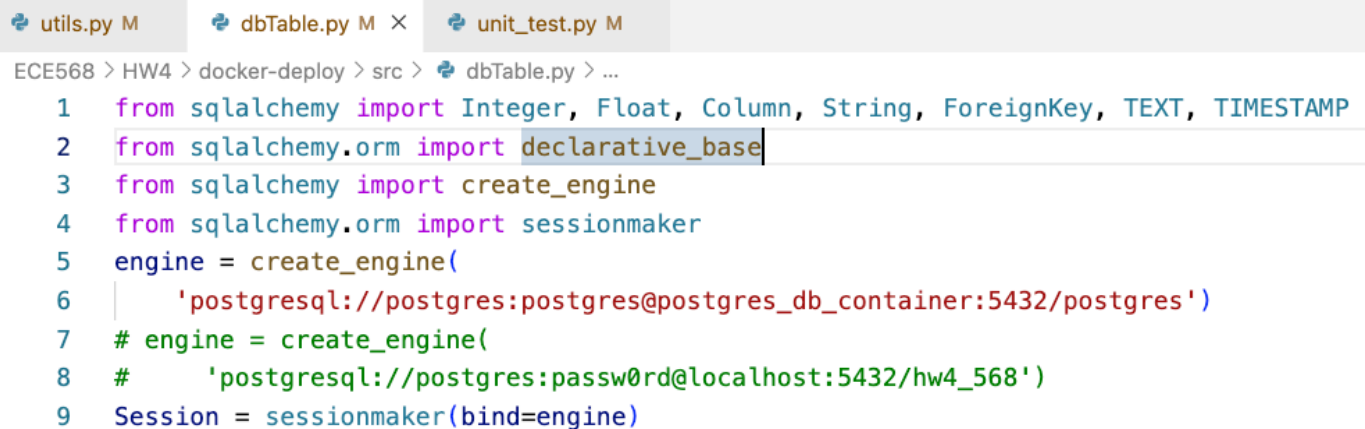


ECE 568 – HW4 danger log:

Name: Fan Yang(fy62), Shiyang Pan(sp645)

General points:

1. For the ORM we use to manipulate the database, which is *sqlalchemy*, it has its own syntax for a database transaction called **Session** which is a class bind with the **engine** connected to the database. Each time we do a database transaction, we should set up a individual **session** object to *add* or *commit*. Originally we using one session and passing into different transaction, which will cause the problem, that the previous exception happens in the session, but not be flushed, and another session is committed, an inside exception is thrown, which will cause the exception giving unclear error message for clients. However, even in this previous version, this error will not make our server down. Further, we change to create a **new session** object for each operation to ensure the session is separate for each transaction. In addition, as we are setting the server to multiprocessing, so we need to **dispose** the engine for each process setup, and this is included in the *Pool* initializer.
2. To ensure the safe modification to the database, we need to have the lock for manipulating row and tuple. However the ORM we used, which will be covered in **Scalability** section, may run into **dead lock** situations. We have to do the `session.commit()` to unlock it. Especially, in some cases, we will detect some illegal situations, and raise the exception to get rid of the current modification, but the lock may be remained, and another request coming in and try to modify it which will cause the dead lock.
3. Although our scalability tests which will mention in the next section, we have already cover all the operations in the **client_test.py**, but if you would like to try out the server **outside the docker**, please set up the engine for a local database, setting in **utils.py**.



```
utils.py M dbTable.py M × unit_test.py M
ECE568 > HW4 > docker-deploy > src > dbTable.py > ...
1 from sqlalchemy import Integer, Float, Column, String, ForeignKey, TEXT, TIMESTAMP
2 from sqlalchemy.orm import declarative_base
3 from sqlalchemy import create_engine
4 from sqlalchemy.orm import sessionmaker
5 engine = create_engine(
6     'postgresql://postgres:postgres@postgres_db_container:5432/postgres')
7 # engine = create_engine(
8 #     'postgresql://postgres:passwd@localhost:5432/hw4_568')
9 Session = sessionmaker(bind=engine)
```

5. For our testing below, we are not running in one time server running, as the testcases are the same but under different conditions. If the server does not restart, it won't drop the database, so the requests in the testcases will receive the error like the account exists. In this case it doesn't perform any transaction to the database, which will run less time than the first execution. You may not be able to reproduce our tests results.
6. The **unit_tests.py** mentioned in the **Functionality** part will provide a bunch of testcases that involves all kinds of potential results according to the specification, Feel free to try out. This is running outside of the docker by it self, so please change the database.

Scalability

As mentioned in the scalability write up, to make our server to be capable to handel multiple concurrent requests, we need to achieve parallelism in our server. As the python could make use of performance through multi-threading, we need to use pre-forked processes with the help of multiprocessing pool. The detailed performance and scalability results is given in the writeup, here we want to elaborate the implementation a bit further.

As we are doing multiple concurrent update and modification to the database. We originally tried to make the database isolation level to **Serializable isolation**, but this will dramastically degrade our performance, so instead of that, we need to change the serializable isolation to the default of the postgres, which is the **Read Committed isolation**, but there will be phantom read in the implementation of our results. For example, if two concurrent requests want to add the same symbol to an account who doesn't already have this stock, they will both try to write a new tuple into this user. However, as specified in our database design, the primary key for this *Position* relation is the *user_id* + *symbol_name*, so postgres will throw the exception. To solve this problem, we use our try except block to catch this exception, and do a rollback, which will result in a right position addition. Additionally, if we are trying to modify one tuple that is already exsiting in the database, we have to ensure this database transaction is atomic, otherwise we may lost updates. For the ORM we used to manipulate the operation to postgres is *sqlalchemy*, which supports a **Row-level lock** by using the syntax **with_for_update**. Most cases, this will help us to do database transactions. However the database relational design, make ordering transaction involved two tables update and different rows updating. There will be lost update even we doing this with the row-level lock. As a result, we introduce the process lock for the ordering transaction, this is not helpful for scalability but this could make sure correctness very well, at the same time we could make sure the parallism for other operations.

As for the multiple core utilization, we used taskset, by speicying the number 0, 0-1, 0-3, we could set the cores we want our server to run on. The results and analysis are given in the writeup.

```
1 | taskset -c 0-1 server.py
```

Functionality correctness of scalability test:

For the functionality of our stock server, we have fullfilled the requirement, and we have unit tests for each database transaction, additionally with the combined unit test by integrating the parser. All these tests are located in the **unit_test.py** in *src* directory, this is a file that you can modify which unit test you want to run in the main function and run independently to see the results.

Additional, we have set up a **print.py** file which could also be ran directly, which will print the database information including all the account information and transaction out. This is easy for you to check the database information.

For the functionality correctness of the testcases given in the scalability writeups, here we will presented the results. This could be reproduce by specifying the corresponding parameters in **server.py** **client.py**, and also set the taskset command when running the **server.py**. As the tests setup given in the setup that:

To test this scalability, we construct a scenario for stock creation. By specifying several total users, all the odd id users will create their account information with a balance of 10000 and create two symbol called "TELSA" and "X", with the shares number 0 and 100 respectively. They will also place their two buy orders each with the symbol of TELSAs and 50 amount, 100 dollars. On the contrary, all the even id users will create their account with 0 balance, and a "TESLA" symbol with 100 amounts. Similarly, they will place a sell order with "TESLA" symbol, with 100 shares and 100 dollars.

These two parts are ensuring the functionality of the transaction matching, after the running, the result of the accounts information should be that the odd users should have a 0 balance, and have 100 shares of TESLA, at the same time the even users will own the 10000 balance, and 0 shares. As the orders are matched together, so the balance and shares will be flipped for the odd and even users.

As the scalability tests given on different condition, here we will only present the results of 10 users running on serializable and concurrent requests: The left 1/3 is the running server, and the middle is the print of the database, before and after client connected, the right is the running time of the client. To make the running time more accurate, we disable the request and response printing, you could enable this in **client_test.py** additionally with the other property like number of users and concurrent or searlize.

Concurrent Test:

```
fy62@vcm-30458:~/ECE568/HW4/docker-deploy/srcs$ python3 server.py
Drop tables successfully
Server is listening on 0.0.0.0:12345

fy62@vcm-30458:~/ECE568/HW4/docker-deploy/srcs$ ./print.py
id | balance | symbol | amount
1 0.0 X 100
1 0.0 TELSAs 100
2 0.0 TELSAs 0
3 0.0 TELSAs 100
3 0.0 X 100
4 0.0 TELSAs 0
5 0.0 TELSAs 100
5 0.0 X 100
6 0.0 TELSAs 0
7 0.0 TELSAs 100
7 0.0 X 100
8 0.0 TELSAs 0
9 0.0 TELSAs 100
9 0.0 X 100
10 0.0 TELSAs 0
tid | uid | amount | limit | symbol | sid | name | shares | p
rice | time
open orders:
sell orders:
buy orders:
Executed orders:
Canceled orders:

fy62@vcm-30458:~/ECE568/HW4/docker-deploy/srcs$ ./print.py
id | balance | symbol | amount
1 0.0 X 100
1 0.0 TELSAs 100
2 0.0 TELSAs 0
3 0.0 TELSAs 100
3 0.0 X 100
4 0.0 TELSAs 0
5 0.0 TELSAs 100
5 0.0 X 100
6 0.0 TELSAs 0
7 0.0 TELSAs 100
7 0.0 X 100
8 0.0 TELSAs 0
9 0.0 TELSAs 100
9 0.0 X 100
10 0.0 TELSAs 0
tid | uid | amount | limit | symbol | sid | name | shares | p
rice | time
open orders:
sell orders:
buy orders:
Executed orders:
2 2 -100 100.0 TELSAs 2 executed -50 100.0 2023-04-06 15:09:05
2 2 -100 100.0 TELSAs 12 executed -50 100.0 2023-04-06 15:09:05
4 4 -100 100.0 TELSAs 4 executed -50 100.0 2023-04-06 15:09:05
4 4 -100 100.0 TELSAs 15 executed -50 100.0 2023-04-06 15:09:05
6 6 -100 100.0 TELSAs 6 executed -50 100.0 2023-04-06 15:09:05
6 6 -100 100.0 TELSAs 18 executed -50 100.0 2023-04-06 15:09:05
7 8 -100 100.0 TELSAs 7 executed -50 100.0 2023-04-06 15:09:05
7 8 -100 100.0 TELSAs 21 executed -50 100.0 2023-04-06 15:09:05
9 10 -100 100.0 TELSAs 9 executed -50 100.0 2023-04-06 15:09:05
9 10 -100 100.0 TELSAs 24 executed -50 100.0 2023-04-06 15:09:05
11 1 50 100.0 TELSAs 11 executed 50 100.0 2023-04-06 15:09:05
12 3 50 100.0 TELSAs 13 executed 50 100.0 2023-04-06 15:09:05
13 5 50 100.0 TELSAs 14 executed 50 100.0 2023-04-06 15:09:05
14 7 50 100.0 TELSAs 16 executed 50 100.0 2023-04-06 15:09:05
15 9 50 100.0 TELSAs 17 executed 50 100.0 2023-04-06 15:09:05
16 1 50 100.0 TELSAs 19 executed 50 100.0 2023-04-06 15:09:05
17 3 50 100.0 TELSAs 20 executed 50 100.0 2023-04-06 15:09:05
18 5 50 100.0 TELSAs 22 executed 50 100.0 2023-04-06 15:09:05
19 7 50 100.0 TELSAs 23 executed 50 100.0 2023-04-06 15:09:05
20 9 50 100.0 TELSAs 25 executed 50 100.0 2023-04-06 15:09:05
Canceled orders:
1 1 -100 100.0 X 1 canceled -100 100.0 2023-04-06 15:09:05
3 3 -100 100.0 X 3 canceled -100 100.0 2023-04-06 15:09:05
5 5 -100 100.0 X 5 canceled -100 100.0 2023-04-06 15:09:05
8 7 -100 100.0 X 8 canceled -100 100.0 2023-04-06 15:09:05
10 9 -100 100.0 X 10 canceled -100 100.0 2023-04-06 15:09:05

fy62@vcm-30458:~/ECE568/HW4/docker-deploy/srcs$

fy62@vcm-30458:~/ECE568/HW4/docker-deploy/src/testing$ ./client
t_test.py
Running time is 0.7647085189819336
fy62@vcm-30458:~/ECE568/HW4/docker-deploy/src/testing$
```

```
dbTable.py client_test.py X print.py M
ECE568 > HW4 > docker-deploy > src > testing > client_test
1  #!/usr/bin/env python3
2  import xml.etree.ElementTree as ET
3  import socket
4  import time
5  import threading
6  SERVER_ADDR = 'localhost'
7  USER_NUM = 10
8  set_concurrent = True
9  set_serialize = False
10 set_print = False
11
```

You may need to close the server, and rerun the server, then run the modified the client_test. As the client won't help you drop the tables.

Searializable Test:

```
fy62@vcm-30458:~/ECE568/HW4/docker-deploy/src$ python3 server.py
Drop tables successfully
Server is listening on 0.0.0.0:12345
[]

fy62@vcm-30458:~/ECE568/HW4/docker-deploy/src$ ./print.py
id | balance | symbol | amount
tid | uid | amount | limit | symbol | sid | name | shares | p
rice | time
open orders:
sell orders:
buy orders:
Executed orders:
Canceled orders:
fy62@vcm-30458:~/ECE568/HW4/docker-deploy/src$ ./print.py
id | balance | symbol | amount
1 0.0 TSLA 100
1 0.0 X 100
2 10000.0 TSLA 0
3 0.0 TSLA 100
3 0.0 X 100
4 10000.0 TSLA 0
5 0.0 X 100
5 0.0 TSLA 100
6 10000.0 TSLA 0
7 0.0 X 100
7 0.0 TSLA 100
8 10000.0 TSLA 0
9 0.0 X 100
9 0.0 TSLA 100
10 10000.0 TSLA 0
tid | uid | amount | limit | symbol | sid | name | shares | p
rice | time
open orders:
sell orders:
buy orders:
Executed orders:
2 1 50 100.0 TSLA 2 executed 50 100.0 2023-04-06 15:12:33
3 1 50 100.0 TSLA 3 executed 50 100.0 2023-04-06 15:12:33
4 2 -100 100.0 TSLA 4 executed -50 100.0 2023-04-06 15:12:33
4 2 -100 100.0 TSLA 5 executed -50 100.0 2023-04-06 15:12:33
6 3 50 100.0 TSLA 7 executed 50 100.0 2023-04-06 15:12:33
7 3 50 100.0 TSLA 8 executed 50 100.0 2023-04-06 15:12:33
8 4 -100 100.0 TSLA 9 executed -50 100.0 2023-04-06 15:12:33
8 4 -100 100.0 TSLA 10 executed -50 100.0 2023-04-06 15:12:33
10 5 50 100.0 TSLA 12 executed 50 100.0 2023-04-06 15:12:34
11 5 50 100.0 TSLA 13 executed 50 100.0 2023-04-06 15:12:34
12 6 -100 100.0 TSLA 14 executed -50 100.0 2023-04-06 15:12:34
12 6 -100 100.0 TSLA 15 executed -50 100.0 2023-04-06 15:12:34
14 7 50 100.0 TSLA 17 executed 50 100.0 2023-04-06 15:12:34
15 7 50 100.0 TSLA 18 executed 50 100.0 2023-04-06 15:12:34
16 8 -100 100.0 TSLA 19 executed -50 100.0 2023-04-06 15:12:34
16 8 -100 100.0 TSLA 20 executed -50 100.0 2023-04-06 15:12:34
18 9 50 100.0 TSLA 22 executed 50 100.0 2023-04-06 15:12:34
19 9 50 100.0 TSLA 23 executed 50 100.0 2023-04-06 15:12:34
20 10 -100 100.0 TSLA 24 executed -50 100.0 2023-04-06 15:12:34
20 10 -100 100.0 TSLA 25 executed -50 100.0 2023-04-06 15:12:34
Canceled orders:
1 1 -100 100.0 X 1 canceled -100 100.0 2023-04-06 15:12:33
5 3 -100 100.0 X 6 canceled -100 100.0 2023-04-06 15:12:33
9 5 -100 100.0 X 11 canceled -100 100.0 2023-04-06 15:12:34
13 7 -100 100.0 X 16 canceled -100 100.0 2023-04-06 15:12:34
17 9 -100 100.0 X 21 canceled -100 100.0 2023-04-06 15:12:34
fy62@vcm-30458:~/ECE568/HW4/docker-deploy/src$
```

```
dbTable.py client_test.py X print.py M
ECE568 > HW4 > docker-deploy > src > testing > client_test
1  #!/usr/bin/env python3
2  import xml.etree.ElementTree as ET
3  import socket
4  import time
5  import threading
6  SERVER_ADDR = 'localhost'
7  USER_NUM = 10
8  set_concurrent = False
9  set_serialize = True
10 set_print = False
11
```

From the results, we could see that the database is given as expected, and concurrent test is 0.2s faster than the serializable test.