15619 Project Phase 3 Report

Performance Data and Configurations

Live Test Configuration and	Results
Instance type	m3.large
Number of instances	2
Cost per hour	0.28
Queries Per Second (QPS)	q1/q2/q3/q4/q5/q6/MIX[q1/q2/q3/q4/q5/q6/] INSERT HERE: score[126/0/51/228/234/238/152/0/142/122/138/145] tput [17698.7/16851.3/5441.0/11013.1/17528.1/17881.8/4551.0/ 2478.6/2561.5/2355.4/2475.3/2612.9] latcy [5/2/9/4/2/2/10/9/9/10/9/9] corr [100/0/94/93/100/100/100/0/100/93/100/200] error [0/100/0.01/0/0/0/0/100/0/0/0/0]
Relative Rank [1 - 91] :	Phase 1 :78 Phase 2 :58 Phase 3 :29
Phase Score [out of 100]	====== graded ====== Phase 1:17 Phase 2 Live (selected, mention DB):34(MYSQL) Phase 3 Live:77 ====== others ====== Phase 2 (Pre-Live):26 Phase 2 Live (dropped, mention DB):0(HBase) Phase 3 (Pre-Live):85

Team : cloudlol
Members : Xinyue Chen & Ding Zhao

[Please provide an insightful, data-driven, colorful, chart/table-filled, humorous and interesting final report. This is worth a quarter of the grade for Phase 3. Make sure you spend a proportional amount of time. For instance, if you spent 30 hours building your system, you should spend 10 hours on the report. The best way is to do both simultaneously, make the report a record of your progress, and then condense it before sharing it with us. Questions ending with "Why?" need evidence (not just logic)]

Task 1: Front end
Questions

1. Which front end framework did you use? Explain why you used this solution. [Provide a small table of special properties that this framework/platform provides]
   Undertow. It has following properties:

Property	Detail
Based on Java	Easy for our team to build front-end system, API to connect with Mysql and Hbase is complete in Java. We also have experience building front-end system in Servlet+Tomcat6, which is also supported by Undertow, This makes it easy to transfer our previous work in Tomcat6 into Undertow.
Lightweight	Core jar is small. Even easier to implement with maven.
Non-block IO	Good performance under requirements in this course

2. Explain your choice of instance type and numbers for your front end system.
   We use 1 m3.large instance to build the front-end system.
   In this project, a continuous socket is used to perform the queries and send back response, so it is important to build a low-latency front-end. ELB, provided by AWS, is easy to use but it couldn't perform well as its deliver policy and extra network latency. So I use a single instance to be the front end and most of the time it can satisfy our requirements.
3. Explain any special configurations of your front end system.
   We cached data in the memory of front-end, so it was necessary to enlarge the VM heap space parameter(-XMX7G –XMS7G)
4. What did you change from Phase 1, 2 and why? If nothing, why not?
   In phase 1 we used Tomcat6+Servlet, and it didn't satisfy the minimum requirement of Q1. Then we turned to techempower to see if there's a better front-end framework also based on java. Undertow was on the top of the list, and we learned from the benchmark program provided in the github of techempower. After testing on Q1, we realized that undertow definitely is the right choice. The comparison of performance is in the following Table.

Front-end framework	Q1 performance(QPS)
Tomcat6+Servlet	12632.5
Undertow(httphandler)	17722.2

5. Did you use an ELB for the front-end? Why, or why not? Condense your experience with ELB in the next few sentences.
   I've tried ELB but decided not to use it. As we've done experiments in the weekly projects, ELB couldn't reduce the latency of a continuous connection but in reverse enlarge the latency. In a real world, we can assume that a large quantity of connections come from different clients, thus in this situation it is useful to use ELB to improve the ability to deal with concurrency. However, from our observation, this project use a test service design which is different from our assumption. It uses a fixed number of threads to connect with our server and each thread will not send another get request until it receives response. So the most important thing is to reduce latency. ELB can't help to

reduce latency in theory, nor according to our experiments.

6. Did you explore any alternatives to ELB? List a few of these alternatives. What did you finally decide to use? (if possible) Provide some graphs comparing performance between different types of systems.
   We've considered Nginx. However, we don't have enough time to explore this load balancer or other alternatives (which is proved to be very important to improve the concurrency).
7. Did you automate your front-end? If yes, how? If no, why not?
   No. We've tried but found it hard to set the IP address of back-end system.
   One possible solution is to force the back-end server to upload a file to a specific s3 folder and the front-end system could download this file and automatically set up. This is possible in theory and if we have time, we could make it happen.
8. Did you use any form of monitoring on your front-end? Why or why not? If you did, show us the results.
   We've wrote a script to ping the query address provided by our front-end and if the ping fails, it will send emails to us. This script makes it possible for us to sleep during the live-test.
9. What was the cost to develop the front end system?
   About $10. AWS didn't provide an accurate record grouped by tags so we can only provide an estimation.
10. What are the best reference URLs (or books) that you found for your front-end?
    http://undertow.io/ : the official website of undertow.
    http://www.techempower.com/blog/2014/05/01/framework-benchmarks-round-9/ : it provides the benchmark of all front-end frameworks.
    https://github.com/TechEmpower/FrameworkBenchmarks : the code repository of techempower which is a good start of building undertow.
    http://java-performance.com/ : a good website which gives tips on turning java performance.
11. Do you regret your front-end choice? If yes, what would you change in the way you approached Q1?
    No.

[Please submit the code for the frontend in your ZIP file]

Task 2: Back end (database)

Questions

1. Describe your schema. Explain your schema design decisions. Would your design be different if you were not using this database? How many iterations did your schema design require? Also mention any other design ideas you had, and why you chose this one? Answers backed by evidence (actual test results and bar charts) will be valued highly.

   Let's have an overview of the schema design of all

Query	Row-key	Columns
2	<User-id>;<Time>	One column: List of <tweet-id>:<score>:<censored content> (delimited by '\u0002' and all '\n' is replaced with '\u0002' in the tweet content)
3	<User-id>	One column: List of <user-id>(if mutual, (<user-id>))'\u0002'
4	<date>;<location><rank(with leading zeros)>	One column: List of <tweet-id> (delimited by ',' and end with '\n')
5	<User-id>	Three columns: <Score 1> | <Score 2> | <Score 3>
6	<User-id>	One column: <accumulated count>

Hbase has a relatively high performance on 'get' over 'scan', and if we use filtered 'get' or 'scan' it will be slower. This requires us to design the table so that we could use 'get' only once and get the result.

For example, for Q2, we have designed two different schemas. Except for the final schema, another schema is using the tweet-id as the row-key, and using user id, time, score and censored content as the columns. In this schema, we have to use a filtered scan in the columns to perform a single query. Our experiments shows the difference between these two schemas:

schema	Avg query time(s)
One-fetch 'get'	0.002
Filtered 'scan'	8

For Q2, we did the score calculations and tweet censoring in the ETL process and put the final result as a single column into Hbase. As the '\n' will make the data loading process difficult, we replaced all '\n' with '\u0002' and in the front-end we swap back.

For Q3, we also could do the similar pre-calculation in the ETL and directly load the data into Hbase.

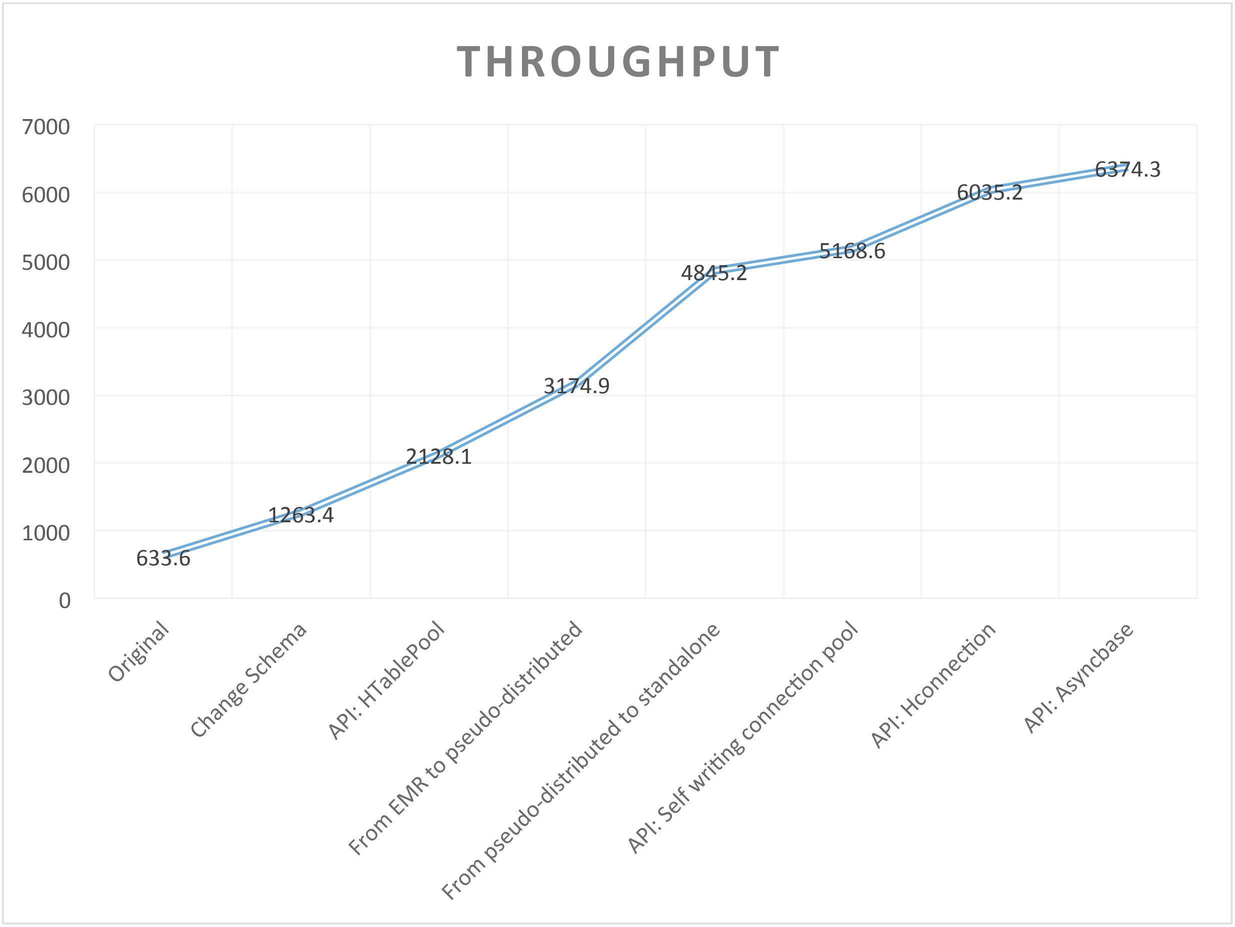
For Q4, it is different. The query asks for a range, which makes it almost impossible to design a one-fetch schema. We add the rank into the row-key, with leading zeros, and from this schema we could perform the query with a ranged scan.

For Q5, it is straight forward to have 3 columns and eliminate the total score which could be added in the front-end and thus save space for Hbase. For a single query, we can do a one-way multiple 'get' using the API instead of do 'get' twice.

For Q6, we first get the separate count for each user id, and then sort the user ids, and then for each user id, we store an accumulated count (a sum from the first user id to this user id) in Hbase. In this way, we can perform the query by doing a one-way twice 'get' using the API, and use the difference of two results as the sum between them.

This one-fetch design is not suitable for Mysql, in which a filtered ‘select’ isn’t that time consuming. Also, it’s convenient to store a variable-length text in Hbase while it will be space-consuming in Mysql.

2. What was the most expensive operation / biggest problem with your DB that you had to resolve for each query? Why does this problem exist in this DB? How did you resolve it? Plot a chart showing the improvements with time.
The network traffic brought by concurrent reading request on the Hbase.
When we send a request to Hbase, we firstly need to establish a connection (which is time consuming), and then send the request to Hbase and get response. In order to achieve high performance, we would like to keep this connection as long as possible and make each request lightweight and no longer to establish connections.
We resolve this problem by using several methods: change the API, change the backend system design and tuning the parameters of Hbase.
The improvements with the time could be described in the following chart (take Q3 as an example):



3. Explain (briefly) **the theory** behind (at least) 10 performance optimization techniques for databases. How are each of these implemented in MySQL? How are each of these implemented

in HBase? Which optimizations only exist in one type of DB? How can you simulate that optimization in the other (or if you cannot, why not)? Use your own words (paraphrase).

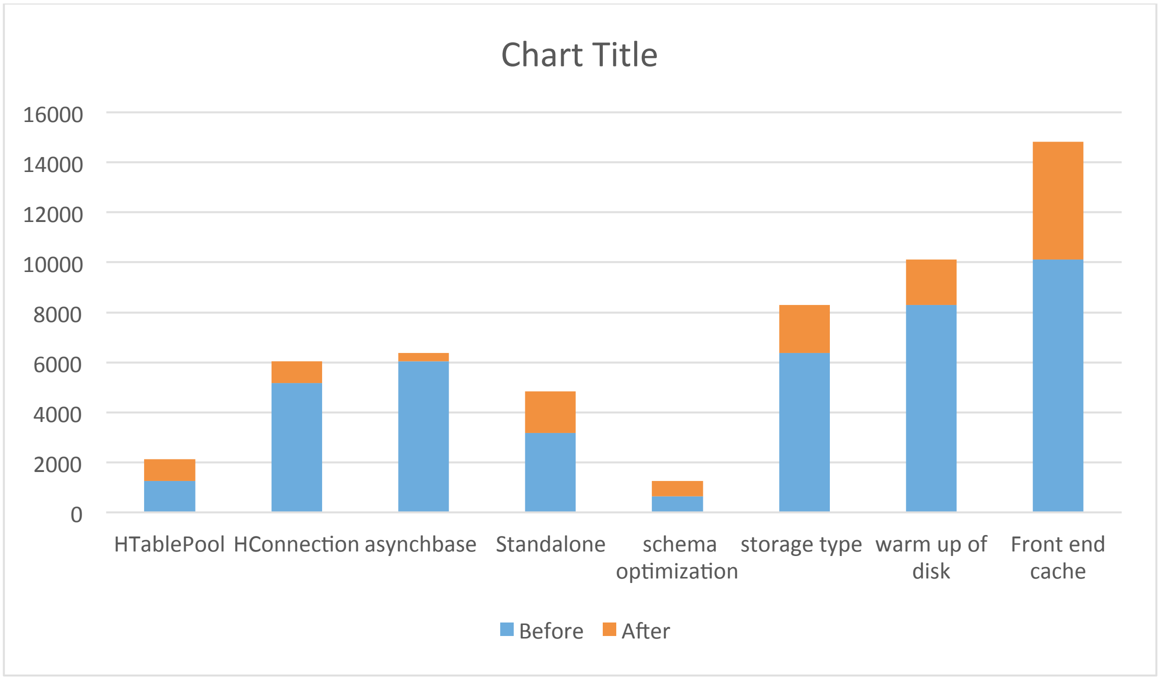
techniques	theory
Use API HTablePool instead of creating connections every time	Creating connections is time consuming. HTablePool is the first API we found that can pool HTable and thus reduce the cost to recreate connections.
Use API HConnection instead of HTablePool	Threadpool is good, but not good enough. It's deprecated since 0.98 of HbaseClient API. HConnection is the right option which creates lightweight HConnection and reuses it to get HTable. It can handle multi-table situations also.
Use API asynchbase instead of HConnection	Asynchbase is not the official API provided by HbaseClient. However, it provides the characteristics of non-blocking IO which is perfect for this project.("A fully asynchronous, non-blocking, thread-safe, high-performance HBase client.") The only problem is that we haven't finish testing on this API and for the live test, we still use HConnection.
From EMR to Standalone	EMR is easy to use, however, it costs more and spend much more time than a standalone hbase sever as it has to distribute the job to each datanode and then merge the results from each datanode. For the scale of the dataset in this project(Q2:30G, Q3:3G, Q4:2G, Q5:2G, Q6:1G), standalone is better option.
Tuning Hbase parameters	It's necessary to change the default parameter settings to better meet the requirements of the test. There are several important settings, including handlers, heap space and cache.
Optimize the schema design	As to our discussion in question 1, the quality of schema design has great influence over the performance. A good Hbase schema design requires one-fetch and should save the space as much as possible.
Load balancing	Even though the load balancing process will increase the latency over the transmitting, it has advantages to solve the 'high latency in concurrency' problem. When multiple threads try to do GET request to the Hbase cluster, the latency will increase dramatically, from 3ms to 8000ms in a short time. We don't know the reason behind this problem, but load balancing will solve this problem. However, we don't have enough time to do tests on this solution, so we don't finish Q2(which we can provide single-thread correct solution but the server will crash in a multi-thread environment)
storage type	Different types of storage have different performance and according to our experiments, provisioned SSD has the best performance (within limit of budget)
Warm up of disk	We've learned this in weekly project and it works! For a new created instance from our pre-created ami, warm-up can improve the performance at a considerable rate.
Front end cache	For some queries, the final dataset is not that big and we can store part of it in the frontend and use it together with requests to Hbase

Index frequent string	The locations which are frequent in q4 can be indexed with numbers, which results in a much smaller storage in both the front end cache and the Hbase.

4. Plot a graph showing results with/without each individual optimization that you used. Extremely impressive will be a timeline of rps v/s submission id (mentioning which optimization was in use at that time).
   Parameter tuning has no explicit experiment results as we always evolved them together with other techniques.
   Load balancing hasn't finished before the deadline of live test. We've tried several approaches over Q2 but all failed, and the failed statistics is as following table:

Solutions	Qps
No load balancing	900
Load balancing front end with ELB	600
Load balancing back end with ELB	100

   We believe that the right solution is using Nginx, but what a pity we don't have enough time to try this!
   Frequent terms indexing has influence on the memory space but not directly on the performance.
   A graph of the improvement(given Q3 as example) is as the following chart.



5. Would your design work if your web service also implemented insert/update (PUT) requests? Why or why not?
   Partially.
   For a heavy writing load environment, standalone is not appropriate option. A Hbase cluster enable concurrent write while in a standalone Hbase server will probably have problem when writes go to the same block.
   Also, parameter tuning will be different. Get and Put have different optimal settings and if

the requests are a combination of Get and Put, we have to balance performance in these two types of requests.

The front end-cache will also be more difficult. When a new write comes, should we only update the cache or send the write requests to the hbase? Also, updating the cache and the hbase will cost more effort than updating hbase only, and increasing memory need will also be a concern.

Other optimizations are available to Put as they try to speed up the process between the client and the Hbase server and their efforts are not relevant with the type of requests.

6. Which API/driver did you use to connect to the backend? Why? What were the other alternatives that you tried?

   Hconnection is our final choice.

API	characteristics
Without connection pooling	Creating connection for every request is time consuming!
HTablePool	An early version of HTable pooling but deprecated in later versions of HbaseClient API
HConnection	Official API for HTable pooling, replacement for HTablePool
asynchbase	A fully asynchronous, non-blocking, thread-safe, high-performance HBase client. However, we don't have enough time to test and implement it for all queries.

7. How did you profile the backend? If not, why not? Given a typical request-response for each query (q1-q6) what percentage of the overall latency is due to:
   a. Load Generator to Load Balancer (if any, else merge with b.)
   b. Load Balancer to Web Service
   c. Parsing request
   d. Web Service to DB
   e. At DB (execution)
   f. DB to Web Service
   g. Parsing DB response
   h. Web Service to LB
   i. LB to LG

   How did you measure this? A 9x6 table is one possible representation.

   I used the logs from Hbase (including all the nodes and JVM GC usage too) and from experimental front-end to record how the DB performs.

   A typical request-response for each query (q1-q6) is as the following the table (we don't have a load balancer, and the unit is ms):

Query	a	b	c	d	e	f	g	h	i
Q1	1		2					1	
Q2	1		2	1	7	1	1	1	
Q3	1		1	1	3	1	1	1	
Q4	1		1	1	8	1	2	1	
Q5	1		1	1	4	1	3	1	
Q6	1		1	1	7	1	1	1	

8. Say you are at any big tech company (Google/Facebook/Twitter/Amazon etc.). List one concrete example of an application/query where they should be using NoSQL versus one where they should be using an RDBMS. Both examples should be based on the same company (you choose).
Twitter:

Application	Back-end choice
User basic information storage	RDBMS
Image storage(or other non-structured data)	NoSQL

9. What was the cost to develop your back end system?
About $20.
10. What were the best resources (online or otherwise) that you found. Answer for both HBase and MySQL.
http://hbase.apache.org/book/quickstart.html : The official tutorial for quick start on build a Hbase cluster.
http://hadoopstack.com/improving-hbase-read-performance/ : a report of improving Hbase reading performance by a Facebook intern
https://hbase.apache.org/apidocs/ : API of Hbase client
https://github.com/OpenTSDB/asynchbase : code repository of asynchbase
http://www.ericsson.com/research-blog/data-knowledge/hbase-performance-tuners/ : parameter tuning for Hbase
http://www.fromdual.com/mysql-configuration-file-sample : A sample cnf configuration

[Please submit the code for the backend in your ZIP file]

Task 3: ETL

1. For each query, write about:

Q2:

a.

Attempt 1: Amazon EMR Hive + Sqoop

Ran Hive scripts on the master node of the EMR cluster, parsed the JSON format twitter data using the jsonserde.jar library, collected data from relevant fields: tweet id, tweet text, tweet time & user id, and stored the resulting table as a text file on the cluster's HDFS.

To reduce the computation workload at front-end hence to improve the throughput, tweet text censoring, sentiment scoring and tweet time format conversion were done at this stage as well, by writing Java class extending the Hive UDF (org.apache.hadoop.hive.ql.exec.UDF).

Loading was achieved by Sqoop, a popular big data analytics tool to copy data from Hadoop distributed file system to MySQL database via the JDBC driver.

However, this programming model worked slowly and performance did not change after modifying the code using the regex library for string processing. Quantitatively speaking, test run on 1.6G could not be finished within one night after which we had to terminate the workflow.

Attempt 2: Amazon EMR Streaming Program with Python (final version used)

JSON parsing was performed by Python JSON module and json.loads method.

Tweet time format conversion and tweet text initial filtering (remove \r or \n) was done at the map stage. At the reduce stage, banned words and sentiment words were stored as hash tables (dict and set in Python). Each words in the tweet text was searched in the two hash tables to see if any actions should be taken.

b.

m3.2xlarge: current generation general purpose instance type with the maximum computational capability, could achieve the lowest running time.

c.

19 m3.2xlarge: limited by the maximum number of spot requests

d & e & f.

19 m3.2xlarge instances running for 2 hours with spot price bid as $0.08

spot cost: $3.04

given the EMR cost of m3.2xlarge is $0.14/hr

total cost = $3.04 + $5.32 = $8.36

g.

Before the final successful run, 3 test runs on 1.6G data and 1 complete but erroneous output run (Unicode handling issue).

h.

1. Tweet text may contain \n or \r which would generate unexpected mapper output thus interfere the normal reducer job. However, simply replace every \n and \r with other character was not sufficient. Additionally, the \n\r or \r\n combination of characters should be interpreted as one newline character but not two. Before noticing this pattern, ETL run resulted in erroneous outputs. The character chosen to replace \n and \r in our design was \002 (ctrl+b) which was not highly unlikely to be present in tweet text hence would not interfere the original tweet text when removing and restoring newlines during map reduce job.

2. Nasty Unicode behavior should be carefully handled. With the help of the Python Unicode('utf-8') module, raw data could be successfully loaded and printed. However, it did not work completely compatible with the Python re module Unicode flag. During testing, we discovered that setting Unicode flag in Python regular expression turned out to be unnecessary.
3. Efficient string processing algorithm proved to be really helpful in reducing processing time. The first algorithm used was to search every banned word (or sentiment word) throughout the whole tweet text which was a complete waste of time (when later we noticed this problem). The tweet text was relatively short and the banned word list and sentiment words list were quite long. Additionally, string pattern search took long time. As a result, modifications were made such that banned words and sentiment words were stored as hash tables (dict and set in Python). Hash tables were used because it had constant time for searching. Also, the search was performed by checking the hash table per word in the tweet text but not the other way around as previously tried. The text was split into words every time it encountered an non-alphanumeric character or word boundary. After this modification, the running performance was considerably increased.
i.
The resulting data: 29G
After loading it to MySQL and creating index, the disk utilization: 40G.
j & k.
MySQL database was backed up by creating the AMI of the hosting instance which took around 20 minutes. The back up AMI had an EBS volume that dedicated around 40G to this MySQL database.

Q3:
a.
Since no complex inter-field processing involved, the Amazon EMR Hive + Sqoop programming model was used for Q3. 2 tables were created: retweet_from (current user id and the original user id) and retweet_by (original user id and current user id)
Later, when developing the backend, the 2 tables were merged locally and loaded into database.
b.
m3.xlarge: current generation general purpose instance type with the relatively high computational capability, lower cost than m3.2xlarge.
c.
19 m3.xlarge: limited by the maximum number of spot requests
d & e & f.
19 m3.xlarge instances running for 5 hours with spot price bid as $0.05
(5 hours including outputting results to S3 as well as to MySQL, actual transforming time within 2 hours)
spot cost: $4.75
given the EMR cost of m3.xlarge is $0.07/hr
total cost = $4.75 + $6.65= $11.4
g.
No incomplete ETL runs before the final run.
h.

Sending queries to 2 separate tables and combined the results back and fed to customer reduced the front end performance. Thereafter, merge on the 2 tables was performed locally, followed by reloading.
i.
the final merged table: around 1G (only storing numbers and relatively small record number contributed to this small database size)
j & k.
MySQL database was backed up by creating the AMI of the hosting instance which took around 20 minutes. The back up AMI had an EBS volume that dedicated around 1G to this MySQL database.

Q4:
a.
Since Q4 involves quite a bit computation between fields (calculating the rank and concatenating involved tweet ids), the EMR streaming program with Python programing model was used.
At the map phase, the Python json module loaded data, filtered tweets for those with hash tags, output tweet text, location as the key and list of (hash tag text, tweet id and hash tag index) as value.
At the reduce phase, 2 lists data structure were used to hold tweets collection for each specific hash tag at specified time/location and to hold hash tag collection for each time/location combination. Each list element was properly aligned such that when performing sort, desired ordering as per specified by the writeup would be retrieved.
b.
m3.2xlarge: current generation general purpose instance type with the maximum computational capability, could achieve the lowest running time.
c.
19 m3.2xlarge: limited by the maximum number of spot requests
d & e & f.
19 m3.2xlarge instances running for 2 hours 13 min (3 hours) with spot price bid as $0.08
spot cost: $4.56
given the EMR cost of m3.2xlarge is $0.14/hr
total cost = $12.54
g.
1 ETL run with wrong output: 19 m3.xlarge ran for 4.5 hours
h.
The complex ordering of the hash tags required quite much effort. After rounds of testing, we decided to utilize the Python list’s embedded sort method by aligning tweets count, tweet id and hash tag index in order in each list element.
i.
resulting: around 1G (only storing numbers and relatively small record number contributed to this small database size)
j & k.

MySQL database was backed up by creating the AMI of the hosting instance which took around 20 minutes. The back up AMI had an EBS volume that dedicated around 1G to this MySQL database.

Q5:
a.
EMR python streaming model.
Information on tweet id, user id and retweet status (user id) were first collected, then printed user id and tweet id key value pair for every tweet. For a retweet, also printed the original user id, tweet id, retweet user id key pair.
At reduce phase: since output from mapper was already sorted, reducer first distinguished if the current processed tweet was a retweet or not. For each user, 3 hash tables (set in Python) were created whose sizes were directly related to s1, s2 and s3. Hash table was used because it did not allow duplicates.
b.
m3.xlarge: current generation general purpose instance type with the relatively high computational capability, lower cost than m3.2xlarge.
c.
19 m3.xlarge: limited by the maximum number of spot requests
d & e & f.
19 m3.xlarge instances running for 4 hours with spot price bid as $0.05
spot cost: $3.8
given the EMR cost of m3.xlarge is $0.07/hr
total cost = $3.8 + $5.32= $9.12
g.
1 ETL run with erroneous output. 19 m3.2xlarge ran for 2hour 17min
h.
Firstly, only user id, original user id and tweet id was outputted (original user id set to -1 if not a retweet) from mapper, with the attempt to store the original user id in one big hash table throughout all mapper output. Wrong output was resulted because reducer jobs ran independently. To solve this, code was modified such that retweet also outputted original user id as mapper output key (which would be sorted and same user would be processed by one reducer)
i.
the final merged table: around 1G (only storing numbers and relatively small record number contributed to this small database size)
j & k.
MySQL database was backed up by creating the AMI of the hosting instance which took around 20 minutes. The back up AMI had an EBS volume that dedicated around 1G to this MySQL database.

Q6:
a.
EMR Python streaming programming model.

b.
m3.xlarge: current generation general purpose instance type with the relatively high computational capability, lower cost than m3.2xlarge.
c.
19 m3.xlarge: limited by the maximum number of spot requests
d & e & f.
19 m3.xlarge instances running for 4 hours with spot price bid as $0.05
spot cost: $3.8
given the EMR cost of m3.xlarge is $0.07/hr
total cost = $3.8 + $5.32= $9.12
g.
1 ETL run with erroneous output. 19 m3.2xlarge ran for 2hour 40min
h.
Duplicate tweets were not attended at first which resulted in wrong outputs.
i.
the final merged table: around 300M (only storing numbers and relatively small record number contributed to this small database size)
j & k.
MySQL database was backed up by creating the AMI of the hosting instance which took around 20 minutes. The back up AMI had an EBS volume that dedicated around 300M to this MySQL database.

2.Critique your ETL techniques. Based on your experiences over the past 6 weeks, how would you do it differently if you had to do the same project again.
After discussing with other teams after the project due, we discovered that EMR streaming program written in Python ran considerably slower than Java.
If I were to do the same project again, I might chose to write the streaming program in Java.

3. What are the most effective ways to speed up ETL?
In our case, the most effective way was to write efficient algorithm and to use proper data structure, such as has table for search.
4. Did you use EMR? Streaming or non-streaming? Which approach would be faster and why?
Yes, we used EMR streaming with Python and EMR Hive non-streaming. Since the code written at the 2 trial were different, we did not get the time to compare the running time of the two model under same algorithm. I guess writing proper (efficient algorithm with appropriate data structure) Hive UDF in Java would achieve faster processing speed.
5. Did you use an external tool to load the data? Which one? Why?
We tried the Sqoop tool to load data in HDFS to MySQL via JDBC driver. Because at first we were using the Hive programming model and resulting table was originally stored in HDFS as text file, while Sqoop performed well on copying data from HDFS to MySQL.
Which database was easier to load (MySQL or HBase)? Why?
It depends. If on standalone database, the loading time did not differ much. However, if on database cluster, most of the time was dedicated to setup the cluster environment.
[Please submit the code for the ETL job in your ZIP file]

General Questions

1. What are the advantages and disadvantages of MySQL for each of the queries you've encountered so far? Which queries are better suited for MySQL (not HBase)?

Query	Advantages	Disadvantages	Better Suitable?
Q2		The data is too big and it's hard to load the data into mysql server and create index on it.	
Q3	The field of userid is of fixed length and it's good to store the contents with fixed lengths.	The front end needs to parse the results from mysql server. Also a sort need to perform to ensure the right order which may decrease the performance	
Q4	The filter on the rank is faster in mysql than hbase. And mysql has value types of int and no leading zeros are needed in storage.		yes
Q5	Fixed lengths for all fields, search according to the indexed column only and no extra filters needed		
Q6	Could get previous row's value fast and thus can save a column which is needed for previous line(in order to compute the result)		

2. What are the advantages and disadvantages of HBase for each of the queries you've encountered so far? Which queries are better suited for HBase (not Mysql)?

Query	Advantages	Disadvantages	Suitable?
Q2	Key-value is good to store the variable-length value. Also the data's scale can be handled by Hbase well.	Concurrency traffic is terrible	Yes
Q3	By a reasonable schema design, the data could be stored as key-value for one-fetch request, which is fast and front-end don't need to parse the response again.		
Q4		There is no known schema design for one-fetch request and the filtered scan is time consuming.	
Q5		Multiple gets are slower than single get.	

Q6		Two scans are needed to be performed to perform a request which is slow.	

3. For your backend design, what did you change from Phase 1 and Phase 2 and why? If nothing, why not?
   From EMR to standalone. EMR wastes time in distributing mapreduce job and merge the results from datanodes. According to the scale of this project, standalone Hbase server is a better solution.
4. Would your design work as well if the quantity of data would double? What if it was 10 times larger? Why or why not?
   No. Our design doesn't have good scalability as we use standalone server and front-end cache. A scalable design should make full use of the Hbase cluster and load balancing.
5. Did you attempt to generate load on your own? If yes, how? And why?
   Yes. We tried to use Jmeter to generate loads. We have a weird problem on Q2 when multiple threads try to connect with our front end and the latency between the front end and back end grows dramatically. We want to observe the pressure limit our system can handle and find a way to balance the load for further load balancing.

More Questions (unscored)

6. Describe an alternative design to your system that you wish you had time to try.
   Nginx for load balancing. We've described it for so many times in this report and we wish we could have more time to try it and test whether it can save our Q2 ☹
7. What were the five coolest things you learned in this project?
   Team communication and leadership.
   The concept of budget.
   The need of progress document.
   How Hbase and Hadoop work.
8. Which was/were the toughest roadblock(s) faced? What was the solution to that problem?
   We have a weird problem on Q2 when multiple threads try to connect with our front end and the latency between the front end and back end grows dramatically. We haven't solved it before the deadline and we believe load balancing by Nginx or other alternatives other than ELB could be the right answer.
9. Design one interesting query for next semester's students.
   Shortest length from one user to another. We could define a path as a directed retweet chain.
10. Did you do something unique (any cool optimization/trick/hack) that you would like to share with the class?
    Front end cache. In order to achieve a low latency, cache on front end is necessary. We could use indexing and mapping to optimize the data structure so that we could cache more in the memory.
11. How will you describe this project (in one paragraph) on your LinkedIn / CV ?