ECE568 HW3 Report

XML Parser

In this project, we use the Xerces as the xml parser library.

After initializing the parser, I set the format checker on. So before parsing the fields of the XML, it will first check the completeness of the XML. As the fields of valid XMLs are fixed, if a necessary field is missing, it will see it as invalid. When the parser get an exception, it will not send instructions to the database. Instead it will directly send the client with an error response.

Database:

We designed three tables in our database.

They are as follows:

ACCOUNT(ACCOUNT_ID, BALANCE)

SYMBOL(SYM, OWNER,AMOUNT)

ORDER(ID, SYM, AMOUNT, PRICE, TYPE, STATUS, ACCOUNT_ID, TRANS_ID, TIME)

The owner in symbol has a foreign key relationship to account_id in account.

The account_id in order also has a foreign key relationship to account_id in account.

Create

`createAccount(int id, double balance)`

We would check whether this id already exists in account table and also check whether balance is negative.

`createSymbol(string name, int owner, int amount)`

We would check whether this owner exists in account and check if amount is negative.

If the primary key (name+owner) already exist, we would add the amount into this record.

Transaction

Order

`createOrder(string name, double amount, double price, int account_id)`

If the amount is negative, we set its type to sell.

If the amount is positive, we set its type to buy.

For sell order, we would check whether this user has enough symbols to sell. If enough, we would minus the amount from this symbol record.

For buy order, we would check whether this user has enough balance to buy. If enough, we would minus balance from this user's balance.

And we would do match order every time a new order is opened.

`matchSellOrder(string name, double amount, double price, int account_id,int trans_id)`

To match a sell order, we would search proper buy orders (price>=sell_price) and order them by price DESC and transaction id ASC.

`matchBuyOrder(string name, double amount, double price, int account_id,int trans_id)`

To match a buy order, we would search proper sell orders (price<=buy_price) and order them by price ASC and transaction id ASC.

Match process

When two orders can be matched. If they has equal amount, we just change the status to executed and set time of these two orders.

If one side has amount left, we would set add new line for this order to be executed, and minus amount on the origin order.

Query

`queryOrder(int trans_id, int account_id)`

We would first check if this trans_id exists and belongs to this account_id.

If not, we would send an error message.

Then we would select from orders, and transfer the result into required format.

Cancel

`cancelOrder(int trans_id, int account_id)`

We would first check if this trans_id belongs to this account_id.

If not, we would send an error message.

And then we would change open order status to canceled and set time.

Then we would call query Order to show the result.

Test

Correctness :

To run the following test cases, please enter the command ./client 127.0.0.1 .

We use comments in those files to indicate what we want to test for those lines.

1. test invalid XML

   The test file name is ./testing/invalid.txt.

2. test create XML

The test file name is ./testing/create_test.txt.

3. test transaction XML

   The test file name is ./testing/transactions_test.txt.

Scalability

Method

For the convenience of testing, we use the local database to test the scalability.

To test it locally, we need to change the line in database.cpp from

`C= new connection("dbname=exchange user=postgres password=passw0rd host=db port=5432")`

to , `C= new connection("dbname=exchange user=postgres password=passw0rd")`, as the previous setting is for database in docker.

After that you can use the following command to start the server.

taskset -c 0 ./server // run one core server

taskset -c 0,1 ./server // run two-core server

taskset -c 0-3 ./server // run four-core server

We write a test.sh to have multiple client run at the same time and each client would send 1000 requests. We put them in ./testing and they are named by the number of clients, for example, test32.sh is for 32 clients, etc. We got the time for executing all the request by query the first order and last order and see their time, so as to calculate the running time.

You can run that by:

chmod o+x testing/test32.sh

sudo ./testing/test32.sh

1. **Order 1000 times per client for same amount**

We first let per client to send 1000 order request with the same account. And we got this request.

(We run each mount of request of each kind of core for 5 times and calculate average)

request\core	1 core	2 cores	4 cores
4000	16s	14.6s	14.4s
8000	19.3s	18.75s	55s
12000	28s	28s	28.75s
16000	38s	37.3s	36.25s
20000	47.3s	46s	47s

Then we found that there is little difference among one core, two cores and four cores.

We analyzed the reason and guess it is because that we are executing orders and operating on the same account at multiple threads, which means that the database will lock our threads. In this way, we are just running like on one core.

2. **Query 1000 time per client**

Then we changed our test strategy to make multiple query requests.

However, we still got pretty same performance for one core, two cores and four cores.

request\core	1 core	2 cores	4 cores
4000	2.6s	2.6s	2.6s
8000	5.6s	5s	7s
16000	10.6s	10.6s	12.3s
32000	21.6s	22.3s	21s

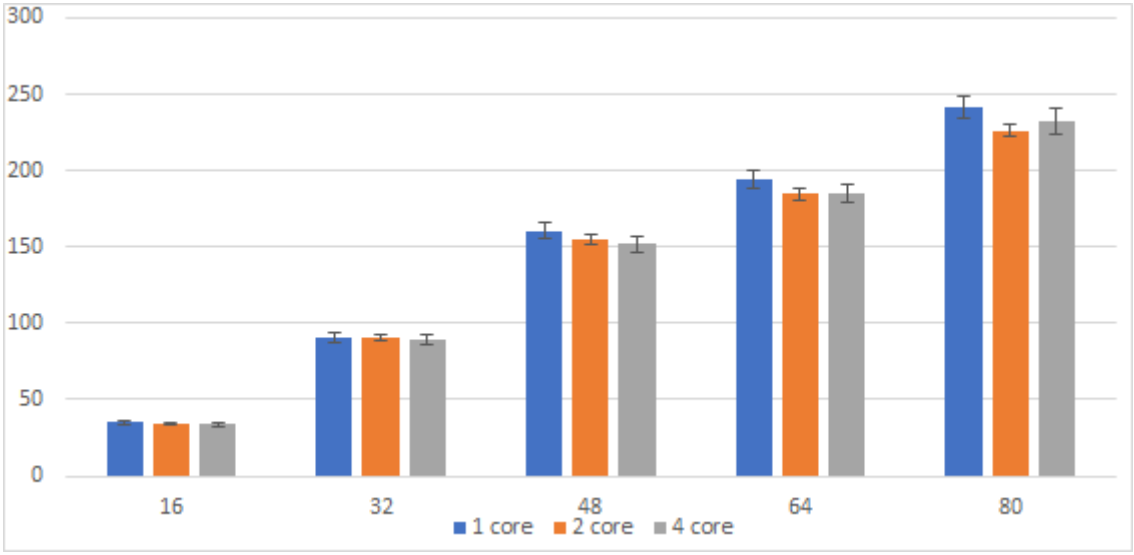
We read some materials and thought maybe we need set the transaction_level of our database to serialized. So we set transaction_level to serialized for every connection and got the following results.

3. **Query 1000 time per client : serialized transaction level**

request\core	1 core	2 cores	4 cores
4000	2.6s	3.6s	3.3s
8000	5.6s	5.3s	7s
16000	11.3s	16.5s	12.6s
32000	22.3s	23s	21s
48000	43s	38.6s	36.6s
64000	49.3s	49s	49s

From the graph, we can see that when the client number is small, there is not much difference in the performance of the server, as the bottleneck is from the request number. While as client number increases, the bottleneck is on the server side. Therefore, the four-core server will perform better than the two-core server than the one-core server and the differences are becoming more obvious.

Here is the result when we let each client send 10,000 requests.



Here the horizontal axis means request amount with unit of 10 thousand, and the vertical axis is running time. You could find out that two core is running faster than one core, and four core is even faster than two cores. But the difference is a bit little and we think the reason is that our request amount is not very big enough and some other measure errors. And the IPC will also drop down the speed when we run on multiple cores.

Request/cores	1core	2core	4core
160000	4571/s	4660/s	4705/s
320000	3529/s	3542/s	3582/s
480000	2987/s	3096/s	3157/s
640000	3287/s	3465/s	3459/s
800000	3305/s	3534/s	3443/s

We calculate the throughput based on our last result, and you could find that on average 4 cores has bigger throughput than 2 cores, and 2 cores is bigger than 1 core.