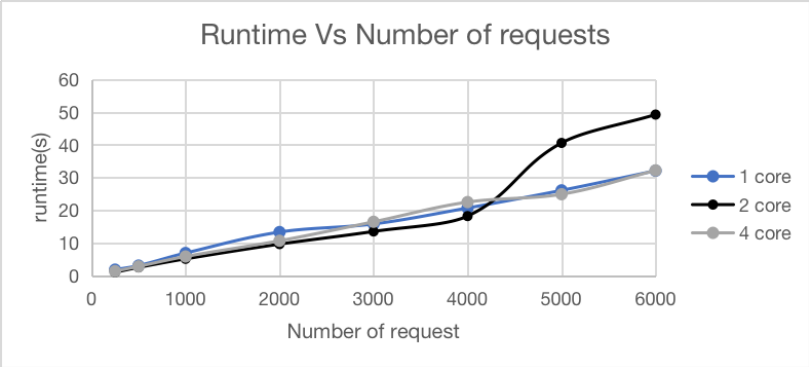

Figure 2 Measured runtime with processing various request with different number of cores.

Table 1 Average throughput of single thread server with different number of core.

Core number	1	2	4
Average throughput	175.98	181.28	184.13

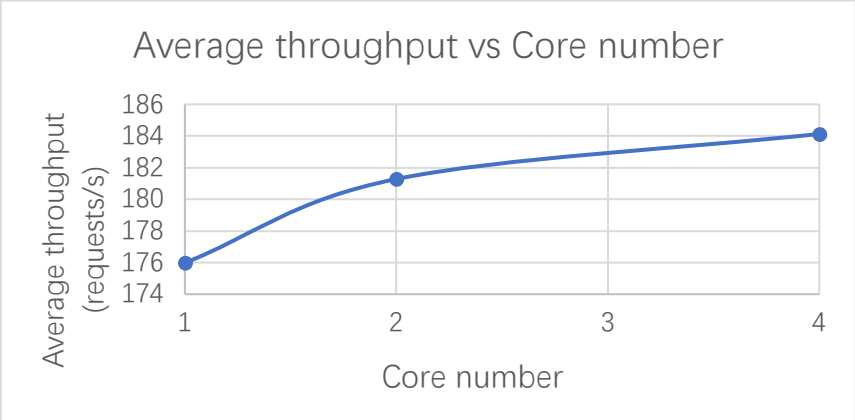

Figure 3 Measured average throughput with processing various request with different core number

The Figure 1 shows that with the same number of cores, the relationship between the number of request and the throughput in single thread server will all have a similar pattern, which keep in a constant value (around 150-200 request/s) with little fluctuation. These results are under our expected since the single thread can only process the request sequentially, therefore, the throughput will not change and keep around constant.

The Figure 2 shows that with the same number of cores, the running time of the single thread program will have linear relationship with the increasing number of requests. The reason of this result is also the single thread can only process the request sequentially, so with the increasing number of requests, the processing time will increase.

The Figure 3 shows that although the average throughput has increasing with the number of cores 1,2,4, but the increasing speed of throughput is smaller than the cores number, which the double number of cores didn't results in double throughput. This is because with the increasing number of cores, there will have some overhead that the system needs to consume more time in manage system resourced and assign each task to different core. Therefore, although the number of cores has double increased, the throughput can not increate doubly.

Experiment 2: Different cores with single thread: set the core number to 1,2,4 and run with different number of requests in single core server. In this experiment, the client will firstly send create account request to create a request into the database, and after receiving the response from the server, it will send another request to make transaction to create order, make query and then cancel the order. The measured time will be the total time from client send the first request to the last response it received.

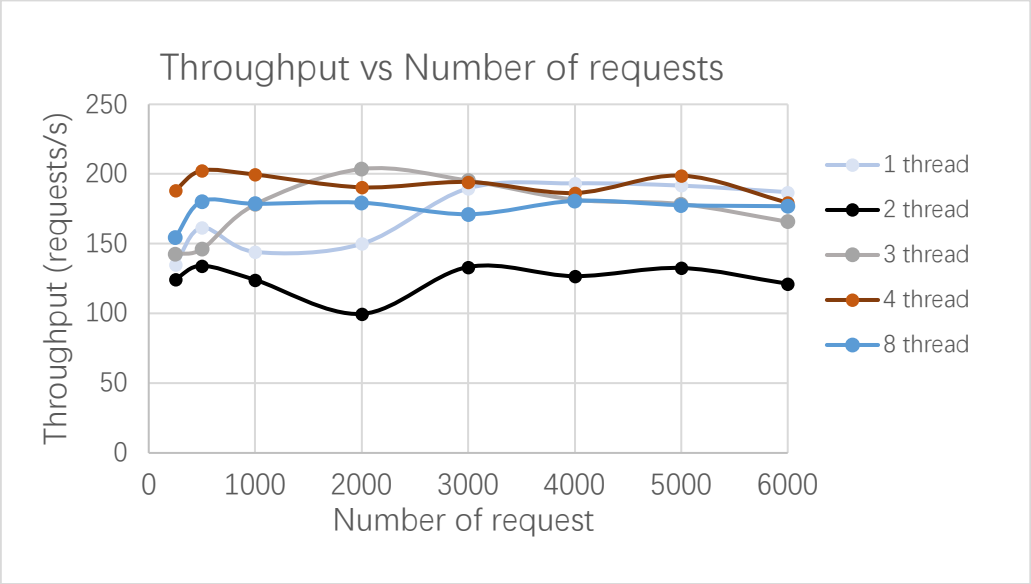

Figure 4 Calculated throughput with processing various request with different server thread number

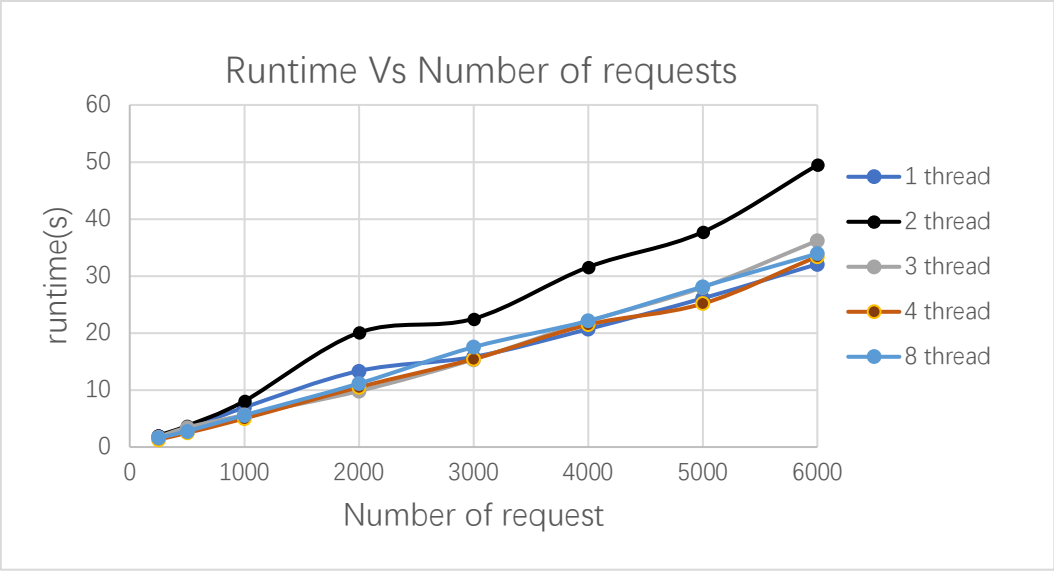

Figure 5 Measured total run time with processing various request with different server thread number

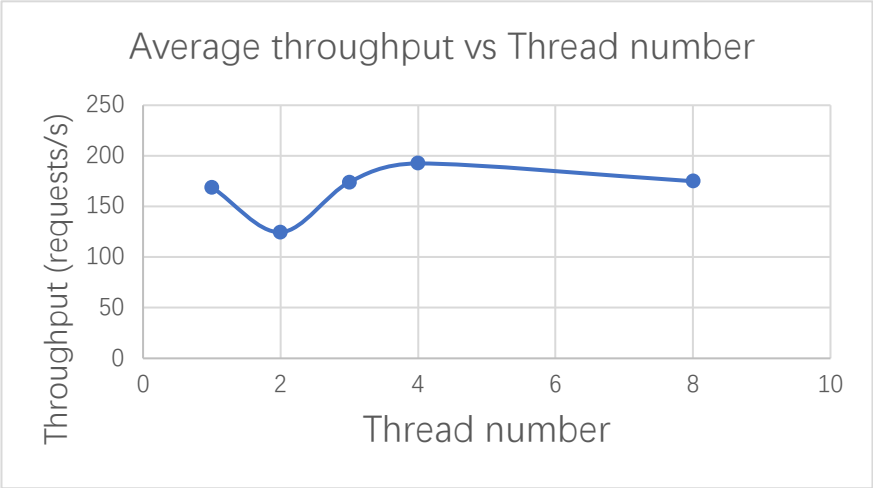

Figure 6 Calculated average throughput with various server thread number

The Figure 4 shows that with the same number of cores, the relationship between the number of request and the throughput in multithread server will all also have a similar pattern, which keep in a constant value (around 120-200 request/s) with little fluctuation. And Figure 5 &6 shows a linear relationship between the runtime and the number of requests. There are 3 potential reasons, the first reason is the client is single thread, which send request sequentially so that although the client can process the request with multithread, it can only receive the request one by one. Therefore, the throughput didn't improve a lot by increasing the number of server thread. The second reason maybe although the thread number increased, there may have overhead for server to manage system resource and assign tasks to each thread in the thread pool. The third reason may because of the system environment will change from time to time and there may be some random error exist in the result (as shown in the 2 threads server has throughput smaller throughput), which can be improved by increase the experiment number of times in the future.

Experiment 3: Multi clients send one request to the server at the same time. This is carried out by implement a thread pool in client side and construct xml file which contain single request and send to the server. And server will put it in the multithread pull to process the request Run time and throughput is measured to indicate the scalability of the server system.

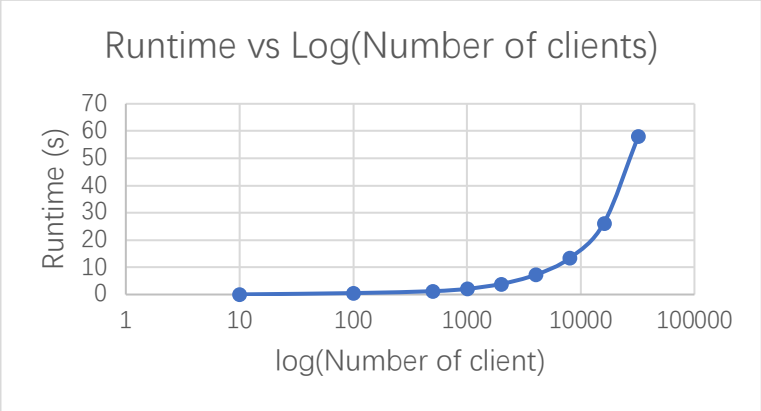

Figure 7 Measure runtime with log scale of client number

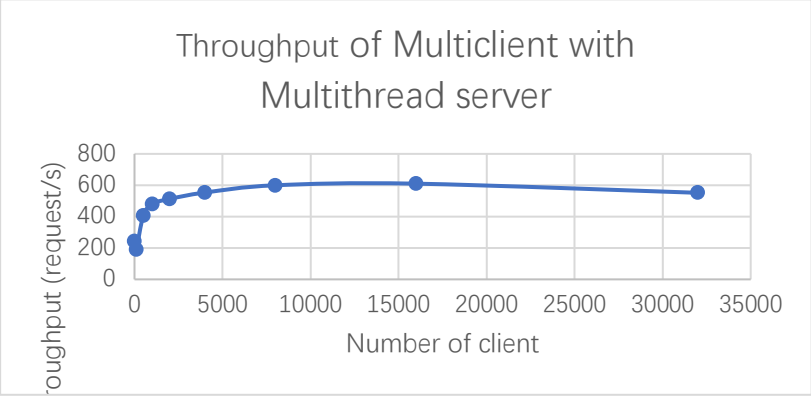

Figure 8 Calculated average throughput with various client number

The results show that the scalability of the server system is around 550 requests/s throughput. If the client number is smaller than 5000, the throughput will increase with the increasing number of client, however, if the client number larger than 5000, the performance of the server system will saturate, so the throughput will not increase and keep around 550, which fit our expectation.

Conclusion:

In the project, the experiments results show that our server system has scalability with throughput around 550 request/s, which fit out expectation. This result may be improved in the future.