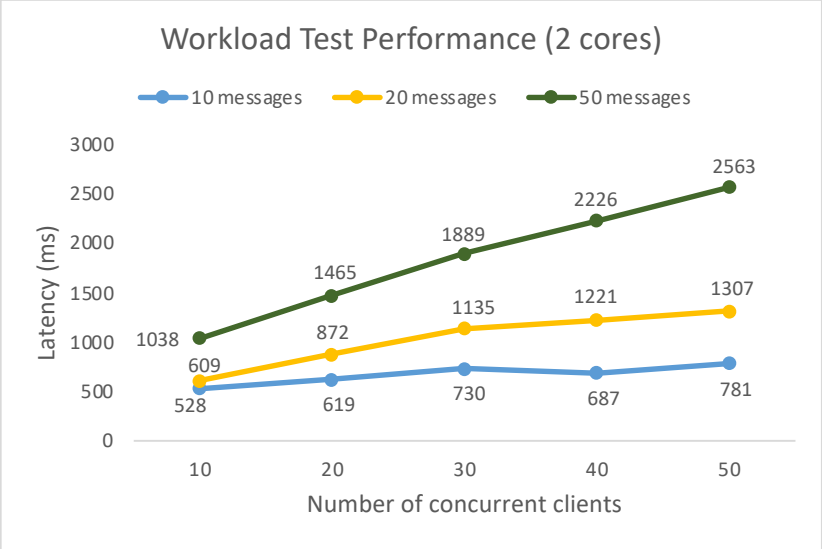
1 Workload Test Performance

We used 2 cores of our VM to test the matching server's concurrency performance.

- The x-axis is the number of concurrent clients.
- The y-axis is the latency in milliseconds.
- We tested three cases where each user sent XML to the server containing 10, 20, and 50 messages.

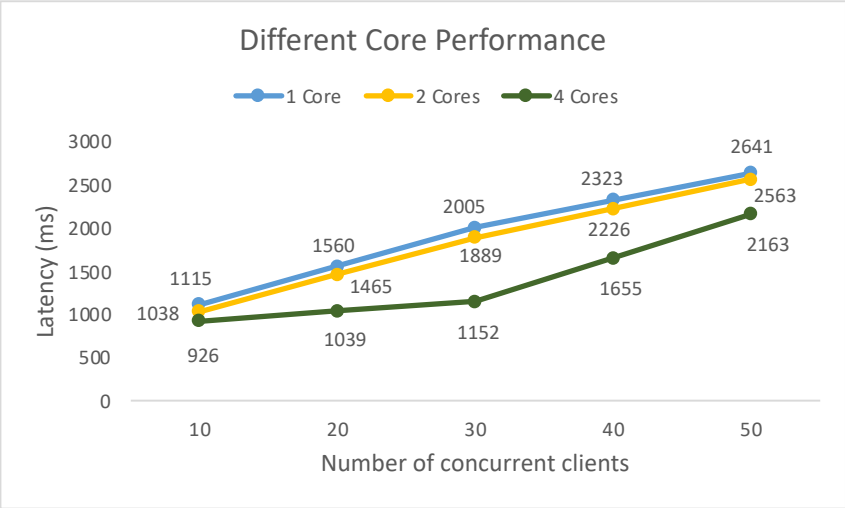


1. As the number of concurrent users increases, the latency generally increases as well. This is expected behavior, as when the server needs to handle more concurrent requests, resource contention will intensify, leading to increased latency.
2. As the number of messages sent by each user increases, the latency also increases. For example, the latency when each user sends 20 messages is generally higher than when each user sends 10 messages. Similarly, the latency is higher when each user sends 50 messages. This indicates that the more messages each user sends, the more tasks the server needs to handle, resulting in increased latency.
3. In some cases, increasing the number of concurrent users results in a decrease in latency, such as when each user sends 10 messages, and the number of concurrent users is increased from 30 to 40, the latency decreases from 730ms to 687ms. This could be due to some adaptive mechanism in the system at play or due to random fluctuations in the statistical data.
4. Overall, the latency shows an upward trend as both the number of concurrent users and the number of messages sent by each user increase. This suggests that the server may require more resources to maintain low latency when handling more users and more messages.

2 Different Core Performance

We used 1 core, 2 cores, and 4 cores respectively for concurrent testing. Again, we set a different number of concurrent users.

- The x-axis is the number of concurrent clients.
- The y-axis is the latency in milliseconds.
- The lines represent the server performance of different cores.



1. As the number of concurrent users increases, the latency also shows an upward trend. This is expected behavior, as when the server needs to handle more concurrent requests, resource contention will intensify, leading to increased latency.
2. Increasing the number of cores does indeed reduce latency. Comparing the data between 1 core and 2 cores, we can see that the latency of 2 cores is generally lower than that of 1 core under the same number of concurrent users. Similarly, there is a similar trend between 2 cores and 4 cores. This indicates that increasing the number of cores helps improve the server's processing capacity, thereby reducing latency.
3. The latency drops noticeably when using 4 cores compared to 2 cores, but not in a linear manner. This suggests that the server may not be fully utilizing the computing power of all cores at 4 cores, or that it may be encountering other bottlenecks (such as I/O, network, etc.).
4. For scenarios with a lower number of concurrent users (e.g., 20 users and 30 users), the latency reduction effect is more pronounced when increasing the number of cores. This might be because, in low concurrency situations, the computational power of a single core has a greater impact on the overall latency. As the number of concurrent users increases, the performance improvement of a single core may be offset by other bottlenecks.

In summary, we can draw the following conclusions:

- Increasing the number of cores can effectively reduce latency and improve the server's processing capacity.
- In low concurrency scenarios, increasing the number of cores has a more pronounced effect on latency reduction.

- As the number of concurrent users increases, the server may encounter other bottlenecks that may limit the effect of latency reduction.

3 Database Concurrency Result

We use optimistic lock, database unique key and set transaction isolation level to achieve database concurrency under the situation like: multiple creation, multiple add, multiple select and update.

User creation and Symbol creation concurrency :
This test set(testing/test1.txt) do create for three account, and we used 10 threads and try to open new account and symbol. To successfully handle this situation, the create for same account id should not be allowed, and the symbol should all be successfully count for 10 times(so final amount = 10 * single amount).

```
postgres=# select * from account;
 account_id | balance | account_num | version
------------+---------+-------------+---------
         42 |    1000 | 1           |       1
         51 |    1000 | 2           |       1
         52 |    1000 | 3           |       1
(3 rows)
```

```
postgres=# select * from position;
 position_id | amount | symbol | account_id | version
-------------+--------+--------+------------+---------
         241 |   1000 | SPY    |         42 |       1
         244 |   1000 | SPY    |         51 |       1
         251 |   2000 | USD    |         42 |       1
         258 |   2000 | USD    |         51 |       1
         262 |   2000 | USD    |         52 |       1
(5 rows)
```

You may also notice that we did not use the id that given by user to be the primary key and also join with account table. This is because it is very dangerous to let any user know what their id is in the database, which may leaking more information(eg. how large is the database) and be used by attacker. So we made another column account_num mapping with the user input id.

Other concurency like combination of matching, cancle, query, order, matching is also supported in parallel by achieving optimistic lock , the version in the table is the way we match the updated information.