S15 15619 Project Phase 3 Report

Performance Data and Configurations

Best Configuration and	Results from the Live Test
Choice of backend (pick one)	MySQL
Number and type of instances	Live Test: 9 M3.large instances with 1 ELB
Cost per hour (assume on-demand prices)	Live Test:(0.125*200/30/24 + 0.140)*9 + 0.025 =1.5975
Queries Per Second (QPS)	INSERT HERE: (Q1,Q2,Q3,Q4,Q5, Q6)
Rank on the scoreboard:	Phase 1: 6 Phase 2:6 Phase 3:14

	Q1	Q2	Q3	Q4	Q5	Q6
score	86.19	162.81	109.27	151.96	49.01	119.29
tput	12928.7	9768.6	10927.3	9117.8	1980.1	11929.4
ltcy	7	5	4	4	25	4
corr	100.00	100.00	100.00	100.00	99.00	100.00
err	0.00	0.00	0.00	0.00	0.00	0.00

Team : Oak
Members :Ziyuan Song, Aaron Hsu, Jiali Chen

Rubric:

Each unanswered question = -10%
Each unsatisfactory answer = -5%

[Please provide an insightful, data-driven, colorful, chart/table-filled, and interesting final report. This is worth 20% of the grade for Phase 3. Use the report as 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 (you may/should copy answers from your earlier report-- each report should form a comprehensive account of your experiences building a cloud system. Please try to add more depth and cite references for your earlier answers)

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].

We used Undertow after investigation to see which front end frameworks has the most efficiency in terms of RPS. This is because on EC2 instances, it has one of the highest performance with both plain text and query throughputs. These can be found on https://www.techempower.com
Undertow is lightweight, embedded, flexible, as argued on numerous sources on the web. It has both synchronous and asynchronous (based on NIO) APIs. It allows multiple combinations of various handlers which can be easily added when needed, and removed when trying to get rid of extra overheads. (EDIT) In addition, it allowed pipelining, using the dispatch function from undertow, we were able to thread multiple connections to both mysql and hbase, thus allowing us to use a single front end for a backend server.
For phase 3, we merged the front end with our backend as one instance, with a total of 9 instances.

2. **Did you change your framework after Phase 1? Why or why not?**

No we did not. We felt that the undertow framework is fast and reliable enough. Since we were able to reach 17000 RPS with only one front end for Q1, our major concerns were not about the front end framework. Our primary bottleneck resides at the backend, which we spent most of our time tuning.

3. **Explain your choice of instance type and numbers for your front end system.**

For Mysql, we used 4 instances of M1.Large instance type. We chose m1 instances over m3 because m1 instances were rumored to be more effective at PV. In addition, we chose large instances over medium and small instances to avoid overheads caused by ELB round robin switching. Also, we chose the minimum number of m1.large instances required to pass

15000 RPS because we wanted to save money **FOR YOU GUYS**.We were under

the limit of 1.25 dollars for Q1, using almost only 8% of the limit.
(EDIT) However after further investigation in Q2, we found out that m3.large is more efficient than m1.large because it has faster processing power, and is cheaper in terms of on-demand price. In Q2, for mysql, we had 3 m3.large instances, and 1 m3.large instance for hbase.

(EDIT) During phase 2, we decided to go with 4 m1.large instances for mysql and 1 m1.large instance for hbase. This was due to our infrastructure of 4 backend servers for mysql and a master node for hbase. However, we realized that ELB does not perform well for mysql Q1. So with hindsight we, should have just provisioned a backend with 20k IOPS and a single frontend without an ELB. Then we would be able to reach the desired score for Q1 and at the same time reaching the scores we got for other Q's.
(EDIT) During phase 3, we decided to go with 9 M3.large instances because it's cheaper in terms of on-demand price and so we are able to add more instances than M1, even though M1 is in fact faster than M3. (We benchmarked it with Q1).

4. **Explain any special configurations of your front end system.**

We used an Amazon paravirtualized machine (since PV should be better than HVM in terms of efficiency),configured the instance to support java, undertow and produced an AMI based on that configuration. This allows us to easily create instances running the front end web-servers when horizontal autoscaling is needed.

We configured our front end to support ordered pipelining request to our backend. (This was done by undertow dispatch method) This allows our one single front end to have multiple connections to a back end, so instead of having one backend and multiple front ends (with back end being the bottleneck), we can have n number of replicated backends paired with n number of front ends (In mysql).
During phase 3, although we used Myisam as our mysql engine, and found benchmarks that show how multiple threads will actually cause myisam to be slower, in our actual test, we did find dispatch to have a higher throughput.

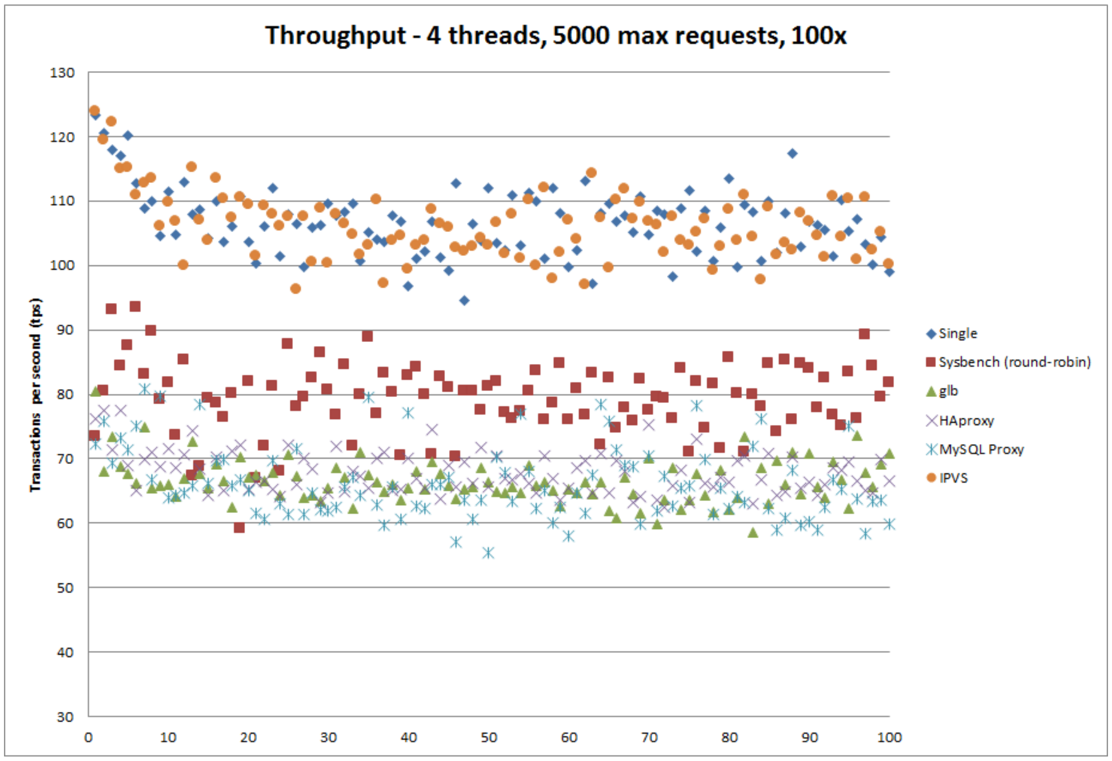
5. **Did you use an ELB for the front-end? Why, or why not? Condense your experience with ELB in the next few sentences. Talk about load-balancing in general and why it matters in the cloud.**

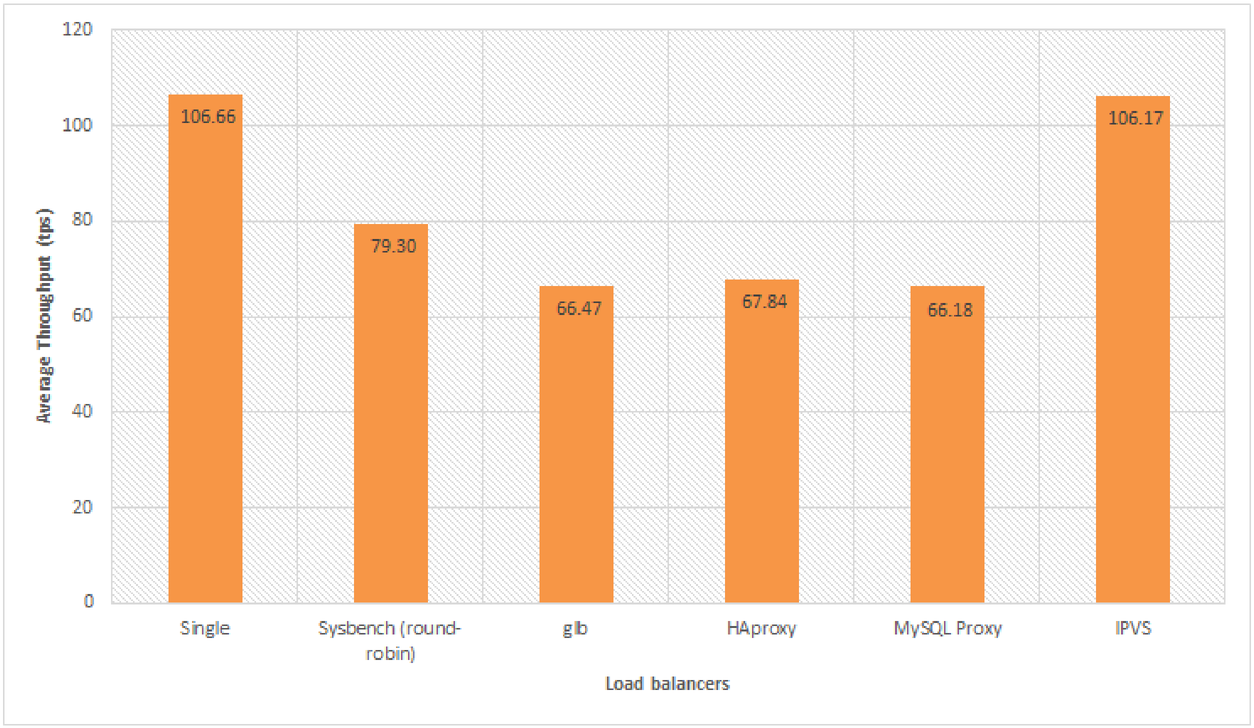
Yes, we used an ELB for the front-end. Because ELB can automatically route traffic across multiple instances to achieve higher levels of fault tolerance, it also automatically scales its request handling capacity to meet the demands of application traffic, since we had multiple front ends for achieving high RPS. It reduces the bottleneck by spreading the traffic to multiple front ends, routing to multiple backends. However, after phase 2, we found out that ELB does not perform well with multiple front end queries for Q1. This was because each frontend does its own threading, but ELB distributes the queries to each threading frontend, so the Q1 score was lower than expected.

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.**

In phase 1, we tried using Haproxy instead of an aws ELB. Unlike the ELB, Haproxy is not a black box, since we were able to see the exact traffic and ports opened to each front end. In addition, it does not require warming up. Prior to warming up, Haproxy performs much better than ELB, however after warming up the ELB, it allows a much higher throughput than Haproxy. (For mysql, Haproxy achieves around 6000 with 2 backend, 2front end, while ELB achieves around 6500 after warming up.) So in phase2, we only use ELB which provides more stable performance.

The two graphs below show two normals(single and sysbench), with three layer 7 load balancers and one layer 4 switcher. Since we wanted to finish the project before investing more time for the complicated IPVS switcher, we decided to use ELB first before the deadline for phase 2. We will most likely test out IPVS in phase 3.



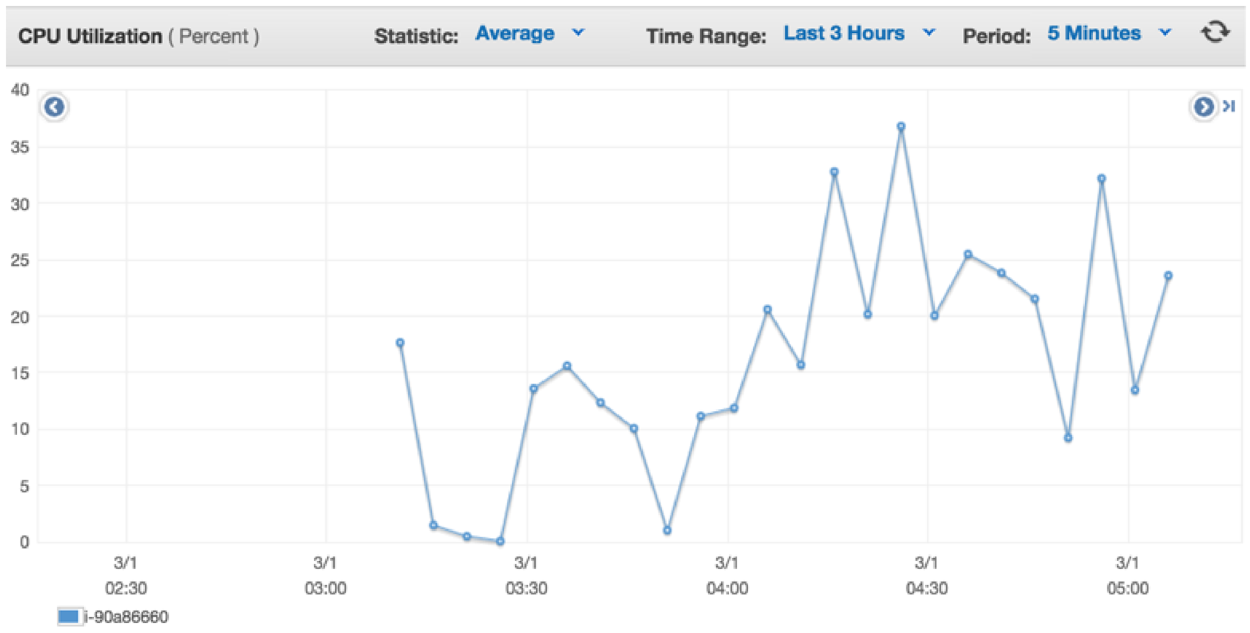


7. **Did you automate your front-end instance? If yes, how? If no, why not?**

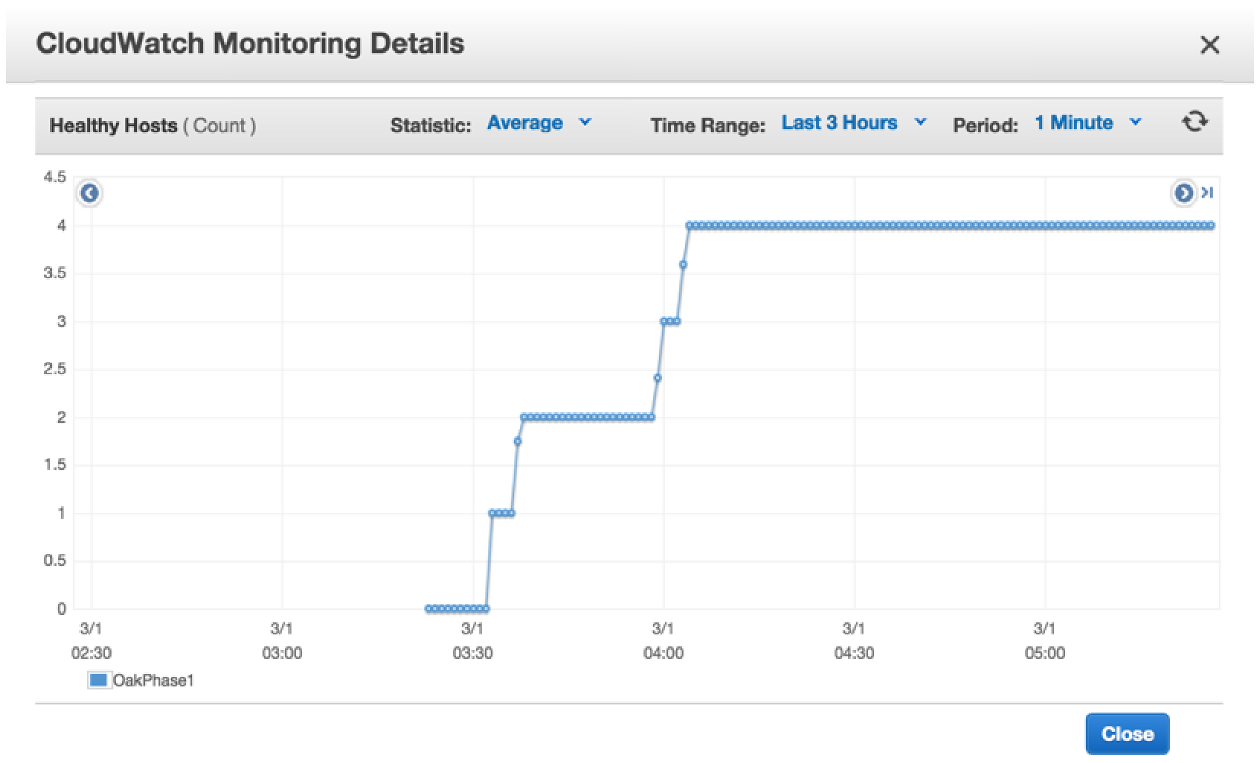
Yes, we wrote a shell script that updates, installs all required packages and added the script that initiates the front end server. Then we added the command that initiates the front end server to crontab, specifying that the server should be initiated when system startup. Then, we created an amazon machine image with it, so then we'll be able to start multiple instances with the server running automatically. (For Auto Scaling purposes that may be required in the future?).

8. **Did you use any form of monitoring on your front-end? Why or why not? If you did, show us a capture of your monitoring during the Live Test.** Else, try to provide CloudWatch logs of your Live Test in terms of memory, CPU, disk and network utilization. Demarcate each query clearly in the submitted image capture.

Yes, we used CloudWatch metrics: Basic monitoring. Because we want to monitor the cpu utilization of front-end instance to check whether we need more instances to meet the demands of traffic. In addition, we also used 'top' to monitor %wa (io wait time), and %id (remaining CPU), and 'free' for checking available RAM so we can decide how much for and whether if swapping is occurring.



We also check the health status of instances to guarantee that every instance is running and receiving url requests.



9. What was the cost to develop the front end system?

In phase 1 it was 17.54+2.85+0.748 = 21.138

In phase 2 and phase 3, we barely allocated any time for tuning our front end system.

10. **What are the best reference URLs (or books) that you found for your front-end? Provide at least 3.**

http://javaarm.com/faces/display.xhtml;jsessionid=j8CbEIX-4bB-B+6C0VcaxRK4?tid=3314&page=1&print=true

http://undertow.io/documentation/core/undertow-request-lifecycle.html

http://repository.jboss.org/nexus/content/unzip/unzip/io/undertow/undertow-core/1.0.14.Final/undertow-core-1.0.14.Final-javadoc.jar-unzip/io/undertow/server/HttpServerExchange.html?nsukey=4rb5LcQ%2Fm%2BVQcSWTkLfVV1wpxMYKHAyXN%2BxmK%2BHWoI066%2BPFeB2jYsoFpJh0IbWimFSSRyEVyLqGZNrYX7mysA%3D%3D

https://www.youtube.com/watch?v=ZZ5LpwO-An4

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

Task 2: Back end (database)
Questions

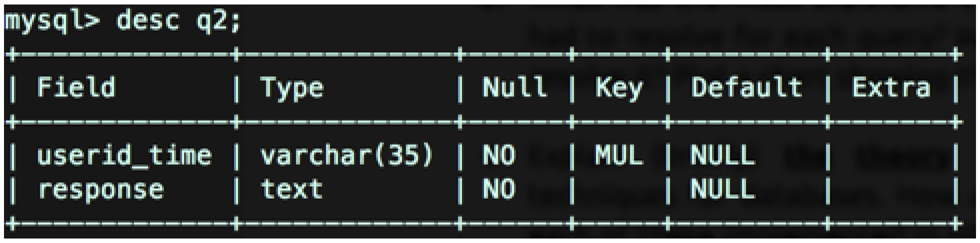
1. Which DB system did you choose in Phase 3? Why? Would any different queries for Q5 and Q6 have influenced you to choose the other DB?

We choose mysql. Mysql is simpler to configure and has higher throughput than hbase according to the result of phase2. If Q5 has a much larger time range, then a columnar store(hbase) would be better, however with myisam, it stores a lot of metadata such as the count of rows and the max of the data if there's an index for it. So it isn't much beneficiary to use hbase. As for Q6, it's just a simple subtraction of results from two rows, hence a columnar store isn't necessary. The only reason that might influence us to use hbase is if the data is much much much larger. Then scalability is a big issue.

For Q5, we need to do range query. Mysql has better performance for small range of rows than that of Hbase's scan operation.

2. 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) **<u>are required</u>**.

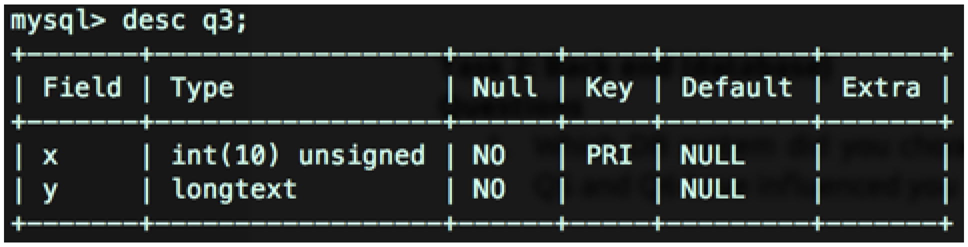
For q2:



```
mysql> desc q2;
+-------------+-------------+------+-----+---------+-------+
| Field       | Type        | Null | Key | Default | Extra |
+-------------+-------------+------+-----+---------+-------+
| userid_time | varchar(35) | NO   | MUL | NULL    |       |
| response    | text        | NO   |     | NULL    |       |
+-------------+-------------+------+-----+---------+-------+
```

Our first column (userid+time) was not unique. At first, we were hesitant about which engine to use, innoDB or Myisam. However, after closer evaluation, we chose to use Myisam because Myisam has several optimization options that specializes for reading operations. The first option was a fixed row optimization, this allows reading simply by pointing at the correct block, thus allowing really fast reads. The benchmark could be found here
http://www.soliantconsulting.com/blog/2012/09/mysql-optimization-faster-selects-myisam-fixed-row-format
However, this requires that we change the text(tweetid+calculated score+censored text) to be a fixed length (max of text length). This requires too much disk space, so we disregarded it. The option we chose was myisam compression + indexing. After we loaded the data into the MYD file, we indexed them with our row key. Then we myisampack-ed them compressing the MYD file and resorted the MYI index file. This enables the database to perform less IO disc reads to find the query data.
http://www.techrepublic.com/article/save-disk-space-by-compressing-mysql-tables/

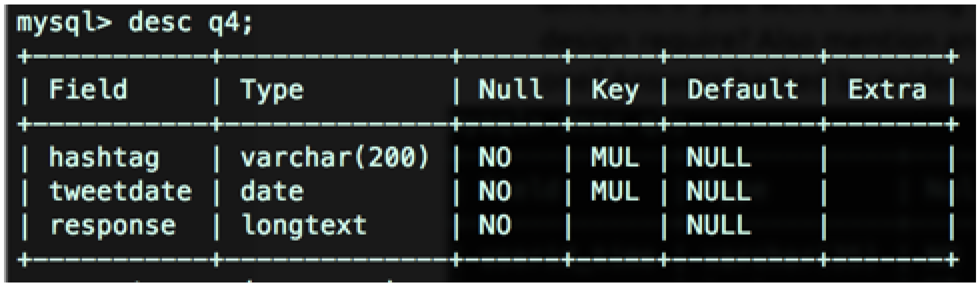
For Q3:



```
mysql> desc q3;
+-------+------------------+------+-----+---------+-------+
| Field | Type             | Null | Key | Default | Extra |
+-------+------------------+------+-----+---------+-------+
| x     | int(10) unsigned | NO   | PRI | NULL    |       |
| y     | longtext         | NO   |     | NULL    |       |
+-------+------------------+------+-----+---------+-------+
```

We have 2 columns for q3. The first one is unique userid, and second is its relations with other userid. We put all of them together in one column. Since the length of that relations is quite big, we use longtext as the type of that relations. Also, because this table has a text column, we couldn't use the fixed row design, thus instead, we compressed is using myisampack.

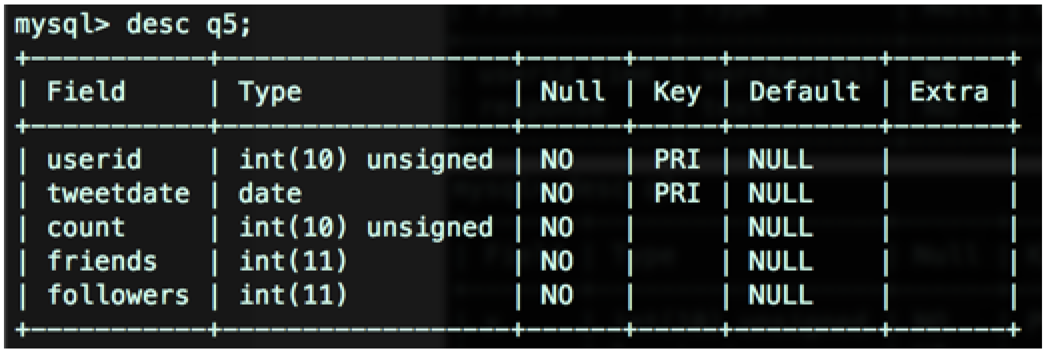
For q4:



```
mysql> desc q4;
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| hashtag   | varchar(200) | NO   | MUL | NULL    |       |
| tweetdate | date         | NO   | MUL | NULL    |       |
| response  | longtext     | NO   |     | NULL    |       |
+-----------+--------------+------+-----+---------+-------+
```

Three columns for q4. Query for data is: "SELECT response FROM q4 WHERE hashtag=? and tweetdate>=? and tweetdate<=? order by tweetdate". We use multi-column index on (hashtag, tweetdate) which has better performance than multiple indexes. The response column contains information of tweetid, userid and tweet time for the hashtag in that day. If one hashtag appears multi times in one day in different tweet id, the response information are all aggregated in one column corresponding to the hashtag and tweet date.

For Q5:

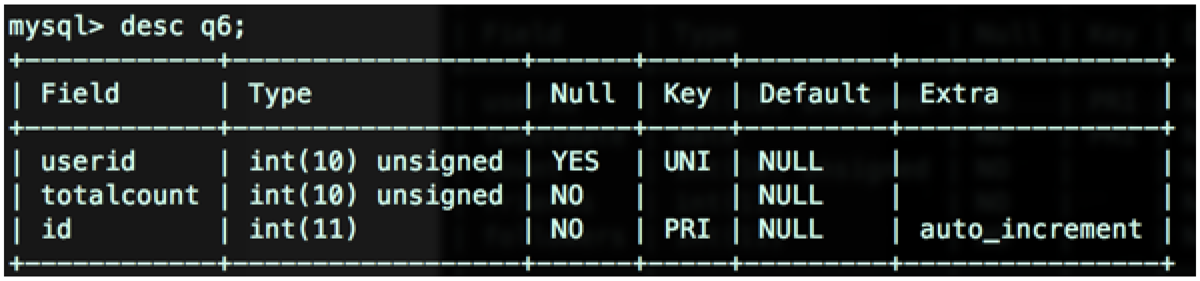


```
mysql> desc q5;
+-----------+------------------+------+-----+---------+-------+
| Field     | Type             | Null | Key | Default | Extra |
+-----------+------------------+------+-----+---------+-------+
| userid    | int(10) unsigned | NO   | PRI | NULL    |       |
| tweetdate | date             | NO   | PRI | NULL    |       |
| count     | int(10) unsigned | NO   |     | NULL    |       |
| friends   | int(11)          | NO   |     | NULL    |       |
| followers | int(11)          | NO   |     | NULL    |       |
+-----------+------------------+------+-----+---------+-------+
```

We use 5 columns for q5. Column count represents counts of tweet id for the userid and that tweet date. Counts, friends, and followers have been aggregated for one userid and one date in MapReduce. Query for data is: "SELECT userid, sum(count)+3*max(friends)+5*max(followers) FROM q5 WHERE userid in ("+userlistS+") and tweetdate between ? and ? group by userid". We use userid and tweetdate as the query condition, so we set them as multi-column primary key.

In the beginning, we calculate points for each userid in the front end, and query each userid per time. Later, we found that connecting to database and query one userid each time cost much more time that query for all user ids in one connection and one statement. So we choose user "select * from userid in()" statement for multiple userids.

For Q6:



```
mysql> desc q6;
+------------+------------------+------+-----+---------+----------------+
| Field      | Type             | Null | Key | Default | Extra          |
+------------+------------------+------+-----+---------+----------------+
| userid     | int(10) unsigned | YES  | UNI | NULL    |                |
| totalcount | int(10) unsigned | NO   |     | NULL    |                |
| id         | int(11)          | NO   | PRI | NULL    | auto_increment |
+------------+------------------+------+-----+---------+----------------+
```

We came up with a great idea to store q6 data. The totalcount column means total counts from 0 to current userid. In this case, we don't need to count userid in the query, hence we can lower the latency since count() operation is a range query, it scans every row in the table between those user ids. Our query for this table only read 2 rows, it is: "select ((select totalcount from q6 where q6.userid = (select max(userid) from q6 where userid<=?)) - (select totalcount from q6 where q6.userid = (select max(userid) from q6 where userid<=?)))".

3. What was the most expensive operation / biggest problem with each 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.

There wasn't any expensive operation for all the queries in myisam Mysql. If we had to give one, then it would be the large number of rows in query2. We solved it by using compression, thus looking up data with index would be easier. In addition to that, we could have partitioned the table into smaller ones, although it was unnecessary.

4. Explain (briefly) **<u>the</u> <u>theory</u>** behind (at least) 11 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, this document goes through plagiarism detection software). **(this question is worth 20 points if not answered in detail)**

a. Replication - This allows different front ends to be able to connect to their own database, reducing the bottleneck of the backend if it is one. So multiple read queries can be done simultaneously without accumulating CPU and IO-wait time.

b. Sharding - Splits the one database into several smaller databases. This allows read and write operations to be performed parallely at the smaller databases, reducing time spent waiting on read/write locks. This also reduces the number of look ups needed to find the data for a query.

c. Indexing - Stores pointers to data that can be found with faster look up speed. For instance the indexes can be store via hash table or Btree.

For MySQL, replication and sharding is done via a mysql cluster. An API node would be the communication node between front ends and all its data nodes, while a master node allows configuration of the mysql cluster. A user could specify the number of sharding or replication nodes through the master node. As for indexing, the engine we used (myisam) allows only Btree indexing.

For Hbase, replication and sharding was done by itself automatically. All we had control over was the size until region splits and to presplit the regions so that we can load values to different region servers, so the traffic does not all go to one region server. Hbase has its own indexing system, it uses the rowkey (userid+time) as the index to quickly find where in the hfiles the queried data exists.
(EDIT)

The other four optimization techniques were compression, multiple threading/connections, increasing memory (RAM) used, and decreasing IO wait time.

For mysql, since we used innodb and used almost all our time on hbase, we didn't feel the need to use compression for our data. As for multiple threading/connections, we sinply used threading at the front end to have multiplied connections to the backend. We also changed multiple variable values that's provided in my.cnf (in the hand in). The most important variable changed was key_buffer size, which allowed a faster search for indexes. Finally, for the decrease of IO wait time, we simply increased the used provision IOPS and thus there were no IOPS quota thus no wait time (%wa was around 0).

In addition, in phase 3, we used mount with option noatime, which tells the machine to not write the last accessed date (one less write operation for every read file).

We made sure we chose an PV image instead of HVI image since PV is better optimized for efficiency (IO and network operations).

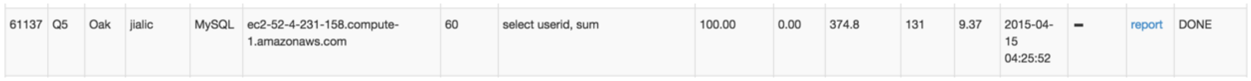
We also set the swappiness of the machines to be 0, and allowed the mysql database to control the memory by itself.

In relations to the my.cnf files, We set the default fsync() option to o_dsync, which was ~500 throughput faster for our specific image (amazon PV). We also disabled all logs, so there are less writes.

There are a bunch of variables that were tweaked but the most important was key buffer size.

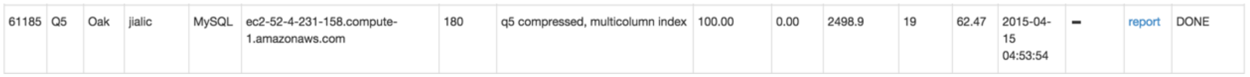
5. Plot a graph showing results with/without each individual optimization that you used. Extremely impressive will be a timeline of rps v/s submission id (mentioning which optimization was in use at that time).

A. Compression:
   a. before compression:



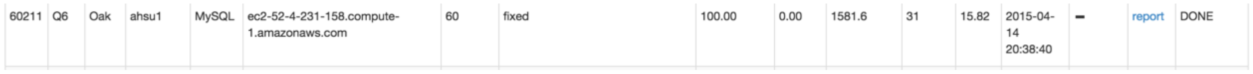
61137	Q5	Oak	jialic	MySQL	ec2-52-4-231-158.compute-1.amazonaws.com	60	select userid, sum	100.00	0.00	374.8	131	9.37	2015-04-15 04:25:52	–	report	DONE

   b. after compression:



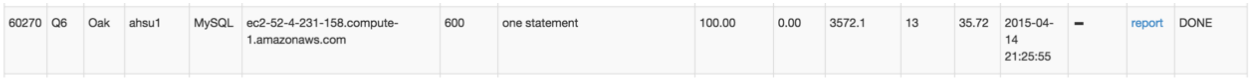
61185	Q5	Oak	jialic	MySQL	ec2-52-4-231-158.compute-1.amazonaws.com	180	q5 compressed, multicolumn index	100.00	0.00	2498.9	19	62.47	2015-04-15 04:53:54	–	report	DONE

B. One connection and one statement for each request:
   a. Two connections for q6 in one request:



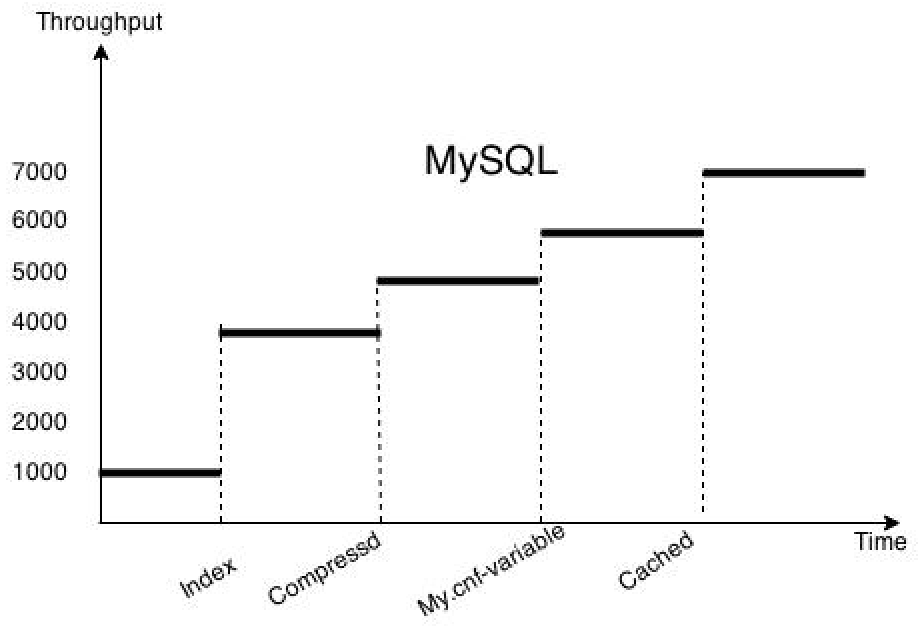
60211	Q6	Oak	ahsu1	MySQL	ec2-52-4-231-158.compute-1.amazonaws.com	60	fixed	100.00	0.00	1581.6	31	15.82	2015-04-14 20:38:40	–	report	DONE

   b. One connection and only one statement for q6 in one request:



60270	Q6	Oak	ahsu1	MySQL	ec2-52-4-231-158.compute-1.amazonaws.com	600	one statement	100.00	0.00	3572.1	13	35.72	2015-04-14 21:25:55	–	report	DONE

The following chart shows individual factors that impact on the final rps.



6. Would your design work if your web service also implemented insert/update (PUT) requests? Why or why not?

For Mysql, definitely not. This is because our database used myisam engine with compression. This automatically makes the table read only.

7. Which API/driver did you use to connect to the backend? Why? What were the other alternatives that you tried? What changed from Phase 1? From Phase 2?

We used the jdbc driver for mysql. We didn't try other alternatives, and found jdbc to be sufficiently fast. http://jtds.sourceforge.net/benchTest.html

Our bottleneck, latency was never caused by parsing. This was because when doing map, we already parsed and loaded the desired results into the db. Thus we did not put too much of our time to investigating which driver was the fastest, instead we invested most of our time in optimizing IO wait time.

8. Can you quantify the speed differential (in terms of rps or Mbps) between reading from disk versus reading from memory? Did you attempt to maximize your usage of RAM to store your tables? How much (in % terms) of your memory could you use to respond to queries?

Reading from memory is around 1 to 2 orders of magnitude faster than reading from disk. Reading from disk should be 300 to 20000 requests per second for SSD. On the other hand, if directly reading from memory, the rps should be at least 15000 and above. (restricted by bandwidth) For Mysql, we did not use RAM to store our tables at all. Instead we allocated most of our RAM for index searching (key buffer size). Since our memory was 8G (RAM), just to be safe from swapping, we used 5G for heap for hbase and 5G for key buffer size for mysql. ⅝ was around 62.5%.

9. Did you use separate tables for Q2-Q6? How can you consolidate your tables to reduce memory usage?

Yes, we use separate tables. We didn't consolidate our tables, however we had a even better idea of consolidating all the columns into a single column. Thus if we queried A, and we have tables that map A -> B, then we get B directly instead of getting more data from different columns and different rows. By creating an index on A, we can get B really quickly.

We also use "explain" sql query to check how index works and which kind of query are more efficient.

10. What are the flaws you have seen in both DBs?

In Hbase, there's a lot of overhead of the communication between different layers. So for a small to medium sized database (our case) it wouldn't be efficient. As for mysql, traditionally, it's not suitable for large magnitude of scaling, however with ndb mysql cluster, it's starting to have the ability to scale out more efficiently. Though it still doesn't support the rich and diverse parallel functions supported by the entire hadoop infrastructure.

11. How did you profile the backend? If not, why not? Given a typical request-response for each query (Q2-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 9x5 (Q2 to Q6) table is one possible representation.

    I blame aws for latency between AZ hence the 20% :)

	Q2	**Q3**	**Q4**	**Q5**	**Q6**
Load Generator to Load Balancer (if any)	<20%	<20%	<20%	<20%	<20%
Load Balancer to Web Service	<7%	<7%	<7%	<7%	<7%
Parsing request	<3%	<3%	<3%	<3%	<3%
Web Service to DB	<1%	<1%	<1%	<1%	<1%
At DB (execution)	<38%	<38%	<38%	<38%	<38%
DB to Web Service	<1%	<1%	<1%	<1%	<1%
Parsing DB response	<3%	<3%	<3%	<3%	<3%
Web Service to LB	<7%	<7%	<7%	<7%	<7%
LB to LG	<20%	<20%	<20%	<20%	<20%
	100%	100%	100%	100%	100%

12. What was the cost to develop your back end system?

The cost of developing was less in phase 3 since most of the configurations and structures were completed during phase 2. The only cost for phase 3 was ETL for Q5 and Q6, optimization for the SQL query statements for all Q's, and most of cost originates from the live test.

13. What were the best resources (online or otherwise) that you found. Answer for HBase, MySQL and any other relevant resources.

HBASE:
http://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/
http://www.slideshare.net/vanuganti/hbase-hadoop-hbaseoperationspractices
http://blog.cloudera.com/blog/2012/08/hbase-replication-operational-overview/
http://www.slideshare.net/cloudera/hbasecon-2013-compaction-improvements-in-apache-hbase
http://www-01.ibm.com/support/knowledgecenter/SSPT3X_3.0.0/com.ibm.swg.im.infosphere.biginsights.analyze.doc/doc/bigsql_compaction.html
http://www.ngdata.com/visualizing-hbase-flushes-and-compactions/
http://www.larsgeorge.com/2009/10/hbase-architecture-101-storage.html
http://abloz.com/hbase/book.html#mapreduce

(EDIT) For phase 2

http://cassandratraining.blogspot.com/2013/05/apache-hbase-default-configuration.html
http://www.oracle.com/technetwork/java/javase/tech/g1-intro-jsp-135488.html
http://www.oracle.com/technetwork/java/javase/tech/g1-intro-jsp-135488.html
http://hortonworks.com/blog/hbase-blockcache-101/
http://www.ericsson.com/research-blog/data-knowledge/hbase-performance-tuners/
http://ronaldbradford.com/blog/bigint-v-int-is-there-a-big-deal-2008-07-18/
http://ronaldbradford.com/blog/bigint-v-int-is-there-a-big-deal-2008-07-18/
http://planet.mysql.com/entry/?id=13825
http://search-hadoop.com/m/qOo2yyHtCC1
http://blog.cloudera.com/blog/2012/06/hbase-io-hfile-input-output/

google's bigtable paper : BigTable: A Distributed Storage System for Structured Data

(EDIT) For Phase 3
https://engineering.eventbrite.com/optimizing-a-table-with-composite-primary-keys/
http://hashmysql.org/wiki/Tuning_System_Variables
http://mysqlresources.com/documentation/other-statements/load-index-cache
http://www.computerhope.com/unix/myisamch.htm
http://www.slideshare.net/AmazonWebServices/maximizing-ec2-and-elastic-block-sto

re-disk-performance-32781088
https://dev.mysql.com/doc/refman/5.5/en/multiple-key-caches.html
http://stackoverflow.com/questions/8155805/how-can-i-determine-whether-using-multiple-key-caches-for-myisam-is-helping
http://flylib.com/books/en/3.30.1.67/1/
http://aimeos.org/docs/Administrators/Optimize_performance/MySQL

Task 3: ETL

1. For each query, write about:
    a. **The programming model used for the ETL job and justification**

For q2 to q6, we both used Elastic Mapreduce on Amazon as the programming model for the data extraction and transforming job. We used 1 master and 18 cores of m1.large to these jobs. We have 1.2 Tb compressed raw data from S3. It will take a very long time if we use only one instance to do the ETL job. EMR can effectively extract data from s3 bucket with compressed files as input stream. And it can apply mapreduce jobs on hadoop cluster, which can decrease the whole time cost to process data parallelly.

b. **The number and type of instances used and justification**

In phase1 and 2, for both q3 and q4, we used 1 master and 18 cores of m1.large to do the final extraction and transforming jobs. More instances can improve efficiency of data processing and large instance can also be more helpful than small and medium. m1.

For etl in phase 3, we used 1 master and 5 cores of m1.large to do the extraction and transforming jobs. It runs about 5 hours, much longer than our previous 1.5 hours in 19 instances. But it also cost less, it takes 120 instance hour, while using 19 instances would take 152 instance hour.

c. The spot cost for all instances used

For ETL in the whole phase 3, spot cost for all EC2 instance are: 4.5(jiali) + 2.29(Aaron) = 6.79

d. The execution time for the entire ETL process

	Extraction and Transformation(MapReduce)	Load(mysql)
Q5	5 hours, 13 minutes	10 min
Q6	8 hours	3min

e. The overall cost of the ETL process:

total :5.5(jiali)+3.29(Aaron) = 8.79

f. The number of incomplete ETL runs before your final run

1 for phase3

g. Discuss difficulties encountered

Phase2:

For q3, we got a too complex idea in extracting and counting relations between userid and it's reteet userid, after brainstorming in team, we got a better idea in dealing with the problem. We record + and - relations in mapper program, and using word counting method and hashmap data structure in reduce to count their relations and get * relations.

For q4, we got stucked in the idea of how to store data in database. For mysql, we decided to store the data in 3 columns. hashtag, tweetdate, tweetid+userid+tweetTime. For Hbase, we set rowkey as hashtag, set unique tweetdate as different column qualifier in the same family. Since we misperceived the maximum date distance, which is actually about 700+ days, we failed in ETL mapreduce 5 times. And finally we had to load data using put commands, which cost us about 24 hours, but fortunately it loaded successfully.

Another problem we met in both q3 and q4 is that we ignored deleting duplicated records, which result in some errors in the live test. We tried to solve it, but went in wrong direction again. We changed the key to userid_tweetid as the output of mapper, and delete the same key in reduce program. It would work well if the sort function in mapreduce is in alphabetical order, however it is not, it just simply put the same key in the same reduce. So next time, we will append tweetid at the end of value rather than key position to solve the duplicated problem.

Phase3:

For q5, we set followers and friends as unsigned integer at first. Later, we found that they might be negative number, so we changed them as normal integer.

h. The size of the resulting database and reasoning

q2: 20G
q3: 3.1G
q4: 4G
q5: 5G
q6: <1G
(all these numbers were after myisam compression)

i. The size of the backup

For mysql, we choose a 200GB EBS to backup all the queries.

2. **What are the most effective ways to speed up ETL? How did you optimize writing to your backend? Did you make any changes to your tables after writing the data? How long does each load take?**

The most effective way to speed up ETL was to perform ETL on previously already trimmed data. So it has less input to read in. Writing to the back end of our mysql db can be sped up by first loading the data locally and without primary key and indexing. Create the index after all the data has been loaded so mysql doesn't load and sort at the same time.

Loading	q2: <2 hours
	q3: <30 min
	q4: < 20 min
	q5: < 20 min
	q6: < 15 min

3. **Did you use EMR? Streaming or non-streaming? Which approach would be faster and why?**

We used EMR streaming with all the queries. This was because using yarn will require a lot more time on coding, and most of the data required for each query were relatively simple.

4. **Did you use an external tool to load the data? Which one? Why?**

We didn't use any external tool to load the data, since the data was relatively simple and not too enormous.

5. **Which database has been easier to load (MySQL or HBase)? Why? Has your answer changed in the last four weeks?**

There isn't much different in the difficulty of loading data into mysql or hbase. The only difference is that in hbase we are able to load data via mapreduce and mysql does it sequentially. I guess after loading data into mysql, the biggest difference is that the data in mysql requires compression and it's done by two external binaries myisampack and misiamchk.

[Please submit the code for the ETL job in your ZIP file]

General Questions

1. **Would your design work as well if the quantity of data would double? What if it was 10 times larger? Why or why not?**

Since we chose Mysql as our database for phase 3, and SQL databases are known for better ACID controls but not scalable, we would definitely change our design work.
First of all, instead of storing a type of query in one table, we would partition the table so that a range of keys are stored in a partition. This allows a much faster elimination of rows, and thus a much better throughput than a single table.
Secondly, or better yet, instead of using MYSQL, we would use mysql cluster or hbase, which allows much better scalability. (mysql cluster due to its sharding and replication techniques, and hbase due to its columnar storage system).

2. **Did you attempt to generate load on your own? If yes, how? And why?**

Yes, we did this because we had to keep the ELB warm while waiting right before the live test. We also did this because for some reason the load generator provided to us did some slow start mechanism for Q1. It greatly affects the throughput generated for smaller timeframe tests.

3. Describe an alternative design to your system that you wish you had time to try.

If we had more time to do this project, we would try to tune Hbase till it's more refined and compare the results with what we got with mysql. In addition, we would partition all tables so that most of them are stored in memory. We didn't store any in memory because the engine that we used (MYISAM) requires a huge key buffer size, which allows the database to really quickly find the exact location of the data.

4. **Which was/were the toughest roadblock(s) faced in Phase 3?**

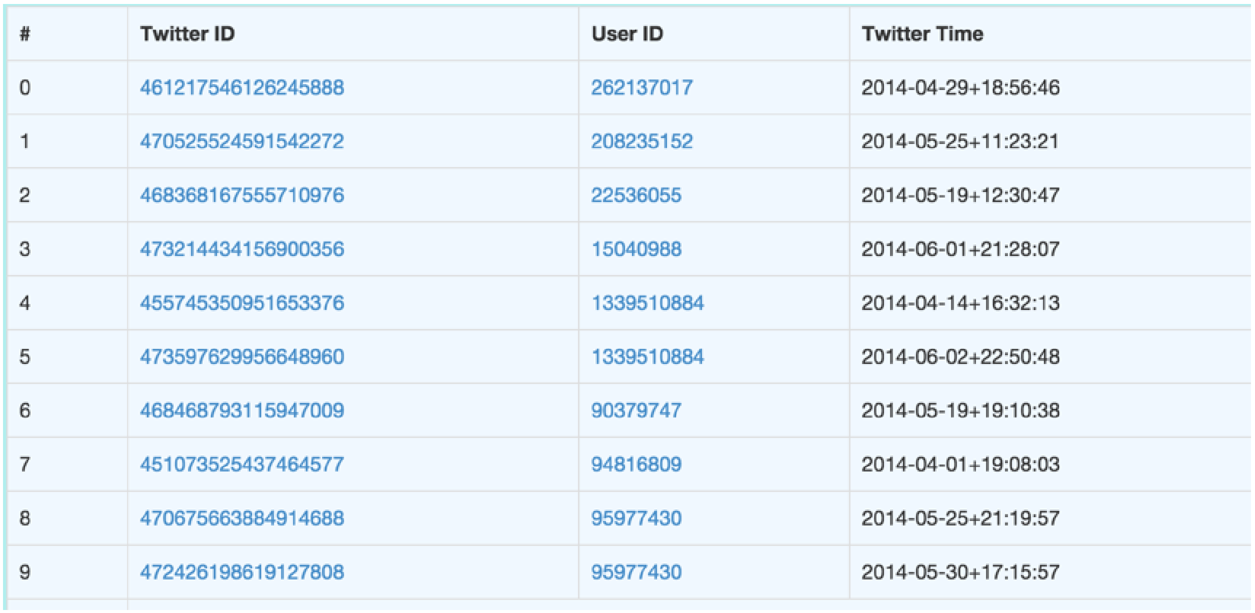
One of the roadblocks we encountered was optimizing the query statement so that it uses the index efficiently and not scan through the whole row.(though we quickly found out that myisam stores metadata about the table, and allows a simple lookup for the count of a table, which we used for q6). The biggest roadblock that we had was regarding Q1. We spent a lot of nights testing and trying to find out why Q1 was not performing as fast as back in phase 1. We even digged so deep as to sniffing the number of TCP connections that's coming from the load generator and the location of packets at a given time. (Since load generator only sends another new packet in one connection after receiving a response.) But we still failed to get full score for q1 at the end, despite the high scores we got for other queries.
At first we thought it was because we were threading (asynchronous calls) at our front end and with that combined with the ELB, it somehow worsened the throughput. However we did test ELB+threading and ELB+non-threading, and both of them produced the same results.

5. **Did you do something unique (any cool optimization/trick/hack) that you would like to share with the class?**

Sleep.time = 0

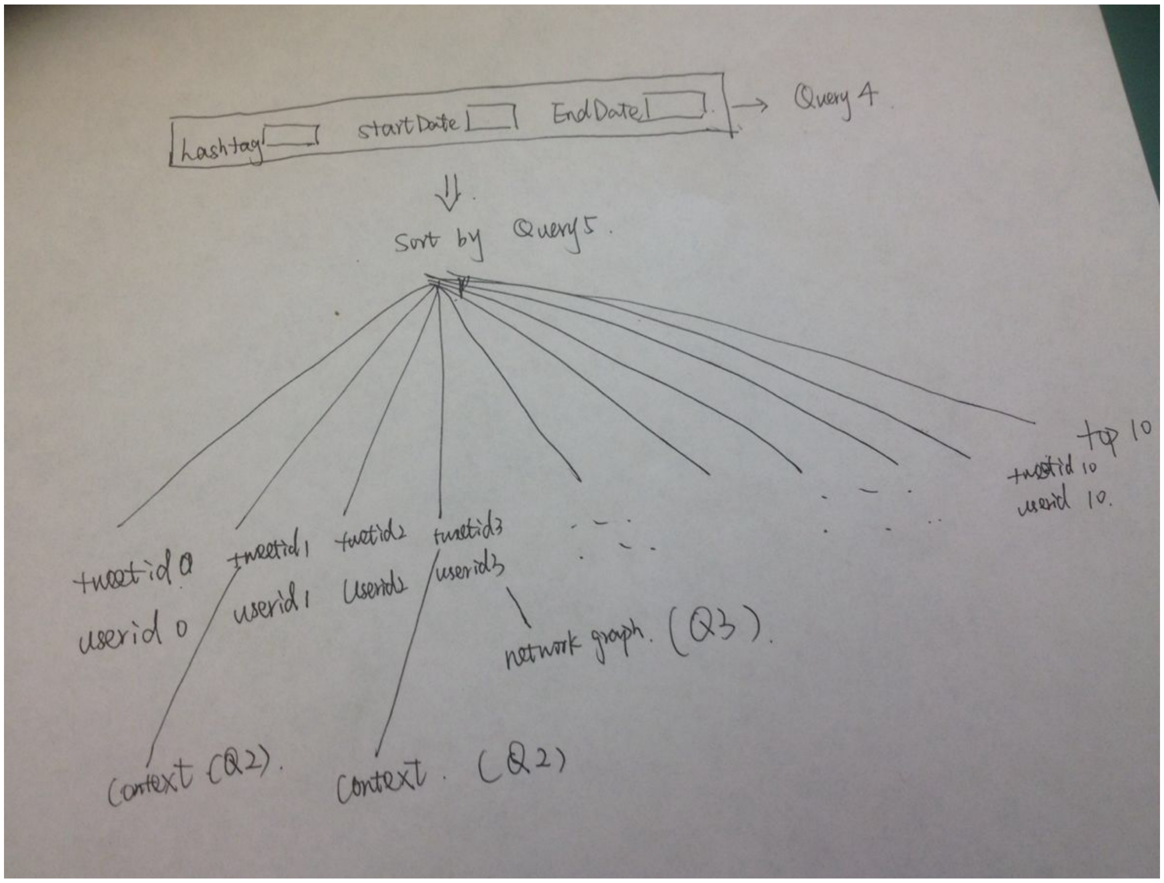
Bonus Questions

1. Draw the Data Dependency DAG of fan-out requests (see P3.3 recitation slide 10 for an example of this graph) for the following sequence of events:
    a. Open the bonus page
    b. Search #beautiful between 2014-01-01 and 2014-12-31
    c. Click on the first 10 tweets
    d. Click on the first 10 users



#	Twitter ID	User ID	Twitter Time
0	461217546126245888	262137017	2014-04-29+18:56:46
1	470525524591542272	208235152	2014-05-25+11:23:21
2	468368167555710976	22536055	2014-05-19+12:30:47
3	473214434156900356	15040988	2014-06-01+21:28:07
4	455745350951653376	1339510884	2014-04-14+16:32:13
5	473597629956648960	1339510884	2014-06-02+22:50:48
6	468468793115947009	90379747	2014-05-19+19:10:38
7	451073525437464577	94816809	2014-04-01+19:08:03
8	470675663884914688	95977430	2014-05-25+21:19:57
9	472426198619127808	95977430	2014-05-30+17:15:57

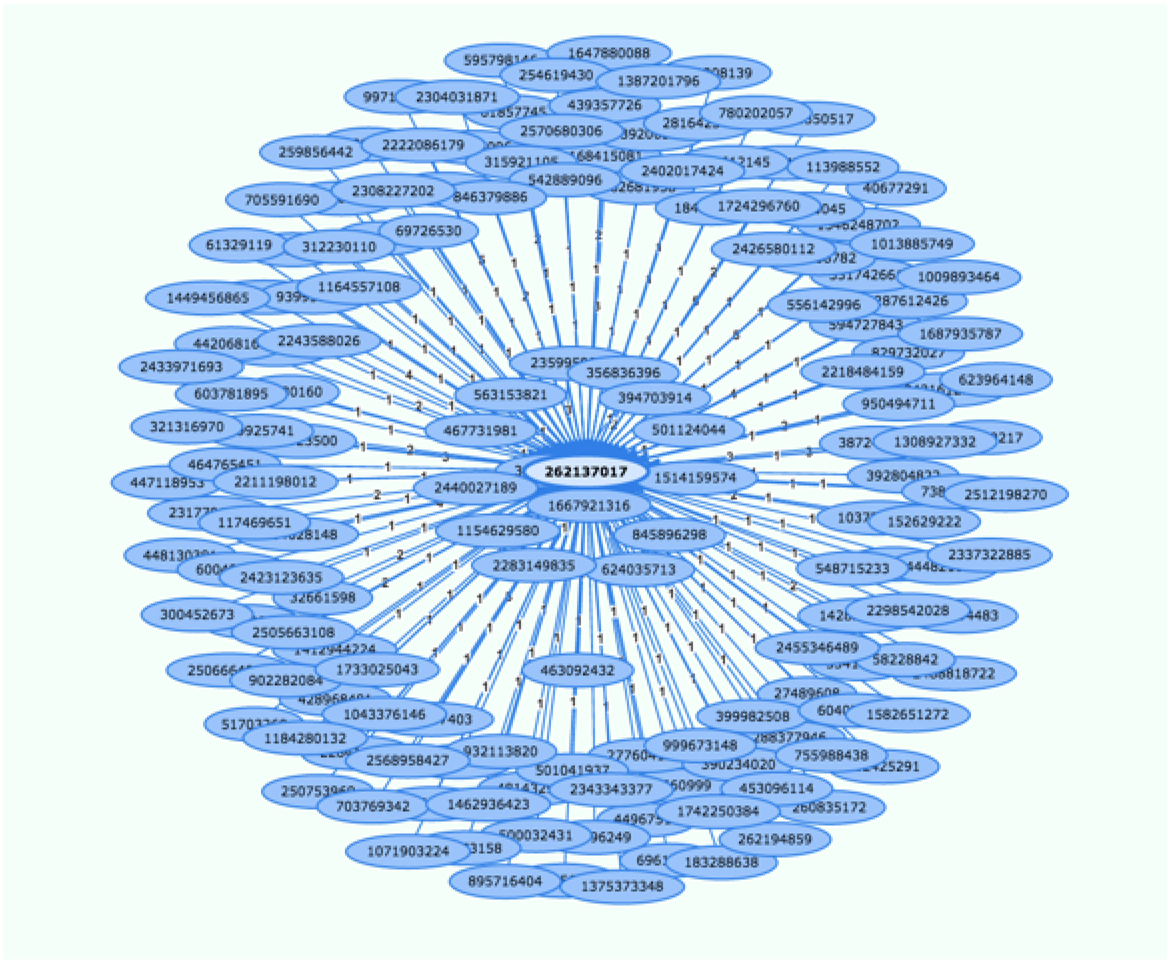
Data Dependency DAG :



2. Did you use any parallelization to speed up this data fetching at the front-end? Did you use any front-end web technique to make it appear faster for a visitor to the web page?

No, We do not use any parallelization to speed up this data fetching. We only use Javascript with ajax to get asynchronized request from the web server. It will allow to serialize the data required by user. It is not necessary to load the whole page. So this web technique will make the request faster and better to feel.

3. Show the network graph from your browser when 1a-1d are performed.
   The following image is the example when we click the top1 userid(262137017):


262137017
1647880088
254619430
1387201796
439357726
2304031871
2570680306
780202057
2222086179
259856442
2402017424
113988552
40677291
542889096
705591690
2308227202
846379886
1724296760
61329119
312230110
69726530
2426580112
1013885749
1009893464
1449456865
1164557108
556142996
1687935787
2243588026
2433971693
356836396
2218484159
623964148
603781895
563153821
394703914
950494711
321316970
467731981
501124044
1308927332
447118953
2211198012
1514159574
2512198270
2440027189
1667921316
117469651
152629222
1154629580
845896298
2337322885
2283149835
624035713
548715233
2423123635
32661598
300452673
2298542028
2505663108
2455346489
1733025043
58228842
902282084
463092432
1043376146
399982508
1582651272
1184280132
999673148
932113820
755988438
2568958427
501041937
2343343377
453096114
703769342
1462936423
1742250384
260835172
1071903224
262194859
183288638
895716404
1375373348