Team Project Phase 2 Report

Summary

Team name: **ThreeCobblers**
Members (names and Andrew IDs):
Leo Guo (jiongtig)
Benny Jiang (xinhaoji)
Yukun Jiang (yukunj)

Live Test Performance & Configurations

Database chosen (MySQL/HBase): MySQL
Number and types of instances: Number of instances: 1 master and 7 workers. All of them are m6g.large.
Cost per hour of the entire system: 0.077 * 7 (7 m6g.large instances)+ (6 * 120 G (Mysql DB data) + 8 G * 7 * 0.1 * G + 2 * 20) / month / 30 days / 24 hours (each instance has disk with 8 G)) + 2 * 0.0225 (kubectl + network load balancer) = 0.6973 (Cost includes on-demand EC2, EBS and ELB costs. Ignore S3, EMR software addition, Network and Disk I/O costs)

Your team's overall rank on the live test scoreboard: We believe that we are in top 3.
Live test submission details:

	M1	M2	M3
score	12.00	20.00	20.00
submission id	717617	717618	717619
throughput	77046.71	56241.94	21178.38
latency	5.27	7.19	19.53
correctness	99.67	100	99.95
error	0	0	0

Rubric

- Each unanswered bullet point = -4%
- Each unsatisfactory answer = -2%
- Use the report as a record of your progress, and then condense it before sharing with us.

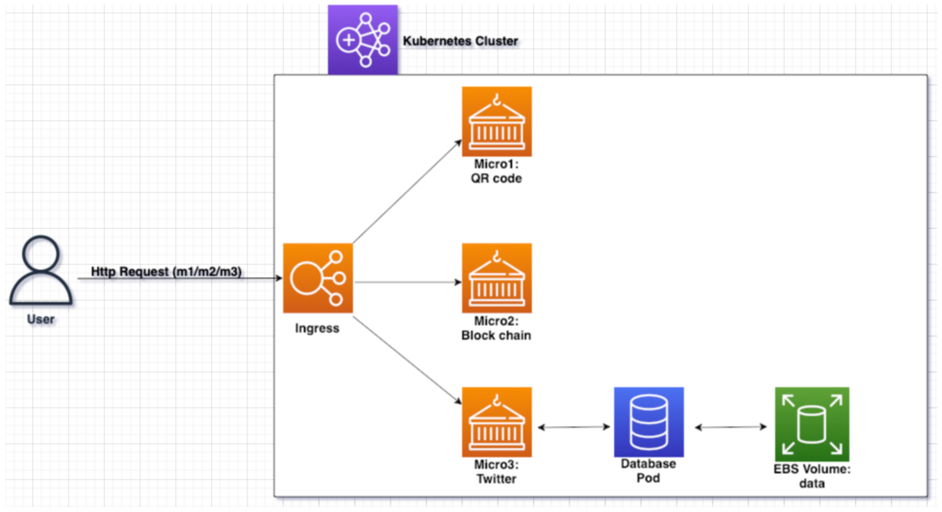
- If you write down an observation, we also expect you to try and explain it.
- Questions ending with “Why?” need evidence (not just logic).
- Always use your own words (paraphrase); we will check for plagiarism.

Task 1: Improvements of Microservices

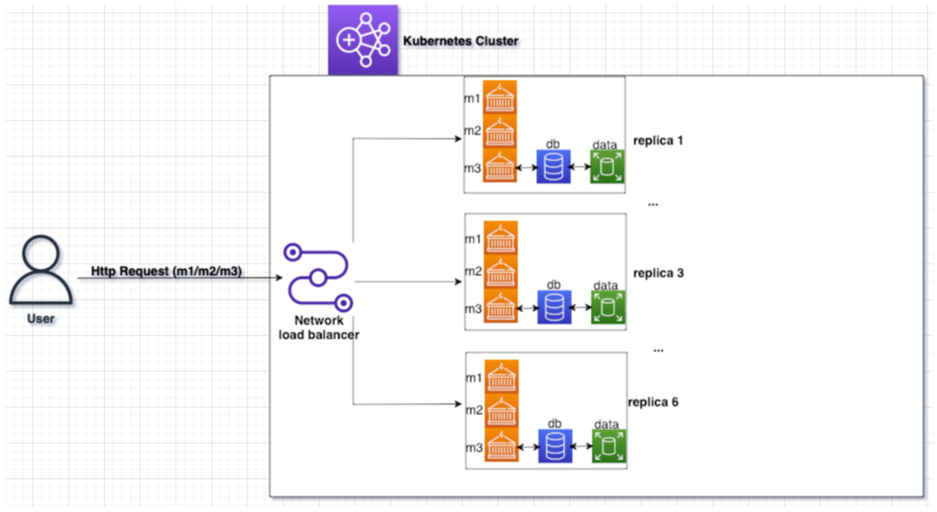
Question 1: Cluster architecture

You are allowed to use several types of EC2 instances, and you are free to distribute the workload across your cluster in any way you'd like (for example, hosting MySQL only on some worker nodes, or having worker nodes of different sizes). When you try to improve the performance of your microservices, it might be helpful to think of how you make use of the available resources within the provided budget.

- List at least two reasonable varieties of cluster architectures. (e.g., draw schematics to show which pieces are involved in serving a request)



Schema1: microservice Ingress routing request



Schema2: monolithic service Network load balancer

- Discuss how your design choices mentioned above affect performance, and why.
  The first schema uses ingress to route http requests to each microservice, and the microservice is split into 1, 2, and 3 individually. Only the pod for microservice 3 is attached with a database and EBS volume containing data.
  The second schema uses a network load balancer to directly route http requests to different replicas of monolithic applications able to serve all 3 services, and each replica is equipped with a backend database and EBS volume containing data.
- Describe your final choice of instances and architecture for your microservices. If it is different from what you had in Phase 1, describe whether the performance improved and why.
  We finally choose the second schema where there are multiple replicas of monolithic applications because we don't know, during the mix test, what's the request ratio between services 1, 2, and 3, therefore in the monolithic architecture, we will be very flexible on this point.

Question 2: Database and schema

If you want to make Microservice 3 faster, the first thing to look at is the database because the amount of computation at the web-tier is not as much as in Microservice 1 and 2. As you optimize your Microservice 3, you will find out various schemas can result in a very large difference in performance! Also, you might want to look at different metrics to understand the bottleneck and think of what to optimize.

- What schema did you use for Microservice 3 in Phase 1? Did you change it in Phase 2? If so, please discuss how and why you changed it, and how did the new schema affect performance?
  Yes, indeed we changed our schema going from phase 1 to phase 2. In phase 1, we store data into three tables, for user info, hashtag and tweets respectively. However, in order to enhance performance, we merge all the information into one big table and the performance's latency improves dramatically. One single query to the database will suffice for one Http request.
- Try to explain why one schema is faster than another.
  We observe that we have an I/O disk burst when adopting the phase 1 schema. And therefore when we switch the phase 2 schema, we only read one table for one http request, it slows down the I/O disk read pressure and improves the service latency greatly.
- Explain briefly the theory behind at least 3 performance optimization techniques (different from the ones in your Phase 1 report) for databases in general. Are they related to the problem you found above?
  Firstly, less query to database is always better since it reduces the pressure on database and disk. Therefore, merging all the information into one table is a very good idea.
  Secondly, caching is also very important, especially for those requests whose return information is very lengthy. If we could cache this information, the next time we will get a free hit and save a lot of network/disk/db access cost.

Thirdly, increasing the key buffer size is also very beneficial because it's for storing the index file, so that we could save the time to load the index from disk and directly read the location of the block we want to access in the memory.

- How is each of these optimizations implemented in your storage-tier? Which optimizations only exist in only one of MySQL and HBase and why? How can you simulate that optimization in the other (or if you cannot, why not)?
  All of the three optimization techniques are implemented in our storage tier and we are using MySQL as the database backend. For HBase, it doesn't support index operation, but we are not using HBase anyway. For such a static data storage and query operation, we believe MySQL as a relational database will outperform NoSQL.

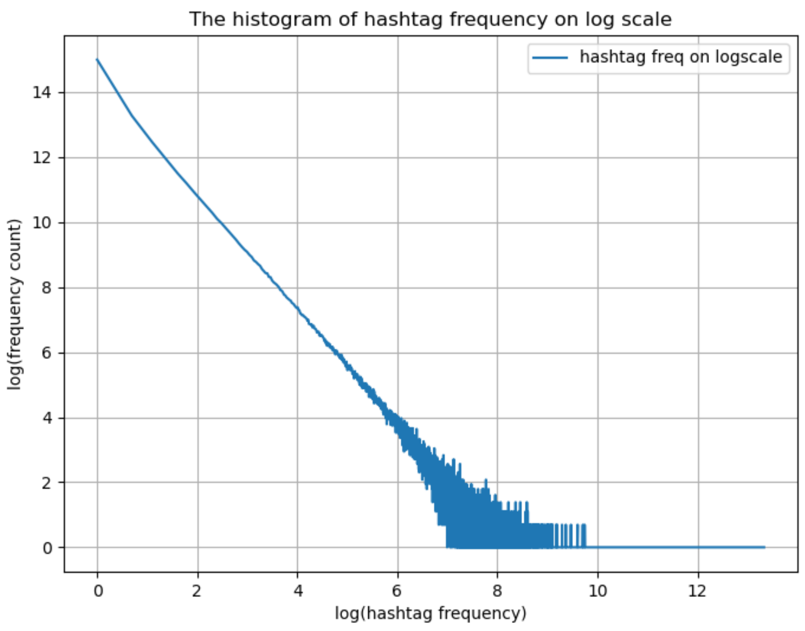
Question 3: Your ETL process

It's highly likely that you need to re-run your ETL process in Phase 2 each time you implement a new database schema. You might even find it necessary to re-design your ETL pipeline if your previous one is not sufficient for your needs. For the following 9 bullet points, please briefly explain:

- The programming model/framework used for the ETL job and justification.
  Still as in Phase 1, We use PySpark on Azure HDInsight Spark Cluster for the ETL pipeline. As we believe PySpark would be very handy for iterative data processing steps, and we have quite a lot of budget left on every team member's account, we believe this would be a wise choice.
- The number and type of instances used during ETL and justification.
  Head Node: E8 V3 (8 Cores, 64 GB RAM) * 2
  Zoo Keeper: A2 V2 (2 Cores, 4 GB RAM) * 3
  Worker Node: E8 V3 (8 Cores, 64 GB RAM) * 10
  Our ETL design idea is quite meticulous and computationally expensive, we put a request to Azure to help increase our HDInsight cluster cpu core limit, therefore we need a rather large and powerful cluster to help do the computation within a reasonable amount of time.
- The execution time for the ETL process, excluding database load time.
  The overall ETL process takes around 3 hours, not including database load time. This time includes data cleaning (filter out malformed tweets), transform the table into the dataframes that we want, outwrite to a .csv static file and collect and store it. In phase 2 specially, we are storing the data all in one table to make the query more efficient.
- The time spent loading your data into the database.
  Now we only have one unified table. We have one .csv distributed file folders to load into three MySQL database tables. It takes around 25 minutes to load into MySQL.
- The overall cost of the ETL process.
  Roughly estimating, we spent a total of around 170$ on the iterative ETL process due to many trials and errors, but this lies within our acceptable range. We spent around 140$ in phase1, and only 30$ in phase2 as we became more experienced in PySpark operation and budget estimation.
- The number of incomplete ETL runs before your final run.

We failed/were unsatisfied with ETL runs around 10 times before we made the final ETL run.

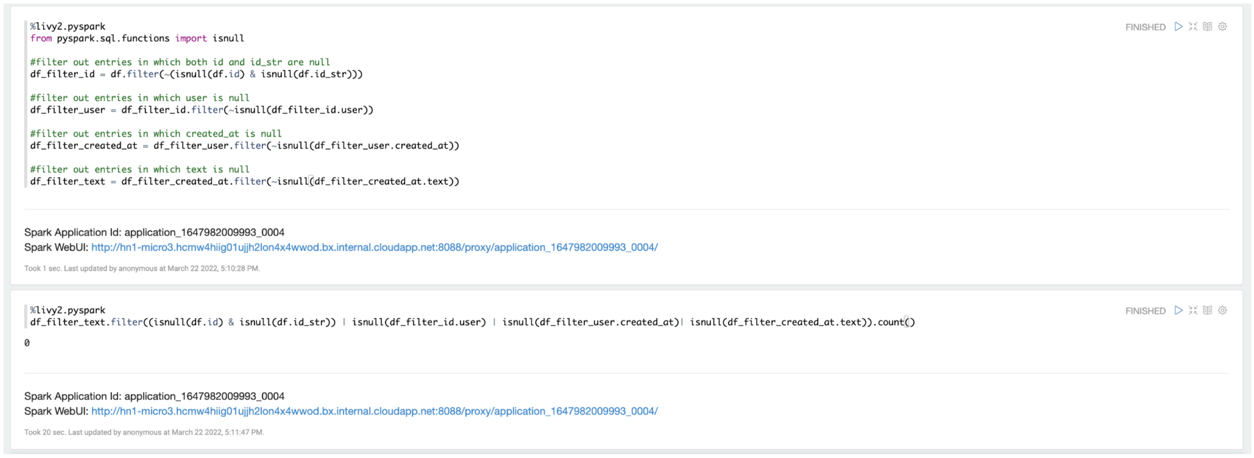
- The size of the resulting database and reasoning.
  The resulting database size is 45 GB in total, not including around 5GB extra storage after making the index since the index is stored as a B-tree structure, it also takes space.
- The size and format of the backup.
  We store the backup .csv distributed file folder in Amazon AWS S3 bucket and therefore we could reload the data when necessary instead of having to run the ETL pipeline again.
- Discuss the difficulties encountered.
  At first, we didn't know how to load a file into a MySQL database. Furthermore, later we only know how to load a single file into a specific database table. Thanks to shell script, we find a way around to load every .csv file in a folder into the database in an automatic way.
  In phase2, we found that our latency is quite high and Disk I/O burst out credit quickly. Therefore we redesign the data schema and redo the ETL once again.
- If you had multiple database schema iterations, did you have to rerun your whole ETL pipeline between schema iterations?
  It depends. Typically speaking, when you change the database schema, at least the last part of your ETL pipeline has to be rerun to get the ideal dataframe and store it as a static file for loading. However, we already plan the schema carefully and hopefully would be able to re-design the database schema and re-run the ETL process, which is time-consuming and constraint on budget. But for phase2, since we are merging 3 existing tables into one big one, it's rather easy to do so.
- How did you load data into your database? Are there other ways to do this, and what are the advantages of each method? Did you try some other ways to facilitate loading?
  The dataset after cleaning is in a distributed mode, i.e. it's scattered in a folder with up to 1000 partitions. We wrote a shell script to load each partition into its belonging MySQL database table. We store the file as a .csv file and load it. As far as we know, we could also store the file as .tsf or .json file and load it into the database. Also, we could use Java/Python programming language to connect to the MySQL database and insert it into the database online dynamically. We choose the static .csv file storage because we want to persist the file and avoid future ETL re-engineering in Phase3. The cost for storing such a static file on an AWS S3 bucket is a relatively inexpensive operation.
- Did you store any intermediate ETL results and why? If so, what data did you store and where did you store the data?
  We envision that the ETL process might take several rounds and want to avoid repeatedly doing the first step of filtering out malformed tweets. Therefore, we store a copy of cleaned-malformed-tweets on the AWS S3 bucket.
- We would like you to produce a histogram of hashtag frequency on **log** scale among all the valid tweets **without hashtag filtering**. Given this histogram, describe why we asked you to filter out the top hashtags when calculating the hashtag score. [Clarification: We simply need you to plot the hashtag frequencies, sorted on the x-axis by the frequency value on the y-axis]



The log-scale hashtag frequency plot is illustrated as above. We could observe that for those popular tags (whose appearing frequency is greater than *e^10*), their frequency value is very tiny, which means they take a minor portion of the whole hashtags set. Therefore, we can safely discard them at low risk. This will help to prevent data imbalance problems introduced by the hot-key and improve the overall ETL pipeline efficiency.

- Before you run your Spark tasks in a cluster, it would be a good idea to apply some tests. You may choose to test the filter rules, or even test the entire process against a small dataset. You may choose to test against the mini dataset, or design your own dataset that contains interesting edge cases. **Please include a screenshot of the ETL test coverage report here.** (We don't have a strict coverage percentage you need to reach, but we do want to see your efforts in ETL testing.)
  Note: Don't forget to put your code for ETL testing under the **etl** folder in your team's github repo **master** branch.

We first filter out tweets with null id/id_str, user, created_at, and text. After doing so, we counted the number of tweets violating the above rules, and we got zero such tweets.



```
%livy2.pyspark
from pyspark.sql.functions import isnull

#filter out entries in which both id and id_str are null
df_filter_id = df.filter(~(isnull(df.id) & isnull(df.id_str)))

#filter out entries in which user is null
df_filter_user = df_filter_id.filter(~isnull(df_filter_id.user))

#filter out entries in which created_at is null
df_filter_created_at = df_filter_user.filter(~isnull(df_filter_user.created_at))

#filter out entries in which text is null
df_filter_text = df_filter_created_at.filter(~isnull(df_filter_created_at.text))
```

Spark Application Id: application_1647982009993_0004
Spark WebUI: http://hn1-micro3.hcmw4hiig01ujjh2lon4x4wwod.bx.internal.cloudapp.net:8088/proxy/application_1647982009993_0004/

Took 1 sec. Last updated by anonymous at March 22 2022, 5:10:28 PM.

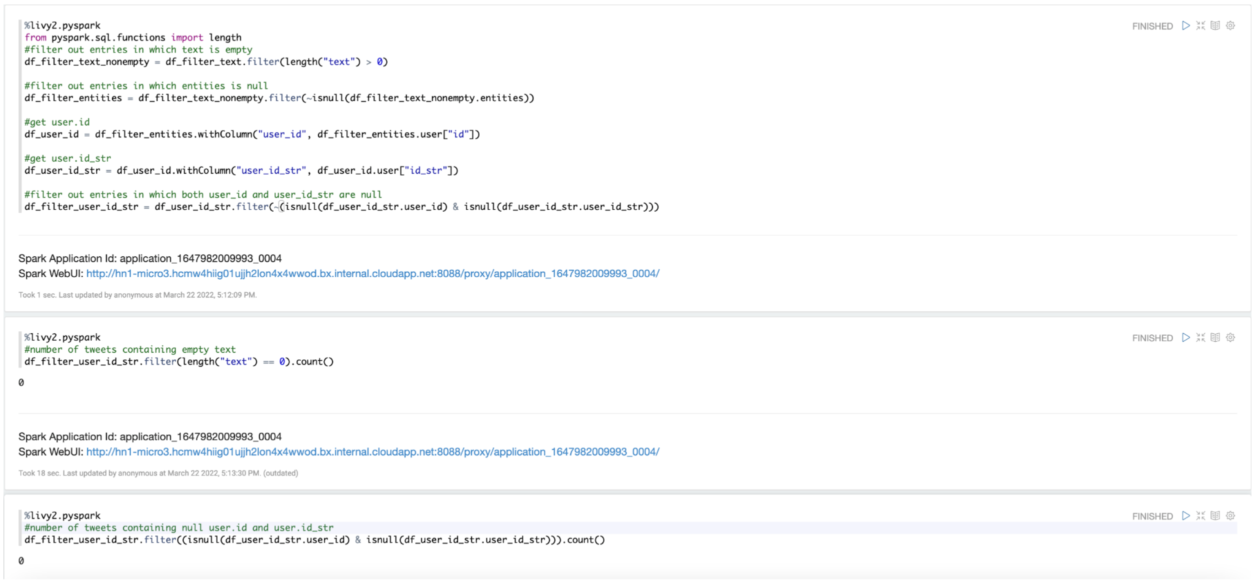
```
%livy2.pyspark
df_filter_text.filter((isnull(df.id) & isnull(df.id_str)) | isnull(df_filter_id.user) | isnull(df_filter_user.created_at)| isnull(df_filter_created_at.text)).count()
```

```
0
```

Spark Application Id: application_1647982009993_0004
Spark WebUI: http://hn1-micro3.hcmw4hiig01ujjh2lon4x4wwod.bx.internal.cloudapp.net:8088/proxy/application_1647982009993_0004/

Took 20 sec. Last updated by anonymous at March 22 2022, 5:11:47 PM.

Then, we filter out tweets with text with zero length, and tweets without user.id/user.id_str. After this, we counted the number of tweets violating the above rules, and got zero such tweets.



```
%livy2.pyspark
from pyspark.sql.functions import length
#filter out entries in which text is empty
df_filter_text_nonempty = df_filter_text.filter(length("text") > 0)

#filter out entries in which entities is null
df_filter_entities = df_filter_text_nonempty.filter(~isnull(df_filter_text_nonempty.entities))

#get user.id
df_user_id = df_filter_entities.withColumn("user_id", df_filter_entities.user["id"])

#get user.id_str
df_user_id_str = df_user_id.withColumn("user_id_str", df_user_id.user["id_str"])

#filter out entries in which both user_id and user_id_str are null
df_filter_user_id_str = df_user_id_str.filter(~(isnull(df_user_id_str.user_id) & isnull(df_user_id_str.user_id_str)))
```

Spark Application Id: application_1647982009993_0004
Spark WebUI: http://hn1-micro3.hcmw4hiig01ujjh2lon4x4wwod.bx.internal.cloudapp.net:8088/proxy/application_1647982009993_0004/

Took 1 sec. Last updated by anonymous at March 22 2022, 5:12:09 PM.

```
%livy2.pyspark
#number of tweets containing empty text
df_filter_user_id_str.filter(length("text") == 0).count()
```

```
0
```

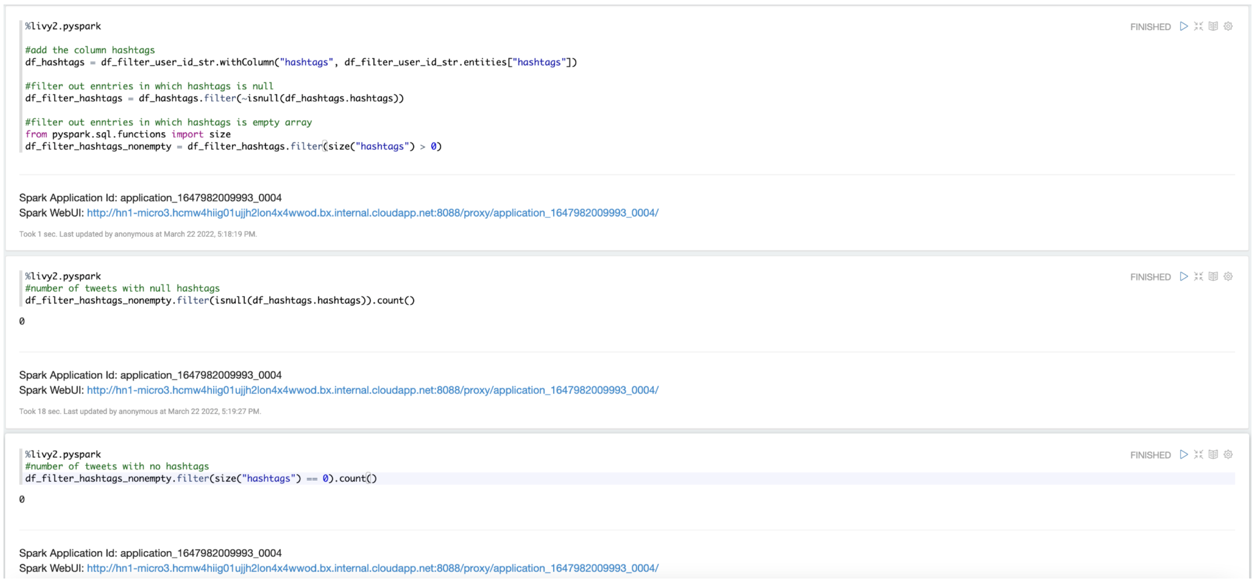
Spark Application Id: application_1647982009993_0004
Spark WebUI: http://hn1-micro3.hcmw4hiig01ujjh2lon4x4wwod.bx.internal.cloudapp.net:8088/proxy/application_1647982009993_0004/

Took 18 sec. Last updated by anonymous at March 22 2022, 5:13:30 PM. (outdated)

```
%livy2.pyspark
#number of tweets containing null user.id and user.id_str
df_filter_user_id_str.filter((isnull(df_user_id_str.user_id) & isnull(df_user_id_str.user_id_str))).count()
```

```
0
```

After this step, we filter out tweets with null hashtags or with zero hashtags. After this, we counted the number of tweets violating the above rules, and got zero such tweets.



```
%livy2.pyspark

#add the column hashtags
df_hashtags = df_filter_user_id_str.withColumn("hashtags", df_filter_user_id_str.entities["hashtags"])

#filter out enntries in which hashtags is null
df_filter_hashtags = df_hashtags.filter(~isnull(df_hashtags.hashtags))

#filter out enntries in which hashtags is empty array
from pyspark.sql.functions import size
df_filter_hashtags_nonempty = df_filter_hashtags.filter(size("hashtags") > 0)
```

Spark Application Id: application_1647982009993_0004
Spark WebUI: http://hn1-micro3.hcmw4hiig01ujjh2lon4x4wwod.bx.internal.cloudapp.net:8088/proxy/application_1647982009993_0004/

Took 1 sec. Last updated by anonymous at March 22 2022, 5:18:19 PM.

```
%livy2.pyspark
#number of tweets with null hashtags
df_filter_hashtags_nonempty.filter(isnull(df_hashtags.hashtags)).count()

0
```

Spark Application Id: application_1647982009993_0004
Spark WebUI: http://hn1-micro3.hcmw4hiig01ujjh2lon4x4wwod.bx.internal.cloudapp.net:8088/proxy/application_1647982009993_0004/

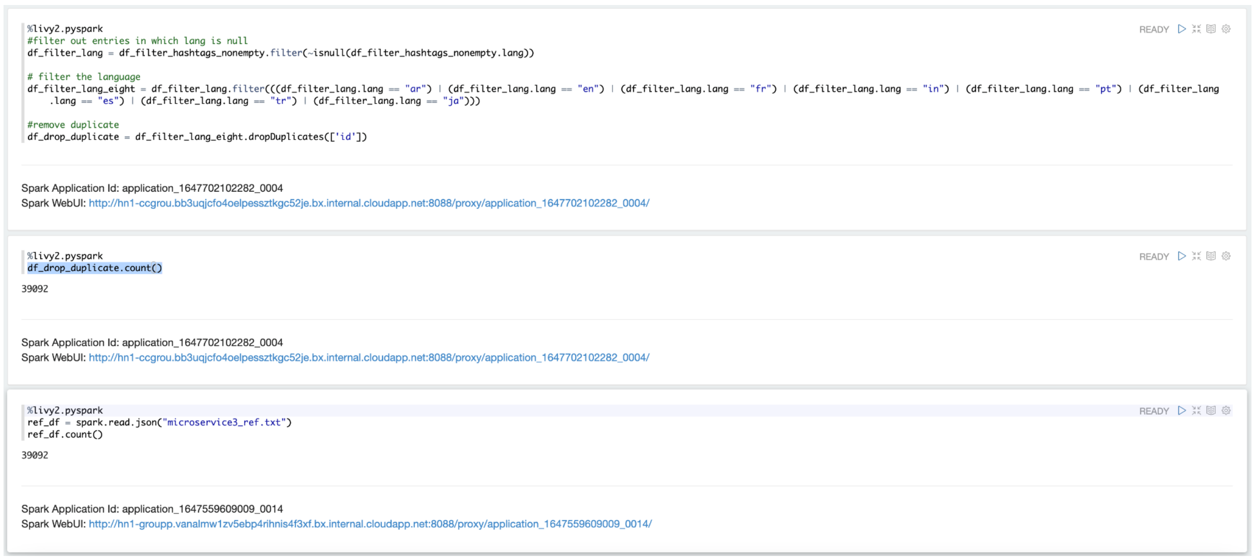
Took 18 sec. Last updated by anonymous at March 22 2022, 5:19:27 PM.

```
%livy2.pyspark
#number of tweets with no hashtags
df_filter_hashtags_nonempty.filter(size("hashtags") == 0).count()

0
```

Spark Application Id: application_1647982009993_0004
Spark WebUI: http://hn1-micro3.hcmw4hiig01ujjh2lon4x4wwod.bx.internal.cloudapp.net:8088/proxy/application_1647982009993_0004/

Finally, we keep tweets with correct language, and remove duplicates. The final dataset contains 39092 tweets, which matches the reference dataset.



```
%livy2.pyspark
#filter out entries in which lang is null
df_filter_lang = df_filter_hashtags_nonempty.filter(~isnull(df_filter_hashtags_nonempty.lang))

# filter the language
df_filter_lang_eight = df_filter_lang.filter(((df_filter_lang.lang == "ar") | (df_filter_lang.lang == "en") | (df_filter_lang.lang == "fr") | (df_filter_lang.lang == "in") | (df_filter_lang.lang == "pt") | (df_filter_lang
    .lang == "es") | (df_filter_lang.lang == "tr") | (df_filter_lang.lang == "ja")))

#remove duplicate
df_drop_duplicate = df_filter_lang_eight.dropDuplicates(['id'])
```

Spark Application Id: application_1647702102282_0004
Spark WebUI: http://hn1-ccgrou.bb3uqjcfo4oelpessztkgc52je.bx.internal.cloudapp.net:8088/proxy/application_1647702102282_0004/

```
%livy2.pyspark
df_drop_duplicate.count()

39092
```

Spark Application Id: application_1647702102282_0004
Spark WebUI: http://hn1-ccgrou.bb3uqjcfo4oelpessztkgc52je.bx.internal.cloudapp.net:8088/proxy/application_1647702102282_0004/

```
%livy2.pyspark
ref_df = spark.read.json("microservice3_ref.txt")
ref_df.count()

39092
```

Spark Application Id: application_1647559609009_0014
Spark WebUI: http://hn1-groupp.vanalmw1zv5ebp4rihnis4f3xf.bx.internal.cloudapp.net:8088/proxy/application_1647559609009_0014/

Question 4: Optimizations and Tweaks

Frameworks and databases do not come in the best shape out-of-the-box. Each of them has tens of parameters to tune. Once you are satisfied with the schema, you can start learning about the configuration parameters to see which ones might be most relevant, and gather data and evidence with each individual optimization. You have probably tried a few in Phase 1. To perform even better, you probably want to continue with your experimentation.

Hint: A good schema will easily double or even triple the target RPS of your Microservice 3 without any parameter tuning. It is advised to first focus on improving your schema and get an okay performance with your best cluster configuration. If you are 10 times (an order of magnitude) away from the target, then it is very unlikely to make up the difference with parameter tunings.

- List as many possible items to configure or tweak that might impact performance, in terms of web framework, your program, DBs, DB connectors, OS, etc.
  1. web/DB connection: connection pool size.
  2. DB: number of tables, replications, storage engine, key buffer size, cache size, mysql version.
  3. OS: x85, or arm system.
- Apply them one at a time and show a table with the microservice you are optimizing, the optimization you have applied and the RPS before and after applying it.

schema (# of tables)	Storage engine	Key buffer size	Cache size	Pool size/server instance	Best 600s RPS
3	Innodb	16M	0	5	1086(one instance)
2	innodb	16M	0	5	375(one instance)
2	MyISAM	16M	0	5	523(one instance)
2	MyISAM	16M	0	30	524(one instance)
2	MyISAM	256M	0	30	478(one instance)
2	MyISAM	512M	0	5	844(one instance)
2	MyISAM	16M	16M	30	723(one instance)
2	MyIsam	16M	8M	5	924(one instance)
1	MyIsam	16M	0	5	7402(one instance)
1	MyIsam	8M	0	5	12916(on cluster)
1	MyIsam	8M	0	2	10584(on cluster)
1	MyIsam	1G	0	2	20832(on cluster)

- For every configuration parameter or tweak, try to explain how it contributes to performance.
  With fewer tables in our schema, we will request fewer queries to MySQL and potentially fewer random accesses to the disk, which will improve performance and keep from IO burst. Increasing key buffer size and cache size will also alleviate the load on disk IO, and speed up query performance. Besides, MyIsam is generally faster than Innodb when the task is read-only.

Task 2: Development and Operations

Question 7: Web-tier orchestration development

We know it takes dozens or hundreds of iterations to build a satisfactory system, and the deployment process takes a long time to complete. We asked about the automated deployment process in the Phase 1 Final Report. Answer the following questions to let us know how your automation has changed in Phase 2.

- Did you improve your web-tier deployment process? What were the improvements or changes you made? Include your improved/changed Terraform scripts, Kubernetes manifest, or Helm charts under the folders referring to Phase 1 folder structures.
  Our web tier can now handle requests from different requests. We mostly change the web-frameworks so that it differentiates and routes different web requests. Other than these, we didn't make any other changes to our web tier.

Question 8: Storage-tier deployment orchestration

In addition to your web-tier deployment orchestration, it is equally important to have automation and orchestration for your storage-tier deployment in order to save labor time and avoid potential human errors during your deployment. Answer the following questions to let us know how your storage-tier deployment orchestration has changed in Phase 2.

- Did you improve your storage-tier orchestration? What were the improvements or changes you made? Include your improved/changed Terraform scripts, Kubernetes manifest or Helm charts under the folders referring to Phase 1 folder structures.
  We used Terraform to automate the provisioning of storage volumes from snapshots. We orchestrated volumes and storage tiers by configuring stateful sets combined with MySQL services, PersistnentVolumes, and PersistenceVolumeClaim in our helm charts. We also applied mysql-read in StatefulSet to distribute connections between storage-tier and web-tier.

Question 9: Live test and site reliability

Phase 2 is evaluated differently than how we evaluated Phase 1. In Phase 2, you can make as many attempts as you want before the deadline. However, all these attempts do not contribute to your score. Your team's Phase 2 score is determined by the live test. It is a one-off test of your web service during a specified period of time, and so you will not want anything to suddenly fail; in case you encounter failure, you (or some program/script) probably want to notice it immediately and respond!

- What CloudWatch metrics are useful for monitoring the health of your system? What metrics help you understand performance?
    1. Number of healthy instances in the autoscaling group.
    2. We can also observe whether there are any drops in disk IO or CPU utilization.
- Which statistics did you collect when logged into the EC2 VM via SSH? Which tools did you use? What do the statistics tell you?
  We observed CPU utilization and available memory through the "top" command. Based on our experience, CPU utilization robustly reflects the performance of our web server. We can try locating the bottleneck by checking whether the CPU is fully utilized.
- Which statistics did you collect using kubectl? What do the statistics tell you?
  We use "describe hpa" to check the CPU utilization of each deployment pod. We also collect the number of pods through kubectl. These statistics are telling us whether the servers are serving requests or are bottlenecked by other components.
- How did you monitor the status of your storage-tier? What interesting facts did you observe?
    1. We check the connections and execution times on MySQL in storage-tier pods by commands like "show processlist". This measure shows whether there are healthy connections between the web tier and the storage tier.
    2. We also observe the disk IO throughput on Cloudwatch. We found that there are occasional drops in IO throughput because EBS throttles disk performance when a burst happens.
- Should something crash in your system, would (and how) your microservices still be able to serve requests (although perhaps slower)? Think about different scenarios, such as program crashes, network failures, or even VM terminations.
  If the disk is throttled, our server can still serve at a lower speed. For program crashes, our program can mostly tolerate it because we have multiple deployments of identical server pods on different machines.
- During the live test, did anything unexpected happen? If so, how big was the impact on your performance, and what was the reason? What did you do to resolve these issues? Did they affect your overall performance?
  Nothing too surprising happens. We had planned what would happen during the live test and how we should react. We have allocated enough resources and disk I/O burst credit for the best performance output during the live test. Therefore everything works as safe and sound as we expected.

Task 4: General questions

Question 10: Scalability

In this phase, we serve read-only requests from tens of GBs of processed data. Remember this is just an infinitesimal slice of all the user-generated content at Twitter. Can your servers provide the same quality of service when the amount of data grows? What if we realistically allow updates to tweets? Here are a few good questions to think about after successfully finishing this phase.

- Would your design work as well if the quantity of data was 10 times larger? What about 100x? Why or why not? (Assume you get a proportional amount of machines.)
  Yes, we definitely can handle 10x times larger data. But since we are using a MySQL database backend with many replicas, we probably would not be able to handle 100x data volume. At that time we would probably have to migrate to NoSQL database schema, for example HBase.
- Would your design work if your web service also implemented insert/update (PUT) requests? Why or why not?
  Yes, our database backend stores the tweets and hashtags necessary for computing data. If we are to support insert/update (PUT) operations, we need to update the data storage accordingly and be careful about read-write through consistency.
- What kind of changes in your schema design or system architecture would help you serve insert/update requests? Are there any new optimizations that you can think of that you can leverage in this case?
  We need to be careful about write-read consistency, i.e. ACID property in the database. Particularly when we have many multiple replicas of the same database, if we need to update, only 1 database will receive this request, yet for consistency purposes, it needs to propagate this information to everybody else to make sure at any given time, the data copy at all database storage is the same.

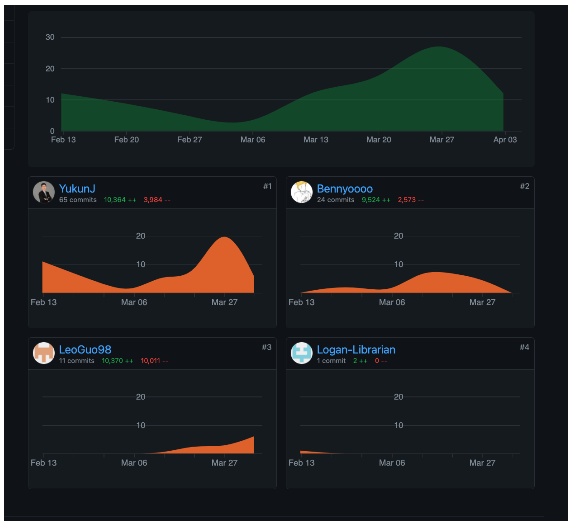
Question 11: Postmortem

- How did you set up your local development environment? Did you install the databases locally on your machines to experiment? Did you test all programs before running them on your Kubernetes cluster on the cloud?
  Yes, surely we did local development testing. We firstly use a local MySQL database and run the java program to directly try connecting with the database. Afterward, we try to make docker images to containerize the service and still use the local database bypassing the "–network host" parameter when running docker. And then we go to the Kubernetes cluster using only 1 work node to test everything works correctly before scaling up.
- Did you attempt to generate a load to test your system on your own? If so, how? And why?
  Yes, we did. It's important to generate loads on our own and test both the correctness and efficiency of our service. We typically use "curl" to send a single HTTP request to our website to test specific queries which we doubt the accuracy of our service. To test efficiency, we write a single HTTP request sender function in Python and run it to test the RPS of our web service.
- Describe an alternative design to your system that you wish you had time to try.
  We could have used NoSQL Hbase as our backend database, it's something worth trying, although we currently believe that it will not perform as well as MySQL.
- What were the toughest roadblocks that your team faced in Phase 2?
  When we had two tables in the database, our performance was bottlenecked by disk IO throughput because we requested tons of disk read, and performance was throttled by EBS. We spent three days and hundreds of experiments to locate the problem and finally resolved it.

- Did you do something unique (any cool optimization/trick/hack) that you would like to share?
  One interesting thing is that we found that when you run docker image service but still want to use a local MySQL database, you could do “--network host” to let them share the same network layer. This comes very handy during the testing and development phase.

Question 12: Contribution

- In Phase 1 Final reports, we asked about how you divided the exploration, evaluation, design, development, testing, deployment, etc. components of your system and how you partitioned the work. Were there any changes in responsibilities in Phase 2? Please show us the changes you have made and each member’s responsibilities.
  In phase 2, since we have basically finished the functionality of the servers and schema, we collaborated more on building the cluster architecture and tweaking database parameters. Benny is still focusing on issues related to the web server, Yukun on databases, and Leo on ETL.
- Please show Github stats to reflect the contribution of each team member. Specifically, include screenshots for each of the links below.

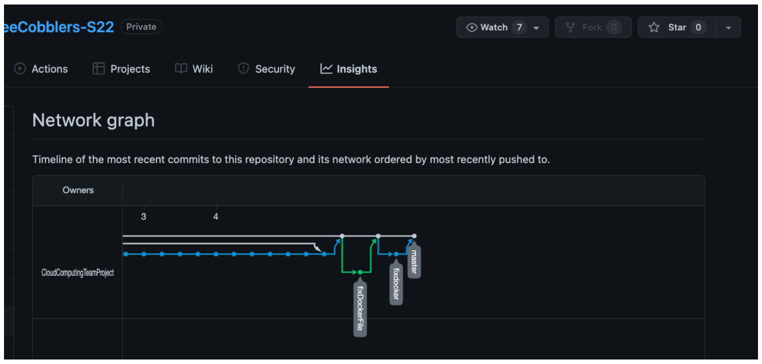


YukunJ #1
65 commits 10,364 ++ 3,984 --

Bennyoooo #2
24 commits 9,524 ++ 2,573 --

LeoGuo98 #3
11 commits 10,370 ++ 10,011 --

Logan-Librarian #4
1 commit 2 ++ 0 --



Network graph

Timeline of the most recent commits to this repository and its network ordered by most recently pushed to.