

Hardware Configurations

CPU Speed	2.90 GHz
Number of cores	2
RAM	16.0 GB (15.9 GB usable)
Processor	Intel(R) Core i7 - 7500U
OS	Windows 10
Disk Read Speed Range	20-40 MB/s
Disk Write Speed Range	100-200 KB/s
Disk Average Response Time	1.0 ms
Capacity of Disk	932 GB

Speeds


Tweets inserted per second (average)	10.07
Home Timelines retrieved per second (average)	0.0115 (about 86.6 seconds per timeline)

Data Model Assumptions

For the million tweets we generated, we choose to have 6000 users because we assumed that on average a user has about 160 tweets. We did research that led us to believe that the number of tweets that a user made is highly variable, but as shown in the table below a vast majority of twitter users, 80.61% to be precise, have tweeted less than 500 times. The table also illustrates that as the number of tweets made rises, the percentage of Twitter users who fit into that category goes down exponentially.

# Tweets made	% Twitter users	Accumulative %
0 - 499	80.61	80.61
500 - 999	7.17	87.78
1000 - 4999	9.53	97.31
5000 - 9999	1.64	98.95
10000 - 19999	0.75	99.70
20000 - 24999	0.12	99.82
25000+	0.18	100.00

Source: sysomos.com



Due to our relatively small sample size of a million tweets, compared to the number of tweets twitter has accumulated, we choose to skew the distribution towards the lower end of the number of tweets made per account. Furthermore, we also assumed that the number of tweets per account was random because the number of followers is indicative of how famous a person is, not how active on twitter they are. In other words, some users with very few followers use Twitter as a soapbox to announce every idea they have, while users with tens of millions of followers rarely post anything. Therefore, by randomly assigning tweets to users some will have almost no tweets, some will have a lot of tweets, and the vast majority of users will have an average number of tweets.

On the other hand, we fit the number of followers an account had to an exponential distribution because we figured that most Twitter users had very few followers, while very few users have a lot of followers. So we decided to fit the distribution to a few points; (0,1), (4200,25), (5700,50), and (6000,80), representing that the user with the user_id 0 is being followed by 1% of the total users, the user with the user_id 4200 is being followed by 25% of the total users, etc. Finally, we decided to use the distribution:

$$Y = 3.003542 - (-0.0005111736/-0.0007832683)*(1 - e^{(+0.0007832683*X)})$$

$$X = \text{user_id}$$

$$Y = \text{percent of total users following that user}$$

These implications vastly affected our results, in both good and bad ways. By choosing to skew the distribution of the number of tweets an average user has towards the lower end of the number of tweets made per account, it meant each user has fewer tweets, on average, which makes retrieving the home timeline for the average user quicker, as there are fewer tweets to join. By assuming the number of tweets an account makes is random, it made the time it takes to retrieve the home timeline more standard because the vast majority of users have around the average number of tweets, 160. By assuming the number of followers was an exponential distribution where the majority, 70%, of users are being followed by less than 25% of the total users it also made the time it takes to retrieve the home timeline more standard because most users do not have that many followers, therefore, less joins to perform. I do not believe the performance of inserting tweets changes as our database fills up because inserting a tuple has nothing to do with how many tuples a table already has, but the performance of retrieving home timelines gets drastically worse as the tweets and followers database fills up because getting a home timeline requires a few joins, and the more tuples there are to join, the slower the query will be. If we had been asked to create a billion tweets, our computers would not have reacted well. It would have made the process of loading tweet tuples, at our current rate of 10.07 tweet inserts per second into the database extremely long and retrieving a home timeline impossibly long because adding more tweets would increase the amount of work every join would have to do in retrieving a home timeline.

We were incredibly far from optimal performance - reading all million tweets from a disk only took 1.842 seconds, 542,888.165 tweets per second, which was much quicker than when we inserted tweets into MySQL, it was only able to insert 10.07 tweets per second. Reading 10 tweets from a disk only took 25 milliseconds, while retrieving a home timeline from MySQL took

86,600 milliseconds on average. Although, we should keep in mind that these are different actions because retrieving a home timeline requires finding out who follows a user, collecting all their tweets, ordering them by date, and returning the latest 10. In conclusion, our results indicating a vital need to find another database technology that can handle these requests quickly and efficiently.

Speed Improvement Analysis

Originally we started inserting tweets into the database without any indexes or primary keys and we were able to insert 8.50 tweets per second. We tried many solutions to increase the speed of inserts. Firstly, we cleaned up our code to make sure that our tweet posting method was as basic and efficient as possible, which meant simplifying the random date generation and removing pieces of code that would increase the runtime of posting a tweet. That sped up our inserts per second to 10.07 inserts per second. Next, we tried adding an index, on tweet_ts, which sped up our home timeline requests but slowed down our tweet inserts to 9 inserts per second, so we decided to get rid of the index because we wanted to prioritize increasing the speed of our tweet inserts. Finally, we tried adding a primary key on the tweets table, which also sped up our home timeline requests but slowed down our tweet inserts to 9.67 inserts per second, so we decided to get rid of the primary key because we wanted to prioritize increasing the speed of our tweet inserts. So, in the end, we settled on using a schema that did not involve indexing or primary keys because these may increase our home timeline requests at the sacrifice of the insert speeds. Ultimately, these results may be worse than expected but we were expecting so due to the hardware limitations, a large amount of user and tweet data, and deficiency of the relational model. In the next assignment, we hope that using a NoSQL database will drastically increase the insert and request speeds for home timelines.