HW4 Report - Scalability: Exchange Matching

hw210 Haili Wu

yw314 Yunhe Wang

Abstract:

We implemented this matching server in C++ with PostgreSQL. We use the server-client model to send XML requests from clients and handle requests on the server side. Every time we run the server, it would drop all existed tables and create new tables. Basically, we build our database with 6 tables, which are explained below. The server will respond to the client with the specific information based on the contents of XML sent.

Database:

We selected PostgreSQL as the relational database management system because it has the functionality of multi-version concurrency control and its default isolation level is read-committed, which ensures the consistency of database operation. We constructed 6 tables for conveniently handle our tasks.

ACCOUNT: The table stores information of all account id and its associated balance.

SYM: The table stores information of a symbol and its related account id and amount.

TRANSACTION: The table records the transaction id of each order.

OPEN: The table stores opened transaction orders which are waited for matching.

EXECUTED: The table records executed transaction orders with its detailed information.

CANCELED: The table records canceled transaction orders.

Functionality:

Create:

For a creating account request, the server would check whether an account with this account_id has already existed. For a creating symbol request, the server would check whether this symbol has already existed under this account to decide to insert or update SYM table.

Transaction:

For an order tag request, the server would confirm this order first. If it is a buying order, the server would check the ACCOUNT table to see whether there is enough balance remain in this account to pay for this buying. If it is a selling order, the server would check SYM to see whether there is enough num of this symbol to sell.

After the server confirmed the order, it would try to pair this order with current open orders in the OPEN table. We use a loop to deal with the condition that the new open order can be shared with multiple orders.

For a cancel tag request, the server would select from the OPEN table all orders with this trans_id and insert them into the CANCELED table. We use row lock in this selection so request from other transactions cannot access those selected rows until we commit this transaction.
For a query tag request, the server would select orders from OPEN, CANCEL and EXECUTED tables with this trans_id and return them.

Scalability:

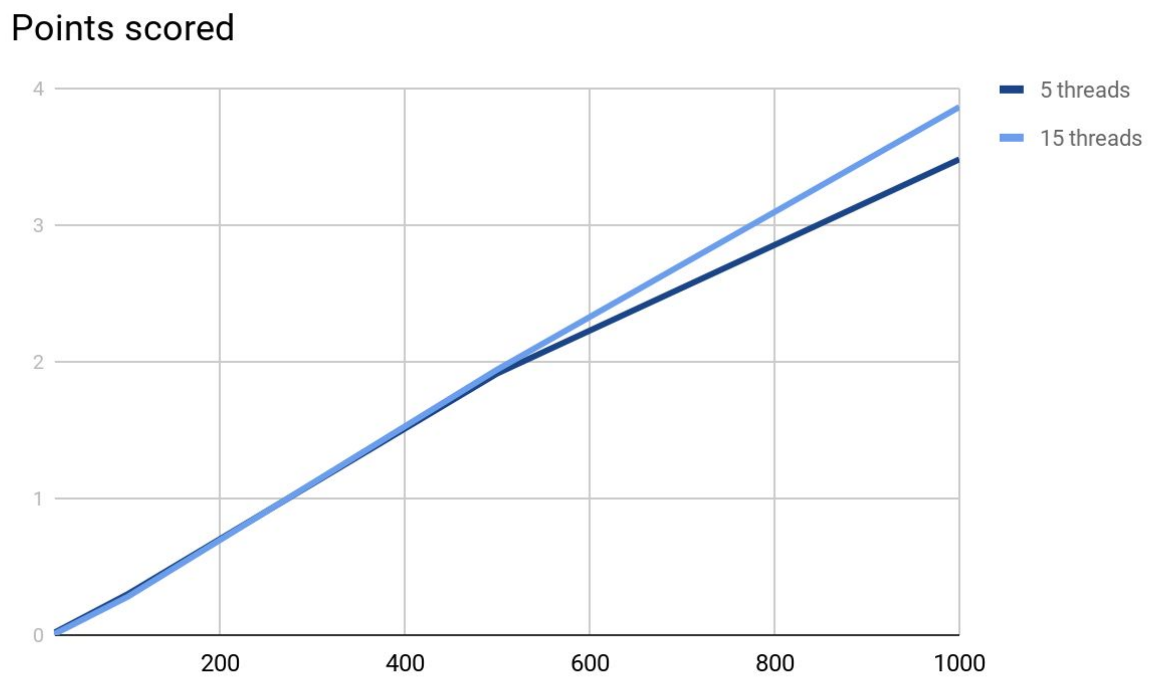
A program is said to scale if it is suitably efficient and practical when applied to large situations. To improve the scalability of our program, we add something to ensure the program works when a quantity increases.
1.Rows level lock
When multiple users(clients in this project) try to access the same area of the database, this might lead to some race conditions. For example, when a new order comes into the database, we need to select all pairable orders from the database and try to pair them with the new order by priorities. But it is not an atomic process. So there is a situation that someone owns one order of those pairable orders cancels his/her order. Then the new order would pair with a canceled order. We avoid this kind of "read-modify-write" situation by adding rows level lock when we select from our database, and this lock would be unlocked after this transaction committed.
2.Thread pool
To ensure good performance of the server when the quantity of input requests increases. We use a multi-threads pool to allocate available threads to requests and enqueue those requests by their arriving time. What we did is "one XML one thread" strategy. Actually, we have considered about using "one request tag one thread" strategy, but it needs to parse XML doc before coming into its own thread. That means the server thread would do too much work for each request and would decrease the performance of the server. Besides that, we use a thread pool to limit the maximum number of threads can be created for request handling. When a big quality of input requests comes in, it might need to create too many threads for those requests, which would lead to stack overflow.

Scalability Analysis

Runtime Table

Thread #\Request #	5	15	30	60
20	0.016s	0.006s	0.01s	0.041s
100	0.300s	0.281s	0.244s	0.338s
200	1.915s	1.944s	1.657s	1.86s
1000	3.481s	3.864s	3.734s	4.186s

As we can see from the runtime table, when the server increases its available threads, the time needed to handle a certain number of requests decreased generally. mainly because of the enque waiting time for each request decreases.But the runtime is not strictly decreasing when we increase the number of available threads. We think it might the condition that requests are not handled fast enough to make full use of that many threads.



Core #\Request#	5	15	30	60
10	0.006s	0.016s	0.041s	0.01s
50	0.301s	0.300s	0.338s	0.244s
500	1.944s	1.915s	1.86s	1.657s
1000	3.864s	3.481s	4.186s	3.734s

For the multiple cores part, we test the server with the different number of requests,but it looks like no big difference between different cores.