**Evaluation Doc**

Steps to run on AWS:

1. Access student account via canvas and set up session as indicated in hw 6
2. Create instance:
   1. aws ec2 run-instances --image-id ami-0d73480446600f555 --instance-type m5a.large --key-name vockey > instance.json
3. Authorize port 22 (SSH) and port 8001 which is what my frontend server runs on
   1. aws ec2 authorize-security-group-ingress --group-name default --protocol tcp --port 8001 --cidr 0.0.0.0/0
4. SSH into 5 instances (catalog, frontend, 3 replicas)
5. Clone git repo
6. Install python 3.7 which is needed for http.server
7. Run server

Results:

I ran with 5 clients on my local machine and with the server on aws. I measured average latency with cache and without for each different function (query, buy, and GET order). Below is the graph of the results. Running with cache is in the solid line and without cache in the dotted lines.

We see that buy requests have the highest latencies and query request are faster. This is expected as the buy requests involve a write lock and needs to both read the data and write to it, whearas the query requests simply use a read lock that reads the data and can do so concurrently with multiple readers.

Additionally, we see that the cache noticeably speeds up performance. The element of main interest is its effect on query which is what the cache is used for. We do see that it improves performance (on average 12% performance improvement).

Finally, I also varied the parameter p (probability of buying from 0 to .8). Each session for each client included 50 queries and potential buys. Obviously, there are no buys/orders when p = 0. As p increases, we do see fluctuations, but there is not too clear of a trend of how latency correlates with parameter p.

Part 3 Fault tolerance:

I showcased already in my output file, but I will display the results here as well. I tested with every combination of simulating failure and reinstating later on. Everything worked as intended where a new leader is elected, it resumes duty right where the failed node left off, and on the client end nothing is noticed. The cases of highest interest were where the leader node failed. I documented this case with screenshots.

Here is the output for a failure simulation where replica 3 fails after 5 orders. The screenshots are of replica 3, replica 2, and the client respectively. We see that replica2 picks off where replica3 left off, and from client perspective, nothing is detected and everything works as intended.

Text

Description automatically generated

A screenshot of a computer

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated