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: 6 m1.large as the front end connected behind the load balancer having 1 m1.large back-end connected to each.
Cost per hour (assume on-demand prices)	Live Test: $2.35
Queries Per Second (QPS)	

	Q1	Q2	Q3	Q4	Q5	Q6
score	121.49	1.62	43.2	43.68	4.83	146.43
tput	18225.8	126.2	4408.2	2675.1	228	14643
ltcy	5	403	11	18	221	3
corr	100	78	98	98	85	100
err	0.01	1.27	0.01	0.02	0.39	0.00

Rank on the scoreboard:	Phase 1: 36 Phase 2: 19 Phase 3: 35

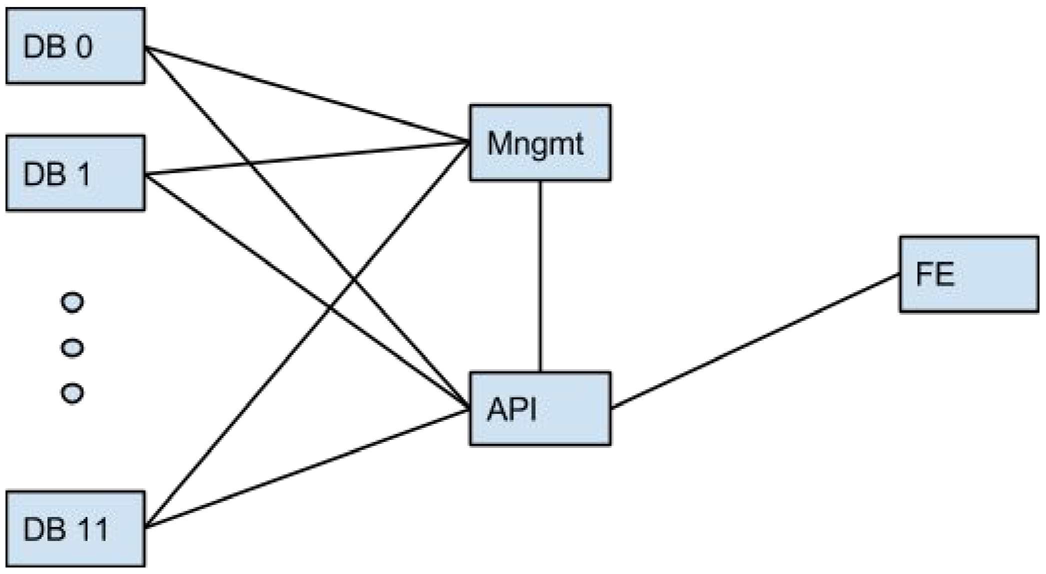
Team : (-_-)
Members : Abhishek Garai, Mithun Mathew, Jiayi Zhu

Rubric:

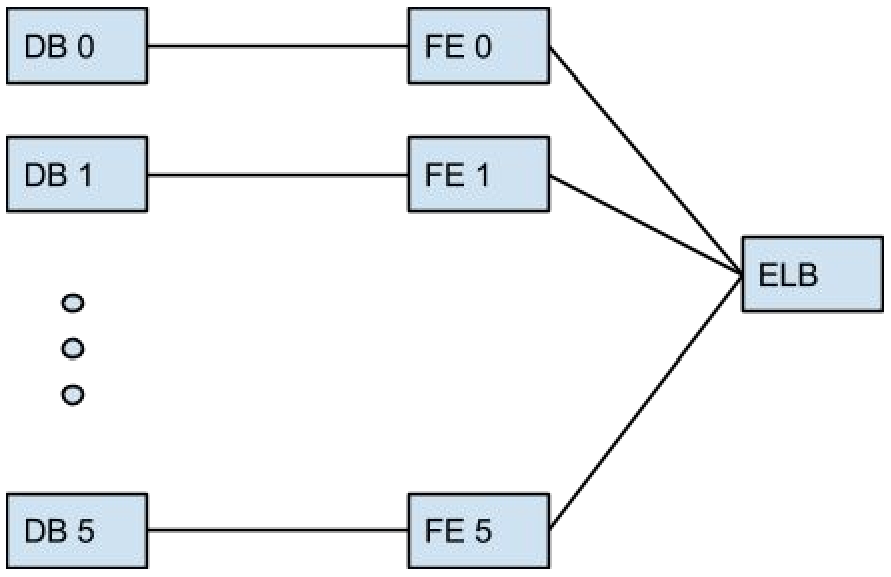
Each unanswered question = -10%
Each unsatisfactory answer = -5%

[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)]

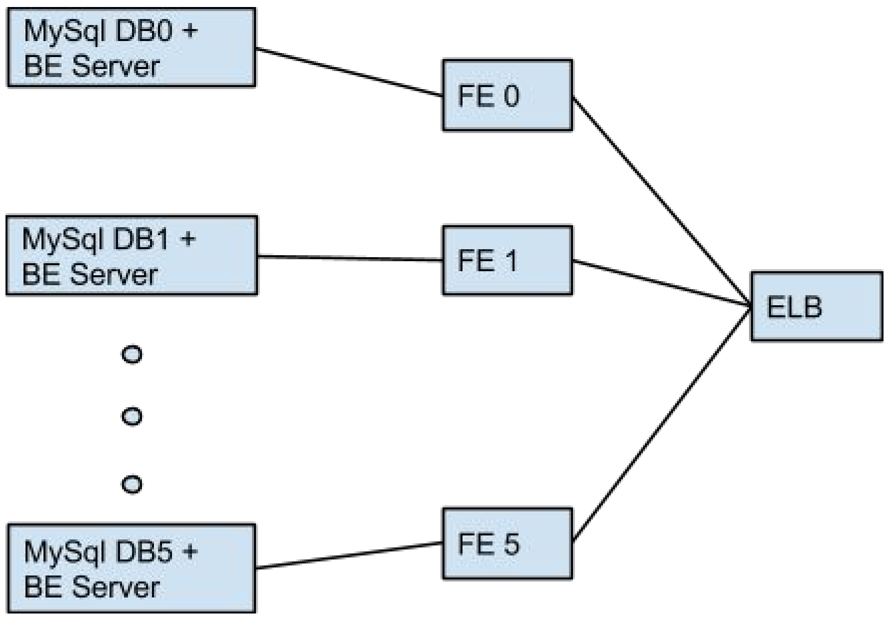
Phase 1 Topology (Sharded)



Phase 2 & 3 Topology (Replicated)



Phase 3: What we Wanted to TRY



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].

   A. We used undertow web-framework as the front-end web framework for our project. Firstly, we chose undertow because of its best database access response time for m1.large EC2 instances i.e. the front end instance type, according to the results from Tech Empower. Secondly, when we implemented other frameworks like java servlet for Apache Tomcat we attained lower throughput using the same number and type of instances i.e. undertow performed better for the optimizations that we could perform on both the types of frameworks. Also, in Phase 3, we experimented with the different variations of the undertow framework i.e the Builder API, Servlet version and the Listener configuration. Although as per the documentation the Servlet version would provide the best response but we could not tune the parameters properly to get the desired results and thus remained with the Builder API and got our service working. Some of the Properties are listed as below:

LightWeight- Various options for deploying
HTTP Support- Support HTTPS requests and responses
Web Socket Support
Servelet 3.1-Web Servlet deployment
Embedded-Embedded using few lines of code also using wildfly
Flexible-Various configuration options

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

No we did not change our Framework, although we experimented with some others as well. We used undertow for Phase 1 and Phase 2. We tried to change the webserver and performed testing with Apache Tomcat, Java Servlet and different configurations of undertow. But after test runs we found that undertow had a better performance as compared to servlets, resulting in higher throughput and lower latencies for equivalent configurations. Apparently we switched back to undertow for the further phase. Specifically for Phase 3, apart from just the framework we also tried different combinations of the performance enhancement in the front-end by using Prepared

Statements, increasing the IOPS, caching all the data for all the queries similar to the distributed Key-Value pair learnt in Project 3.5, using several design configuration of the front-end and back-end combination using sharding, replication before finalising on the design and the framework.

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

   Since we had the restriction on the front-end instance type of m1.large, firstly, we performed the tests using the back-end instances of type m3.medium and m3.large. Initially, we chose m3.medium because we thought that the back-end instance only had the database server running and only had to fetch the data and respond which would not require much processing power. Although, we did not take into account that the database fetch was a memory intensive operation. So, further we used m3.large hoping to utilize the memory available for fetching the data from the database. But due to the difference in the processing power of both m3.large front-end and m1.large back-end the performance of the operations went down, that we monitored using the cloud watch. And so finally we decided to settle for m1.large instance type as both the front-end and the back-end instance. Although we had a plan to use more number of back-end instances but restricted to six instances for front-end, back-end configuration with the ELB as we expected that it would cross the budget. In phase 3 we performed an experiment with a smaller dataset, which gave us an improved performance but could not implement as we ran out of time. It was related to the Front-End, Back-End combination i.e. we used both the Front-End and Back-End as m3.large and gained a higher performance in a Test DataSet for Query 2. But due to the lack of time for Loading the whole DataSet we could not actually implement it. Although we were pretty sure that it would have improved the RPS as the ECU was higher for m3.large and it had a higher memory as well.

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

   We used prepared statement in this phase to create the Mysql queries. The advantage to the prepared statement is that in most cases, this SQL statement is sent to the Mysql database right away with the only modification being the value of the parameters for each run of the query. So when the prepared statement is executed, the Mysql database can just run the prepared statement without having to compile it multiple times for each query, rather can just compile it ones and substitute the values of the parameters each time. During the tests we found that using it could improve the query performance to some extent. Also, we cached the Query6 dataSet completely on our Front-End thus not requiring to access the database for the requests for it. We also planned to shard the complete database for and store it in the 45 GB of memory available with all the Front-Ends, but due to the time constraint and coordination we missed on that.

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.

   So, after the experience we had in phase1 with the ELB we tried different configurations for distributing the load across the back-end instances. So apparently, we tried the following configurations. 1) One Load balancer connected to multiple front-end instances and they in-turn connected to one back-end instance each. 2) Having a single front-end and multiple back-ends and distributing the load among the back-ends within the front-end program. 3) Deploying the front-end and the back-end within the same instance to reduce the network latency of connecting the back-end and the front-end. And, finally we settled with load balancing multiple front-ends each with a single back-end. Although we knew that this would not have been the best strategy, but due to lack of time to experiment we settled with it. Load balancing is important in cloud because of the very nature of the structure i.e. there exists multiple physical infrastructure behind the hypervizer or virtualization software. So, it's kind of mandatory that the the hypervisor should have sufficient capability to utilize the resources present behind it very efficiently by distributing the load equally for optimal performance. Also, load balancing in general provides better efficiency as every instance is being utilised to the optimum capacity independently.

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.

   Our experience and experiments with different types of ELB was restricted to phase 1 and we used the ELB for phase 2. We explored the load balancers like HA Proxy, nginx and Varnish. Although ELB delayed the forwarding of the requests, but we could not configure the other load balancers to reach the expected performance after load balancing. For instance we tested nginx using two m1.large back-end instances and expected a higher throughput. But apparently we got a 16000 rps with one m1.large and 8000 rps with two m1.large instances behind nginx load balancer. We could not figure out the reason for this behaviour and hence settled with ELB as that was not reducing the throughput to such an extent.

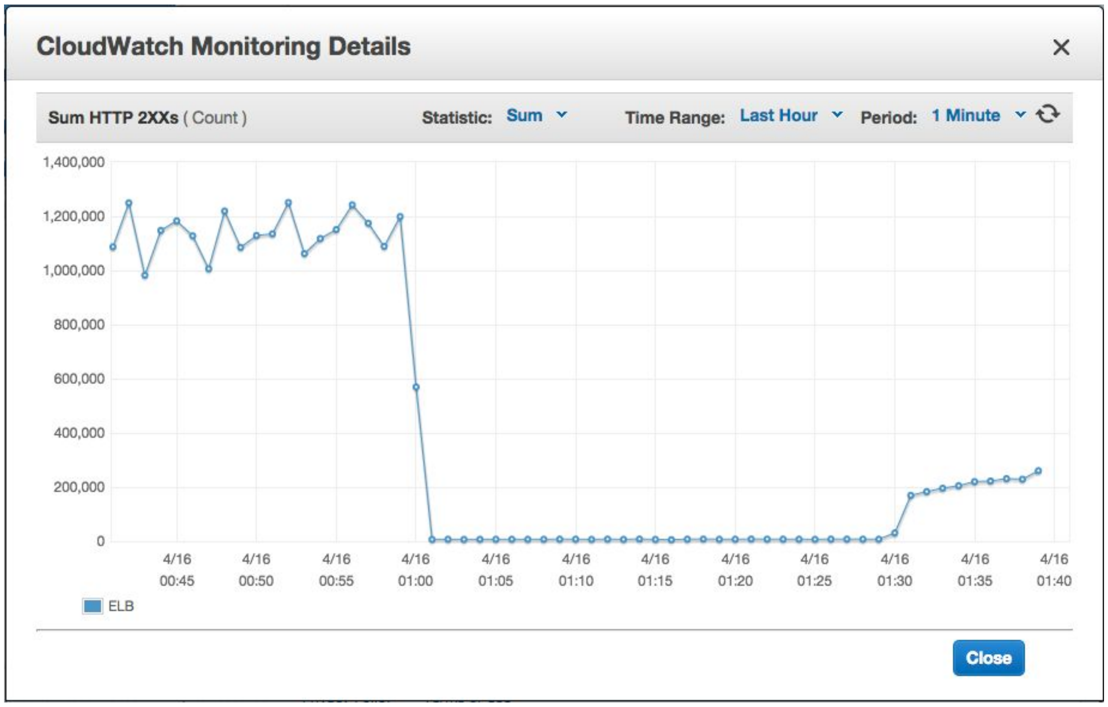
7. Did you automate your front-end instance? If yes, how? If no, why not?

   In terms of automating the front-end instance, we experimented with wildfly and used the undertow server from within it, we could have automated the front-end instance to start the undertow server once the instance was launched. But later we imported the undertow library within the instance and launched the server manually each time. This was because although we installed wildfly for using undertow we could not figure out the initialization script to start the server on boot-up. Further we found starting the server manually allowed us to configure the heap memory that was allocated for server. This allowed us to accurately measure the amount of memory utilised for each query as we experimented by modifying the memory allocation and testing each query to find the amount of caching required for each.

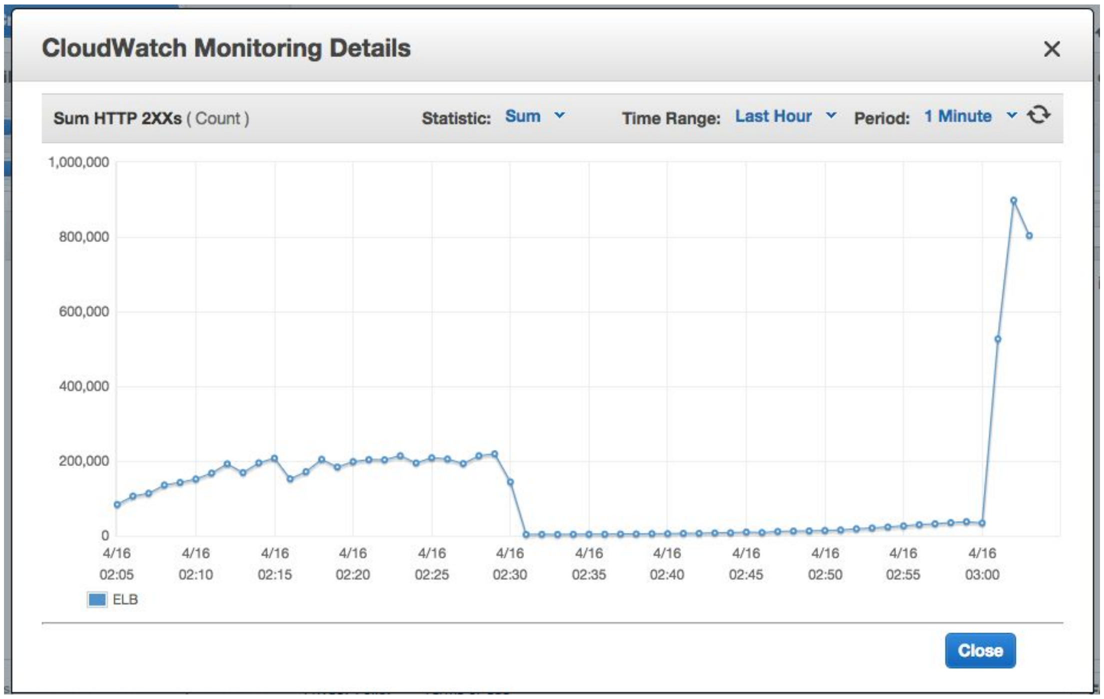
8. Did you use any form of monitoring on your front-end? Why or why not? If you did, show us a capture of your monitoring during the Live Test. Else, try to provide CloudWatch logs of your Live Test in terms of memory, CPU, disk and network utilization. Demarcate each query clearly in the submitted image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

   We were constantly monitoring the instances behind the ELB to ensure that they are healthy, up and running. The ELB parameters were also monitored during the live test. The number of requests coming in for Q1 were high compared to that of other queries. We also noticed the latency shown by the ELB was a good approximation of the latency on our scoreboard.

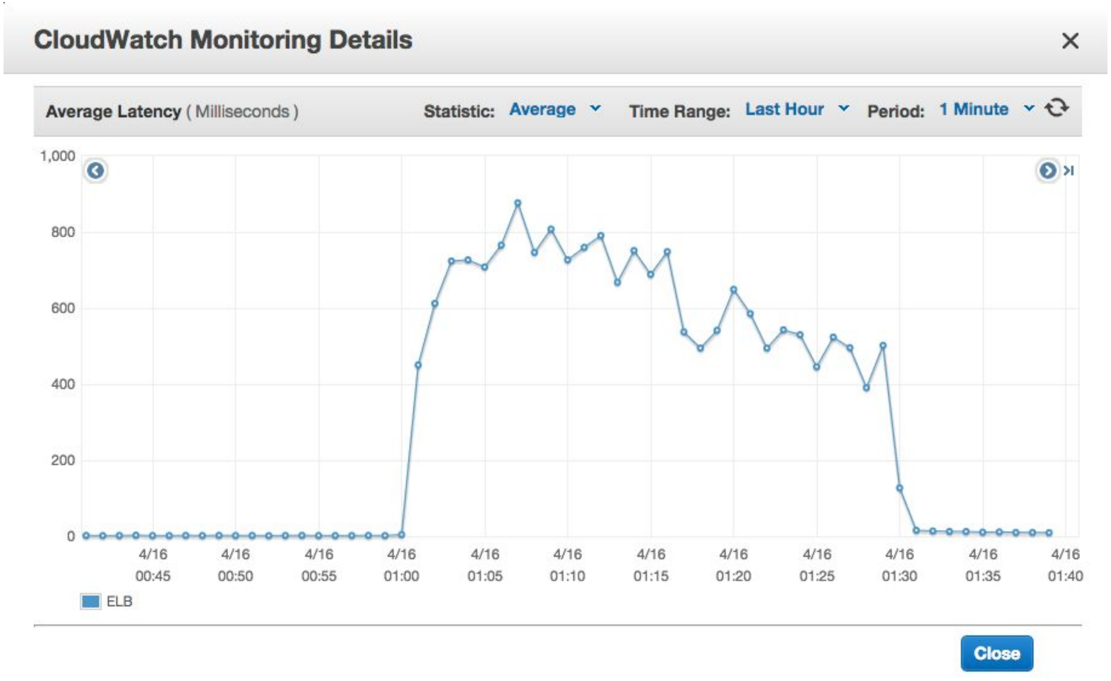
   The number of HTTP 5xx were very low (some appeared towards the end of the test). Latency for Q2 and Q5 were considerably high (a little higher than we expected) compared to the individual submissions we made for each query after warmup. Q1 did way better than expected reaching 18k rps.



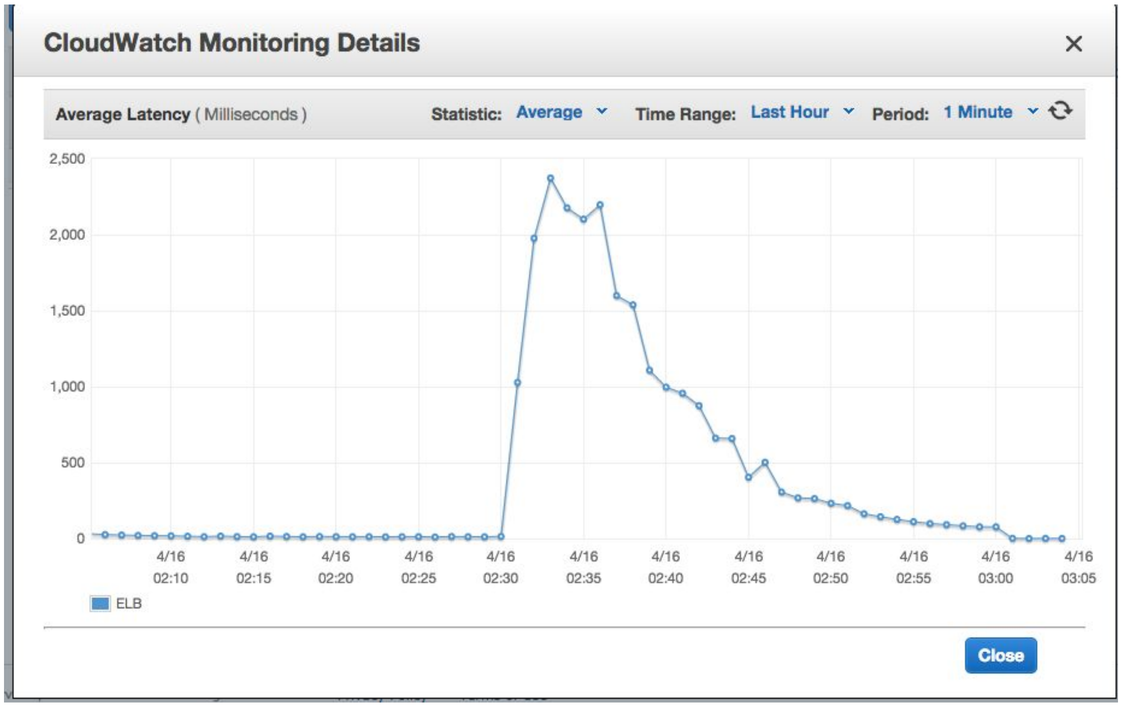
(Q1, Q2 and Q3 Sum HTTP 2XXs Requests)



(Q4, Q5 and Q6 Sum HTTP 2XXs Requests)



(Q1, Q2 and Q3 Average Latency)



(Q4, Q5 and Q6 Average Latency)

The CPU usage for the FE instances was around 20% during Q2 execution. During Q1 phase, CPU usage was much lower. All our instances were working during the live test.

9. What was the cost to develop the front end system?
   A. For MySql, we launched 6 instances for front-end behind the load balancer. So the total cost approximated to $ 1.2.

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

    https://www.safaribooksonline.com/library/view/wildfly-performance-tuning/9781783980567/ch07s03.html

    http://www.javacodegeeks.com/2014/01/entering-undertow-web-server.html

[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?

MySQL DB was chosen for phase 3. During phase 1 and phase 2, we used MySQL DB and achieved considerably good throughput for queries which were stored as request-response format in MySQL.

Q5 required group by and order by clauses according to our table structure. We found this much easier to implement in MySQL. Q6 was cached in the front-end to achieve high throughput.

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>**.

MySQL DB Design - Iteration 1

Table: query2

Column	Data Type	Description
user_id	DECIMAL(15,0)	user_id of the user publishing the tweet
date_created	DATETIME	time format in YYYY-MM-DD+HH:mm:ss
tweet_id	DECIMAL(25,0)	number format to enable sorting
tweet_info	VARCHAR(250)	score from sentiment analysis and the censored text (each tweet is max 140 characters long)

Table: query3

Column	Data Type	Description
user_id	DECIMAL(15,0)	user_id of the user
response	VARCHAR(2000)	response containing the retweet relationship of the user - the response was created during the MapReduce job

Table: query4

Column	Data Type	Description
hashtag	VARCHAR(35)	hashtag to be searched for
tweet_id	DECIMAL(25,0)	number format to enable sorting
user_id	DECIMAL(15,0)	user_id of the user
date_created	DATETIME	time format in YYYY-MM-DD+HH:mm:ss

Table: query5

Column	Data Type	Description
user_id	DECIMAL(15,0)	user_id of the user
date_created	DATETIME	time format in YYYY-MM-DD+HH:mm:ss
unique_tweet	DECIMAL(5,0)	no. of unique tweets by the user on that date
friend	DECIMAL(5,0)	no. of friends the user has on that date
follower	DECIMAL(5,0)	no. of followers the user has on that date

Table: query6

Column	Data Type	Description
user_id	DECIMAL(15,0)	user_id of the user
rank	INT	auto-incrementing key as userids(sorted in ascending order) are loaded into the table

MySQL DB Design - Iteration 2

All the tables described below were our final design for phase 3. We changed the DB engine to MyISAM for phase 3. InnoDB was used in phase 1 and 2. Changes from previous iteration are highlighted in light yellow.

Table: query2

Column	Data Type	Description
user_id	DECIMAL(15,0)	user_id of the user publishing the tweet
compare_date	VARCHAR(14)	time format in YYYYMMDDHHmmss
tweet_id	DECIMAL(25,0)	number format to enable sorting
tweet_info	VARCHAR(250)	score from sentiment analysis and the censored text (each tweet is max 140 characters long)

Table: query3

Column	Data Type	Description
user_id	DECIMAL(15,0)	user_id of the user
response	VARCHAR(2000)	response containing the retweet relationship of the user - the response was created during the MapReduce job

Table: query4

Column	Data Type	Description
hashtag	VARCHAR(35)	hashtag to be searched for
tweet_id	DECIMAL(25,0)	number format to enable sorting
user_id	DECIMAL(15,0)	user_id of the user
date_created	DATETIME	time format in YYYY-MM-DD+HH:mm:ss (used for response)
compare_date	VARCHAR(8)	time format in YYYYMMDD (used only for request queries and indexing)

Table: query5

Column	Data Type	Description
user_id	DECIMAL(15,0)	user_id of the user
compare_date	VARCHAR(8)	time format in YYYYMMDD
unique_tweet	DECIMAL(5,0)	no. of unique tweets by the user on that date
friend	DECIMAL(5,0)	no. of friends the user has on that date
follower	DECIMAL(5,0)	no. of followers the user has on that date

Table: query6

Column	Data Type	Description
user_id	DECIMAL(15,0)	user_id of the user
rank	INT	auto-incrementing key as userids(sorted in ascending order) are loaded into the table

The reason for changing the date format is because the indexing on DATETIME field was not efficient. Also for Q4, two columns were used to store date, one for WHERE clause and the other for selection.

Our next proposed iteration for date data type was integer or number instead of VARCHAR for date_compare field. We did not have enough time to implement this.

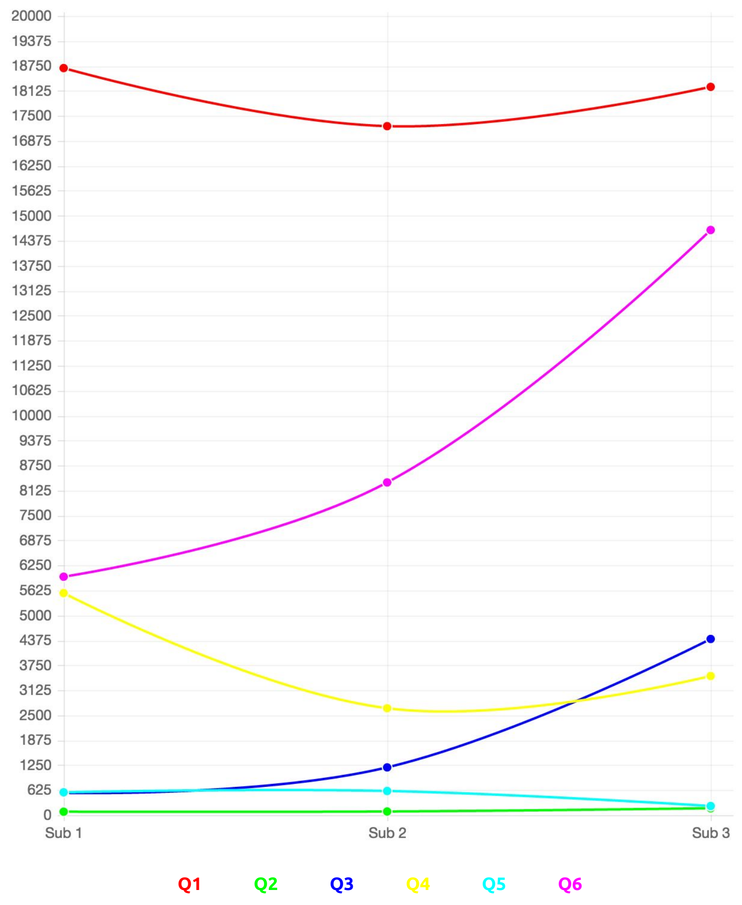
3. What was the most expensive operation / biggest problem with each DB that you had to resolve for each query? Why does this problem exist in this DB? How did you resolve it? Plot a chart showing the improvements with time.

For the previous queries we have already mentioned our roadblocks in the respective report. But specifically for Phase 3 we had a clear idea that for Query6 we would be putting the whole dataset in the Front-End as it was quite small and for Query5 also we were planning to put everything in the Front -End by sharding but the roadblock that we faced while implementing them was due to the range that was present within the request and we actually had to find all possible combination to store it in the database. Although we tried with different data structures in the front-end to find an optimum design to store the requests of Query 5. But we could not optimize it to return a better RPS and hence planned to remain with the design we had previously.

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)**

- Implementing query cache
- Using EXPLAIN for optimization
- Avoid retrieving all columns in the table
- Limit the number of records
- Place the index on the search field (requested parameter)
- Index on join columns
- Always have a primary key
- Disabling unique checks during load phase if not required

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).



The rps was normalized and the no. of submissions were scaled down to accommodate it in the graph.

Q1 and Q6 were cached in the front-end (high rps)
Q2 and Q5 took longer time at the backend DB to execute (low rps)
Q3 had a request - response format, however was not cached in the front-end due to lack of memory (after caching Q1 and Q6).

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

The design would not work for insert/update requests. The system topology that we used for phase 3 is the same as phase 2. “The design would not work for insert and update requests. This is because the system works as 6 independent systems behind a load balancer (refer Phase 2 Structure). Even if the FE can handle PUT request, the load balancer sends the request only to one of the FEs, thus inserting/updating only one of the systems, which would lead to inconsistency.”

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?

The API driver that we used to connect to the Back-End was mysql jdbc driver. We had experimented with both the Asynchronous and the Synchronous versions of the driver for either of the phases. For phase 2 where we could not make the Async version work, we actually made the Async version work in Phase 3. But since it did not give good overall throughput across all the queries we decided to implement the Synchronous version as we could optimize specific elements to get a better RPS.

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?

The front-end RAM was used for caching Q6 completely. Q1 was also cached on the front-end. The RAM to be used during the run of the service was set to 7GB, most of which was occupied by Q1 and Q6. The remaining approx. 0.5 GB was left for the working of the server and any other functioning that required memory. During the experiments our server crashed several times during to memory unavailability and this allowed us to look into the exact memory utilization and configure to allocate the memory accordingly.

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

Separate tables were used for Q2-Q6. We used separate tables just to simplify the querying to the database, although later we realised that we could have normalised the table structure as the queries had repeated information being queried from the table. We actually planned to merge the tables for Query5 and Query 6 i.e. use the same table for Query 6 directly from Query 5 but later planned to put Query 6 completely on cache. Although we understood that normalization would have reduced the amount of loading required for each query and also we could have utilised the memory for caching more data and improve the performance of the queries.

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

One of the flaws that we noticed was that due to some reason the indexing was not working properly or was getting messed up. Infact we executed a series of test this fact, i.e. after loading the data ad before indexing we actually performed some tests and got the baseline RPS, later we indexed the tables on certain columns that were being queried and performed the tests again. This time the RPS improved , but after several runs of the Test it suddenly started giving very poor RPS. Later when we re-indexed the columns by dropping the index the RPS improved again to the expected value.

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.

NOTE: The percentages are estimates based on what we have observed from running tests. No accurate profiling was done at each level.

	Q2	Q3	Q4	Q5	Q6
Load Generator to Load Balancer (if any)	12.5%	15%	12.5%	12.5%	25%
Load Balancer to Web Service	7.5%	10%	7.5%	5%	20%
Parsing request	5%	5%	7.5%	2.5%	7.5%

Web Service to DB	5%	10%	10%	5%	0%
At DB (execution)	40%	20%	25%	50%	0%
DB to Web Service	5%	10%	10%	5%	0%
Parsing DB response **(compute diff. for Q6)**	5%	5%	7.5%	2.5%	2.5%
Web Service to LB	7.5%	10%	7.5%	5%	20%
LB to LG	12.5%	15%	12.5%	12.5%	25%
	100%	100%	100%	100%	100%

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

We implemented 6 m1.large instance as our backend servers that approximated to $1.05 considering the on-demand price.

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

MySQl: http://www.tutorialspoint.com/jdbc/jdbc-sample-code.htm
https://code.google.com/p/async-mysql-connector/wiki/UsageExample
https://code.google.com/p/async-mysql-connector/

Hbase: http://hbase.apache.org/book.html

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

Task 3: ETL

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

MapReduce programming model was used for all three queries for the extract and transform phase. The reducer output was adjusted in such a way that we had a ",\t" separated file of fields, which could be directly used for loading using LOAD DATA command in MySQL.

    b. The number and type of instances used and justification

EMR

Note: The no. and type of instances used for the EMR job were based on a basic approximation of time taken for the job on a dataset. They do not imply exact time calculations.

Q2

m1.large: 1 master and 12 cores

One file took 7 minutes with 1 m1.large master and 1 m1.medium core
793 files approximation = 793 * 7/60 = 92 hours
Dividing by a factor of 4/3 = 70 hours (adjusting between large+medium to large+large)
For 12 cores = 70/12 ~ 7 hours expected

Original time taken: 4 hours 27 minutes

Q3

m1.large: 1 master and 19 cores

One file took 2 minutes with 1 m1.medium master and 1 m1.medium core
793 files approximation = 793 * 2/60 = 27 hours
Dividing by a factor of 2 = 13.5 hours (adjusting between medium+medium to large+large)
For 19 cores = 13.5/19 ~ 42 min expected

Original time taken: 45 minutes

Q4

m1.xlarge: 1 master and 10 cores

One file took 2 minutes with 1 m1.medium master and 1 m1.medium core
793 files approximation = 793 * 2/60 = 27 hours
Dividing by a factor of 4 = 6 hours (adjusting between medium+medium to large+large)
For 10 cores = 6/10 ~ 36 min expected

Original time taken: 12 hours (This is the first time we used xlarge with hopes of getting faster results compared to large instances. But this turned out to be the least efficient EMR job we ever ran.)

Q5
m1.large: 1 master and 19 cores

One file took 10 minutes with 1 m1.medium master and 1 m1.medium core
793 files approximation = 793 * 10/60 = 132 minutes
Dividing by a factor of 2 = 66 minutes (adjusting between medium+medium to large+large)
For 19 cores = 66/19 ~ 3 min expected
Original time taken: 48 minutes

Q6
m1.large: 1 master and 19 cores

One file took 10 minutes with 1 m1.medium master and 1 m1.medium core
793 files approximation = 793 * 10/60 = 132 minutes
Dividing by a factor of 2 = 66 minutes (adjusting between medium+medium to large+large)
For 19 cores = 66/19 ~ 3 min expected
Original time taken: 35 minutes

Load
DB Nodes: 6x m1.large
FE Nodes: 6x m1.large

To handle multiple requests and achieve higher throughput, we configured six databases with its own front end. The load balancer sends requests to each front-end using round robin logic.

For phase 1, there was only one frontend which did not handle as many requests as expected and gave us low throughput. For phase 2, instead of using a MySQL replicated DB cluster, we configured the system to work as 6 independent services behind a load balancer, which worked out better than phase 1. The same topology was used for phase 3.

c. The spot cost for all instances used

m1.medium (test only)	: $0.0081/hr
m1.large	: $0.0161/hr
m1.xlarge (EMR only)	: $0.0321/hr

d. The execution time for the entire ETL process

Q2: 4 hours 27 minutes
Q3: 45 minutes
Q4: 12 hours
Q5: 48 minutes
Q6: 35 minutes

e. The overall cost of the ETL process

$34

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

Q5: 0
Q6: 0

g. Discuss difficulties encountered

Q5: Finding an efficient way to index and store the table without group by or order by.

h. The size of the resulting database and reasoning

Total Size after Extraction and Transformation 81GB. Q5 contributed more to the increase in DB size compared to Q6.

i. The size of the backup

We did not create a backup for the DB. Instead we created an image of the instance on which the DB was installed and data was loaded.

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?

Disabling the tables unique key checks and not defining a primary key for the MySQL table. Also disabling MySQL bin logging before loading are some of the approaches.

We did not make any changes to the table structure or the data it held after the load.

Load times (approx.):
Q2: 1 hour 35 minutes
Q3: 15 minutes
Q4: 25 minutes
Q5: 50 minutes

Q6: 5 minutes

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

We used Non-streaming version of EMR. The streaming EMR did not prove to be the best solution as the tweets had multiple lines and could not be read on a line by line basis. The wrapper class Text provided a good encapsulation for multi-line tweets and was one of the main advantage that we had in using non-streaming EMR.

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

No external tool was used for loading data. MySQL's LOAD DATA command was used. We also tried experimenting with the Bulk Load option, but it failed due to some reason and all the Database nodes used to get disconnected suddenly. So, we went for using the Local infile load for loading the database on a file by file for each table. And as we had our data replicated we actually loaded data in a single instance and launched multiple instances for that.

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

MySQL. This is because it has got a better documentation and more support in terms of utilities and options available for loading i.e. Bulk Loading, Local infile Loading and different configurations available within them to fit the different options and requirements.

Our answer has not changed in the last four weeks. We still feel that MySQL is easier to load and query.

[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?

If the data were double, the current DB would be able it. Our main concern regarding increased amount of data is the response time during querying. The indexing did not seem to be as efficient as we thought since the query time was not as low as expected. Increased amount of data would further increase the query time.

More data -> larger index -> slower response time

Sharding would have solved the issue but we actually ran out of time and coordination to work on an implement it.

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

No we did not generated load on our own this time because we had performed rigorous testing in phase 2 to check the front end speed and working. Although, we performed testing by generating load for Queries 2-4 in the previous phases by writing a java code that would send dummy requests to the load balancer to warm it up and also check the correctness to some extent for the values of userids that were present in the dummy requests. Further we had to concentrate on the working and correctness of Queries that we tested by submitting rather than generating the load by ourselves.

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

Apart from whatever we mentioned and tried in the previous phases, specifically for phase 3 we had a much better design that we wanted to implement, but due to lack of coordination we could not make it. The design would have been similar to the distributed key-value cache that we had learnt in project 3.5. The design would have been, having multiple front end behind an ELB and connected to multiple backends in a mesh topology i.e. all the front ends being connected to all the back-ends in order to facilitate sharding of the query data. Also the backend database would have a server running to receive the requests and fetch it from the database. The front-end server would cache all the queries i.e. Q3-Q6 and the back end server would store only Query 2 in a sharded fashion. This would have allowed us to utilise the memory to the maximum and also achieve a higher rps.

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

Finding efficient indexing strategies to improve from phase 2 - date time format was converted to VARCHAR format with hopes of getting throughput.

The connectivity between the front-end and the back-end that was always our concern i.e.

whenever we fetched something from the database it gave a very low throughput and RPS.

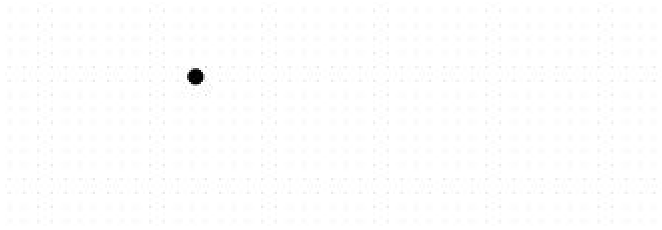
5. Did you do something unique (any cool optimization/trick/hack) that you would like to share with the class?

Q6 userids were sorted and loaded in to the DB with auto-incrementing key, which was used as the rank. The whole Q6 data was loaded into the front-end memory as a hashmap (userid, rank) before starting the server. Fetching the whole Q6 data to the front end with a single query was taking too long and JDBC closes the connection due to time out. To work around this, Q6 data was fetched from the backend 5 million rows at a time (using LIMIT and OFFSET in MySQL) to avoid time out After starting the server, when a request comes in, the hashmap was accessed with the help of methods floorEntry() and ceilingEntry(). Subtracting the ranks gave the required response.

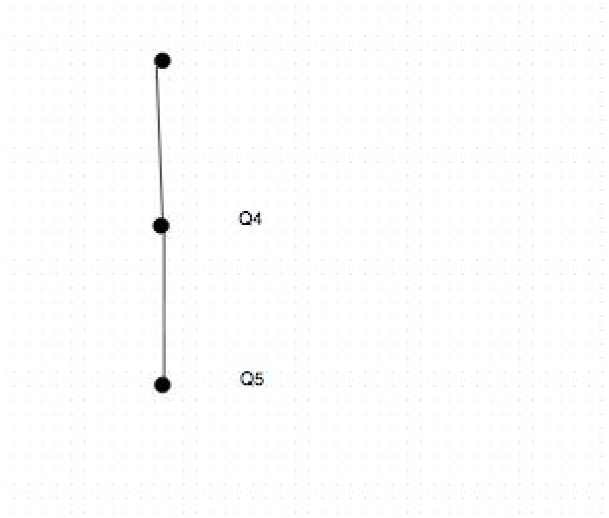
For Q4, the request date contains only YYYY-MM-DD, but the response date field requires YYYY-MM-DD+HH:mm:ss. Two different columns were used for this. One for querying (YYYY-MM-DD) with index. The other was selected for response.

Bonus Questions

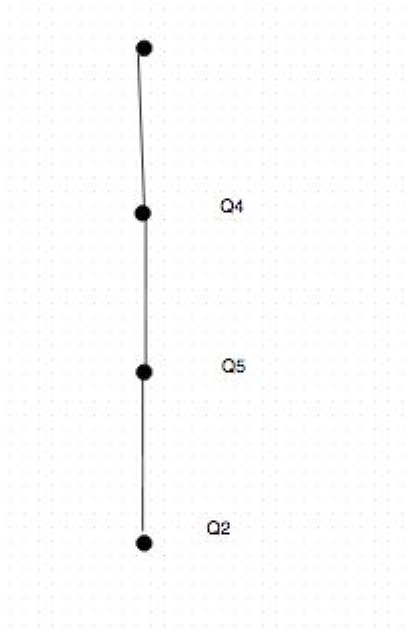
1. Draw the Data Dependency DAG of fan-out requests (see P3.3 recitation slide 10 for an example of this graph) for the following sequence of events:
   Open the bonus page



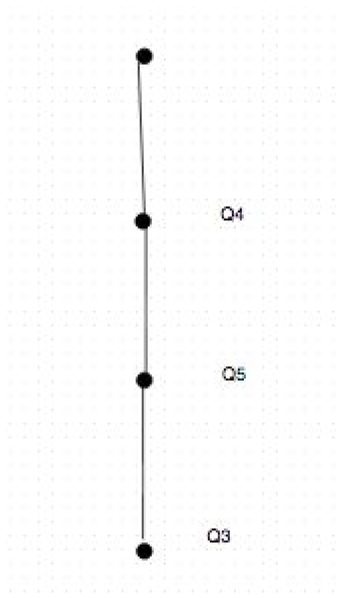
   Search #beautiful between 2014-01-01 and 2014-12-31



   Click on the first 10 tweets



   Click on the first 10 users



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?

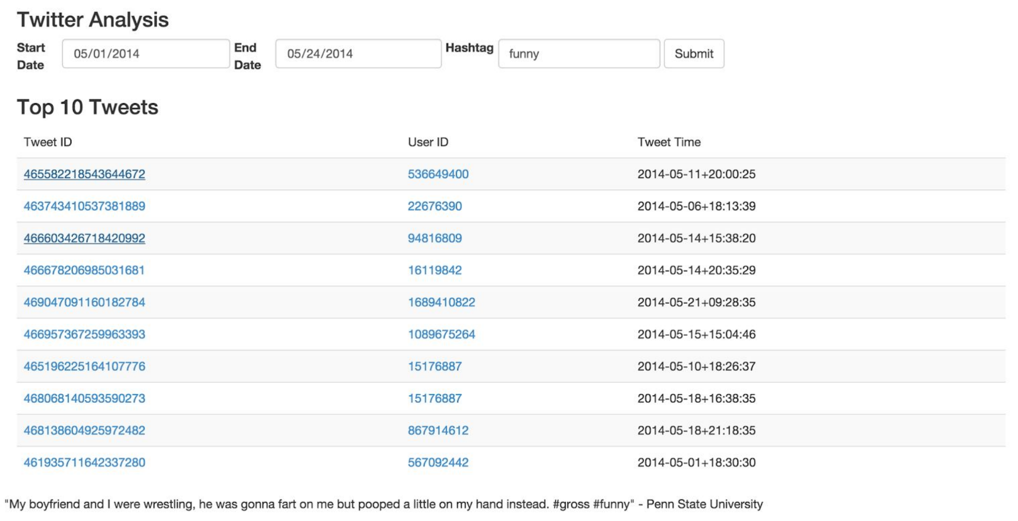
No we haven't used parallelization and other techniques. We just implemented the basic functions for the UI part.

3. Show the network graph from your browser when 1a-1d are performed.

We haven't collected network graph when we do the bonus part. To save the budget we closed the instance now so we are unable to collect them.

Screenshots of the WebPage

1. Showing top ten tweets for hashtag, funny between 05/01/2014 and 05/24/2014. Tweet with tweet ID 465582218543644672 is shown at the bottom of the table.



Twitter Analysis

Start Date: 05/01/2014 End Date: 05/24/2014 Hashtag: funny

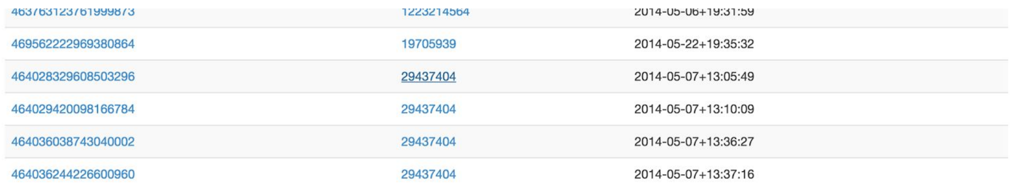
Submit

Top 10 Tweets

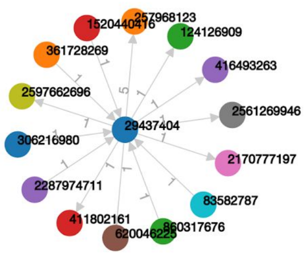
Tweet ID	User ID	Tweet Time
465582218543644672	536649400	2014-05-11+20:00:25
463743410537381889	22676390	2014-05-06+18:13:39
466603426718420992	94816809	2014-05-14+15:38:20
466678206985031681	16119842	2014-05-14+20:35:29
469047091160182784	1689410822	2014-05-21+09:28:35
466957367259963393	1089675264	2014-05-15+15:04:46
465196225164107776	15176887	2014-05-10+18:26:37
468068140593590273	15176887	2014-05-18+16:38:35
468138604925972482	867914612	2014-05-18+21:18:35
461935711642337280	567092442	2014-05-01+18:30:30

"My boyfriend and I were wrestling, he was gonna fart on me but pooped a little on my hand instead. #gross #funny" - Penn State University

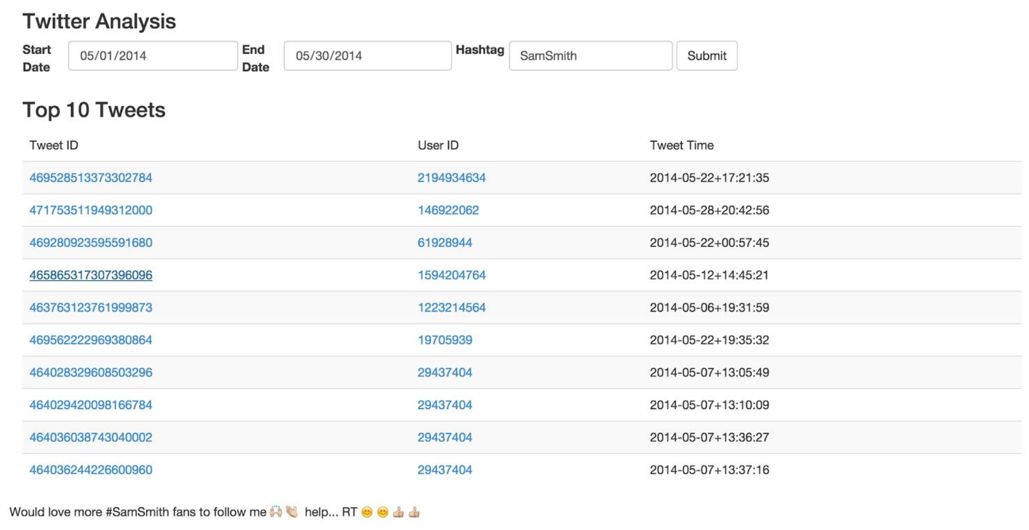
2. Showing network of user 29437404. Each node corresponds to a user. The arrows indicate if the user was tweeted or re-tweeted. The numbers on the edges connecting two nodes indicate the number of tweets/re-tweets.



463763123761999873	1223214564	2014-05-06+19:31:59
469562222969380864	19705939	2014-05-22+19:35:32
464028329608503296	29437404	2014-05-07+13:05:49
464029420098166784	29437404	2014-05-07+13:10:09
464036038743040002	29437404	2014-05-07+13:36:27
464036244226600960	29437404	2014-05-07+13:37:16



3. Tweet with smileys on it (shown at bottom of table)



Twitter Analysis

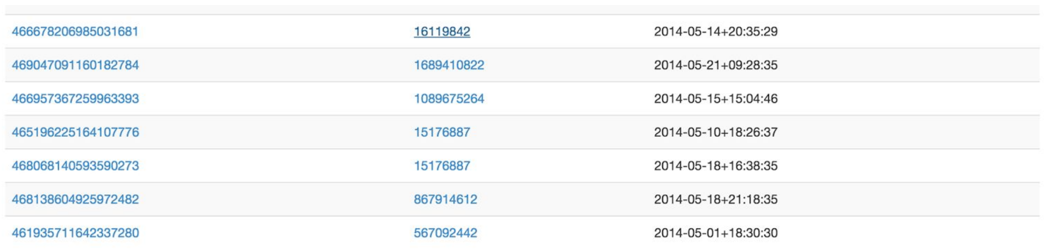
Start Date	05/01/2014	End Date	05/30/2014	Hashtag	SamSmith	Submit

Top 10 Tweets

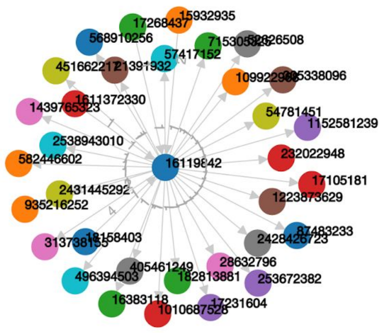
Tweet ID	User ID	Tweet Time
469528513373302784	2194934634	2014-05-22+17:21:35
471753511949312000	146922062	2014-05-28+20:42:56
469280923595591680	61928944	2014-05-22+00:57:45
465865317307396096	1594204764	2014-05-12+14:45:21
463763123761999873	1223214564	2014-05-06+19:31:59
469562222969380864	19705939	2014-05-22+19:35:32
464028329608503296	29437404	2014-05-07+13:05:49
464029420098166784	29437404	2014-05-07+13:10:09
464036038743040002	29437404	2014-05-07+13:36:27
464036244226600960	29437404	2014-05-07+13:37:16

Would love more #SamSmith fans to follow me 🙌 👏 help... RT 😊 😊 👍 👍

4. A user with a larger network of tweets/re-tweets



466678206985031681	16119842	2014-05-14+20:35:29
469047091160182784	1689410822	2014-05-21+09:28:35
466957367259963393	1089675264	2014-05-15+15:04:46
465196225164107776	15176887	2014-05-10+18:26:37
468068140593590273	15176887	2014-05-18+16:38:35
468138604925972482	867914612	2014-05-18+21:18:35
461935711642337280	567092442	2014-05-01+18:30:30


15932935
17268437
568910256
52826508
57417152
1611372330
1439765323
54781451
1152581239
2538943010
582446602
16119842
232022948
17105181
2431445292
935216252
1223873629
87483233
2428426723
18158403
28632796
405461249
496394503
182813881
253672382
16383118
17231604
1010687528