[Team Name] Up_in_the_Air

[Team Members] (alphabetical order by first name)

Qian Mao (qianm), Yinsu Chu (yinsuc), Zhuolin Liu (zhuolinl)

15-619 Project Report

Table of Contents

1 Statement of Assurance 4

2 Web Service Overall Design 4

2.1 Summary of Web Service Architecture 4

2.2 Database Selection and Database Schema 4

3 Implementation 5

3.1 Front End 5

Grizzly or Tomcat 5

Initial Instance Choice and Front-End Configuration 5

3.2 Back End (HBase) 6

3.3 ETL 7

3.4 System Test 8

3.5 Optimize for Throughput and Latency 10

Large or Small 10

Schema Partition 10

Front-end Cache 10

In-memory Data 10

Large data choice 10

Small Data ETL 11

Small Data Choice 11

Optimization Algorithm 12

System After Optimization 13

Optimization Result 13

No ASG 13

Test Result of Plan #1 (9 m1.medium) 14

Test Result of Plan #2 (4 m1.large) 14

3.6 Provision, Load and Prepare for Live Test 14
4 Live Test Performance Analysis 16
5 Review and Suggestions 18
Appendix 18
Source Code Structure 18
ETL/LoadHBase.java 18
ETL/parser_json_csv.py 18
back_end/ 18
front_end/air_grizzly/* 18
front_end/air_tomcat/* 18
utils/Record.java, utils/FormTable.java 19
utils/ExternHelper.java, utils/Merge.java 19
utils/FrontEndASG.java 19
utils/PreprocessHBaseCSV.java 19
utils/RandomRequestTest.java 19
utils/TestReadHBase.java 19

1 Statement of Assurance

We certify that all of the material that we submit is our original work and we write this report only by ourselves.

2 Web Service Overall Design

2.1 Summary of Web Service Architecture

In web service architecture, we primarily think to use some service that write in Java first, because all of us have a strong background in coding in Java. Using other service such as Python (such as Django), JavaScript (such as Node.js) or PHP may be time cost, and cannot promise a better web performance. Based on this, we decide to choose one of two frequently used Java web service containers (Tomcat or Grizzly). As I will talk about in detail in section 3.1, between these two, Tomcat has slightly better performance and is more robust. So, after a series of performance tests, we decide to use Tomcat as our web service architecture.

2.2 Database Selection and Database Schema

As for database, we actually changed our design several times. At first, we chose HBase as our database for all queries, since at first glance of the data, we thought the size of data is too large to be totally put into MySQL. However, after we parsed the data from S3, and put what we need into HBase, we find the data of q3 and q4 is not large (only 312MB in total) at all. After that, we thought the size of data is small enough to be put in memory. Then, thinking about 4.97GB q2 data, we tried to turn to MySQL and finally find the performance of MySQL is better than HBase, because when putting all q2 data in MySQL in front end machine, we got more budget to optimize the merged backend and frontend throughput in the same time via ELB. So, our final decision is q2 data in MySQL, q3 and q4 data in memory and using ELB to get a higher throughput.

As for Database schema, in HBase at first, we use most basic schema for q2. The key is time and column family is named “tweet” which contains one column named “content” (tweetid + tweet). For q3 and q4, the key is user ID, and the column family is named “tweet2” which contains a column that has the accumulated number of tweets from user 1 to current user ID (for q3); and another column that has the retweet list (for q4).

After we put data into memory, the HBase only left the data we need for q2. For q3 and q4 in memory, we create an optimized algorithm to decrease the size of in memory data size and keep the data searching cost in O(1). This algorithm is based on the index technology in search engine and will be described in detailed in section 3.5.

After we put data into MySQL, the data of q2 stored in MySQL became: [primary key: time, conlumn1: content]

Since we change our design for so many times, we will show the performance of each design and the reason why we change the design or schema in detail in following sections. HBase and memory performance will be showed in section 3.2, MySQL and memory will be showed in section 3.5.

3 Implementation

3.1 Front End

Grizzly or Tomcat

Grizzly is a lightweight server which can directly be launched by running Java project with grizzly JAR file. It is commonly used in JUnit test. However, Tomcat is much larger application faced and has a lot of peripheral support to make web service more robust and efficient.

For RESTful web service, we decide to use Jersey web service because it is the most famous Java-based RESTful web application. Also, the Jersey application can be perfectly compatible with Grizzly and Tomcat. So, we decide to make our choice between these two.

We implement two web servers, one with Grizzly and one with Tomcat. The code of tomcat is in air-tomcat repo and the code of grizzly server in air-grizzly repo. And we do the test for each server. As for Grizzly server, we get the q1 qps of 1600 in an m1.small instance and for Tomcat, we get the q1 qps for 1800. Since the performance does not have a too much difference, we finally decide to use Tomcat as our front end server because it is more robust and stable when receive large amount of data.

As a RESTful service implementation, we use Jersey as our package. We implement a listener to do some operations when initiating servlet and initiating database connection and parsing data file. Then for each query, we have a separate class to handle the query, basically get the data from database or searching data in memory and return the result. How to get the data from database or memory with be described in each database back-end part in detail.

Initial Instance Choice and Front-End Configuration

At first, we decide to use one m1.medium instance as our front-end (non-peak time) and it can scale out up to 8 m1.medium at peak time. First, this is cost reasonable since the maximum budge for front-end at non-peak time is $0.2 which means we can use at most one m1.medium or three m1.small. Second, compared to m1.small, m1.medium has more computation and space capacity to support more users accessing considering the overhead of the operating system and the server itself.

On our front-end platform, we deployed Tomcat 7 with Open JDK 6.0. Tomcat is configured to auto startup (crontab) on port 80. The allocated maximum heap size for Tomcat 7 is 4096M and the stack size is 2048M. Therefore, Tomcat 7 can use as much memory space as possible. To maximizing the performance of our front-end system, we stopped and removed any unnecessary services such as pre-installed MySQL server.

The cost rate of our front-end deployment is shown in the table below:

Front-end	Type/Number of Instances	Rate
Non-peak ($0.2)	M1.medium/1 + ELB/1	$0.145/hr
Peak ($1.0)	M1.medium/8 + ELB/1	$0.985/hr

The total development cost is roughly up to $1. Our team configured front-end platform on one m1.large instance with spot bid around $0.026. After the development and configuration, we created our own front-end image, so we can boot new instances with our front-end image without any configuration for future test and use. This is our initial solution.

However, as can be seen from following sections, we did not use ASG as our final optimized solution because it increases the error rate. As a result, we just use ELB and put 9 m1.medium behind it.

3.2 Back End (HBase)

We thought about HBase and MySQL and decided at first to use HBase. The reason is that HBase handles big data better than MySQL and is easier to configure (no need to load balance). The raw data is about 100GB, seems to be large.

Due to the cost constraint, we can only use 1 master and 2 cores in our HBase backend, which are all m1.large. This is almost the only choice that will not exceed the cost constraint. In this way, the cost is (0.24 + 0.06) * 3 = $0.9 per hour. We only use spot instances during our development, and spot instance cost is about $2, the total development cost is about $3.

As for Database schema, in HBase at first, we use most basic schema for q2. The key is time and column family is named “tweet” which contains one column named “content” (tweetid + tweet). For q3 and q4, the key is user ID, and the column family is named “tweet2” which contains a column which has the accumulated number of tweets from user 1 to current user ID (for q3); and another column which has the retweet list (for q4).

We can get the data from front end using HTable object. First we create an HBaseConfiguration, after setting the parameters (zookeeper, hbase master, etc.), we can use the configuration to instantiate an HTable object and use Get to find what we want.

However, after we put all data into HBase, the performance is not as good as we thought. After that, we did a performance optimization and only put q2 data into HBase since we found that q3 and q4 data is so small and is logical to put into memory. So in our improved backend design, only q2 is in HBase. (In our final optimized design, HBase is no longer used, please see later sections).

The code to parse q2 data from S3 to HBase is in air-utils/LoadHBase.java

Following is the HBase test result (front end: one m1.large, backend: 1 master + 2 cores, all m1.large):

q1	q2	q3	q4
3260.3 (error 2.17, 120s) (id 2761, 13ms)	3234 (information lost)	3730.6 (id 3019, 2ms)	7419.1 (id 3084, 2ms)
3353.0 (error 7.82, 300s) (id 2777, 12ms)	2600.5 (id 3511, 7ms)		
2904.5 (error 8.58) (id 3192, 14ms)	2261.8 (id 3372, 8ms)		
2893.8 (error 9.91) (id 3258, 14ms)	3190.3 (id 3552, 6ms)		
4107.4 (error 3.86, 60s) (id 3365, 9ms)	3138.7 (id 3588, 6ms)		
4178.5 (error 4.91, 60s) (id 3662, 9ms)	2600.5 (id 3511, 7ms)		
4412.8 (error 4.37, 60s) (id 3635, 8ms)	3190.3 (id 3552, 6ms)		

As can be seen from the table, we heavily tested q1 and q2 since q3 and q4 does not make much difference - they are in memory in our improved solution.

Actually if we use one m1.large as front end, the cost for front end will exceed. But we did not put all backend data in our backend system, so we asked a question can we merge part of backend with our frontend and adjust the cost accordingly (https://piazza.com/class/hjju766ub905i?cid=1171). After we got a "no" as the answer, we decide to use MySQL solution, so there was no need to test m1.medium as frontend with HBase.

3.3 ETL

The code for ETL is air-utils/LoadHBase.java

Note that this is not only the code we use in our design to read q2 from HBase, but also the code we use when we decided to use MySQL. When we decided to use MySQL, we simply export everything from HBase out to a .csv file and then import the .csv file into MySQL.

We use Map-Reduce for the ETL job. The major reason is that the amount of input data is huge. No matter what kind of output is needed (file, output to HBase or MySQL), the speed of parsing of the huge number of JSON can be dramatically improved by using Map-Reduce since the execution is parallelized.

The JSON parse operation is memory-intensive. We checked the recent spot instance price history, finding that m2.2xlarge is only $0.07, which is a good bargain. So we use m2.2xlarge as core and m1.large as master.

There are totally 1800 JSON files. When the cluster is made of 1 m1.large master and 1 m1.large core, time to parse one JSON file is about one minute, and total time is 30 hours. When we use 1 m1.large master and 1 m2.2xlarge core to parse one JSON file, the time is 30 seconds. We plan to make the ETL time within 2 hours, so we use 10 m2.2xlarge instances (The planning time is short because we only parse time and tweet for backend. For detailed schema please refer to the Optimization section).

10 m2.2xlarge spot instances ($0.07 + $0.21 each) and 1 m1.large spot instance ($0.026 + $0.06), the actual running time is two hours, so spot cost for one ETL job is $5.772. We did the ETL job two times, so total cost is $11.55 (The first time is to load data into HBase, and the second time is to generate .csv files for MySQL, this is because our design change, please refer to Optimization section for details).

Since we did not carry out the formal ETL job until we verified the correctness, there are no failed ETL jobs.

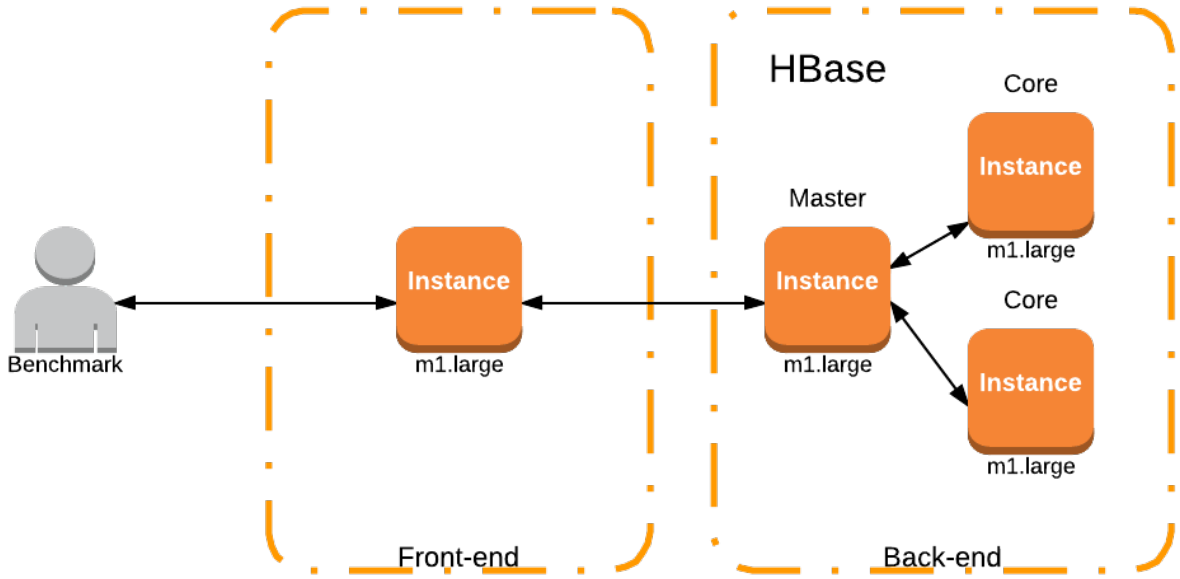
Difficulties are mainly how to load data into HBase, and this question is solved by using HBase client API (HTable, Put, Get, HBaseConfiguration, etc.). Also, for q2 we have to keep the tweet as the same in JSON, so we did not use any JSON package since they all automatically decode the JSON. We simple use string manipulation to extract the tweet portion directly from JSON (the part between "text:" and "source:").

The size of resulting database is about 5GB. Reasoning: the total number of tweets is 36229650 so each tweet is about 138 bytes, which is in a reasonable range. Since later we did not use HBase anymore, we did not keep a backup on S3, just export the .csv and import it to MySQL.

For q3 and q4, we still need a ETL procedure to get them from S3. We simply wrote a Python script and run it on our own laptop to generate q3 and q4 data. Then use the optimization algorithm to load them into memory.

3.4 System Test

After all of design as we have discussed above, our design is following:



Front end:

Tomcat Server with Jersey RESTful web service. Each class file handles a kind of request.

Back end:

For q1: There is a global variable to record the q1 answer string. On the other hand, there is a thread which updates this string every 500 milliseconds. So, whenever a request comes, we directly print the global string onto the screen.

For q2: Using HBase database. The key is time and column family is named “tweet” and the column is “content” (Set of tweeted:tweet) As a result, whenever a request comes, we select time from the database, get the content and directly print onto screen. (Note this is not the optimized solution)

For q3 and q4: we store all q3 and q4 results in memory instead of the database since the total size of the q3, q4 result is only 312MB. Also, according to the optimized algorithm we will describe sooner, we successfully decrease the data store memory from 4.5GB to 1GB and get the result almost in O(1) time. (Why we store q3, q4 in memory and the optimized algorithm will be described in section 3.5)

As for AWS resources, we have used EC2 for instances launching, EMR to load data from S3 to instances that we used and loading HBase based on that.

For instance view, we launch 1 m1.large instance in the front end machine and 3 m1.large instances on back-end machine (1 Master and 2 Cores). We think we can do that because the total budget is 0.24*4 + 0.06*3(EMR price) = 1.14 < 1.2 at any time. Also, since we store some of data in front end, we can borrow some backend budget to front end (but this has been refused on Piazza, see previous link to Piazza).

For the given workload in test service (note: this table is same as the previous table):

q1	q2	q3	q4
3260.3 (error 2.17, 120s) (id 2761, 13ms)	3234 (information lost)	3730.6 (id 3019, 2ms)	7419.1 (id 3084, 2ms)
3353.0 (error 7.82, 300s) (id 2777, 12ms)	2600.5 (id 3511, 7ms)		
2904.5 (error 8.58) (id 3192, 14ms)	2261.8 (id 3372, 8ms)		
2893.8 (error 9.91) (id 3258, 14ms)	3190.3 (id 3552, 6ms)		
4107.4 (error 3.86, 60s) (id 3365, 9ms)	3138.7 (id 3588, 6ms)		
4178.5 (error 4.91, 60s) (id 3662, 9ms)	2600.5 (id 3511, 7ms)		
4412.8 (error 4.37, 60s) (id 3635, 8ms)	3190.3 (id 3552, 6ms)		

The maximum throughput of your overall system for q1, q2, q3 & q4 is:

Q1 : 4412.8, Q2: 3234 Q3: 3730, Q4: 7419.1

3.5 Optimize for Throughput and Latency

Large or Small

Schema Partition

In our initial design, we put all data in one backend database. Soon, we found the performance of q2, q3 and q4 is not so good. The problem is our schema is not designed for a specific kind of query. We cannot balance every query in the concern of performance. Naturally, we turn to split our database into three schemas corresponding to three kinds of query. In addition, we pre-process the data to suit the query. For q2, we take time as key, list of [tweetId:text] as the value. So we can directly extract all [tweetId:text] at a giving time. For q3, we generate a table where the key is userId, the value is the sum of tweets that posted by the users whose userId is less than or equal to current userId. Therefore, for a giving q3 query, we can do two retrieving operations on database and do a simple subtraction. For q4, we created a table where the key is userId and the value is a list of users who retweeted any tweet posed by the userId. This improvement greatly increases the qps since there is little work other than retrieving data from database.

Front-end Cache

In addition, we decided to use cache in the front-end. The cache will cache q2, q3, q4 and their results. After test, we believe it is not a good idea because our front-end is computational and memory limited, however, the size of dataset is too large. We can only cache a very small part of dataset in front-end which is not efficient for improving performance.

In-memory Data

Giving up caching all queries and results, we got an intuition of in-memory data. Though we gave up caching, we were impressed by the front-end in-memory data performance. We found the largest bottleneck is the communication between front-end and back-end. We got the intuition of in-memory data because we found after the partition and pre-processing, the dataset for q3 and q4 is small (roughly 300MB in csv format) while the dataset for q2 is still big (roughly over 4GB). Since m1.medium instance has 3.75G memory, we can safely put dataset for q3 and q4 into front-end machine memory within 1GB if we carefully design the data structure. In theory, this change will greatly increase qps for q3 and q4 since there is no interaction with back-end system and all dataset is in memory and organized in hashmap-like list. Therefore, we can guarantee O(1) time to respond q3 and q4 query.

Large data choice

For back-end system, which is now supposed to storing data only for q2, we decide to use MySQL.

MySQL has three advantages over HBase. First, the dataset is small, it's more reasonable to keep all data in one machine. Second, MySQL is faster since it doesn't need any internal communication. Third, MySQL is much cheaper and flexible. MySQL can be configured on all types of instances, however, HBase requires at least m1.large instances and needs one extra master node to coordinate.

There is only one schema stored in MySQL:

Key	Value
Time: datetime	Tweets: blob

The type of key is datetime since it will be stored using numeric format and speed up lookup operation. The type of value is blob since it will provide maximum 64MB size and should be enough to store all texts posted at the same time. The content of tweet text is exactly same as that appearing in raw JSON file.

The tweets stored in the table have already been processed before stored. We combine all tweets posed at the same time using the format:

tweetId: text \n tweetId: text \n ……

This format is exactly the same with the output returned to user. Thus, we need no more processing but just retrieving from database.

Considering the communication bandwidth, the bottleneck is the transmission speed. Therefore, we decide to use m1.large as the back-end platform as it has the largest transmission capacity. The average performance of q2 is around 2,000qps.

The cost rate of the back-end system:

Back-end Budget	Type/Number of Instances	Rate
$1	M1.large/4 + ELB/1	$0.985/hr

Small Data ETL

As what we had said above, there are only 310MB data finally for query 3 and 4. What we need from the raw data for query 3 and 4 is just userId, number of tweets that user posted and the user that have ever retweet the user's tweets.

How to get this 310MB data from 100GB raw data? We directly read JSON record one by one from s3 and parse data to userid, retweet_userid from. The code of parsing data is in script/parser/parser_json_csv.py. Then, we went on to the second round parsing to make file can be read fast. So, we read userid, retweet_userid file line by line and put the data into a HashMap to get the number of tweets by one userID and his retweeter list. Finally, we formed other file whose schema is userid, num_of_tweets, retweet_list.

To get the first round data, we used 4local computer and 4 large instances to parse the row data simultaneously. We splited the whole data into 19 data sets and run one set each time. It took 10 hours in total to parse the whole data. Then, we parsed the file into what we need to read and it took 10 minutes on local machine.

The total spot cost of all instances in this step is 10*0.03 = $0.3

Small Data Choice

How to make use of 310MB data and make it as fast as we can to send back the result when requests coming? Basically, there are three type of data storage, in memory, in local disk, in remote machine. Apparently, storing in memory is the fast way we can do. Since we only have 310MB data, why shouldn't we do this? However, as the number of the data is still large of this 310MB data, we should definitely not

use linear search to get the result in O(n) time. We should try hard to decrease the time of sorting. In the next section, we will describe how to make the searching in O(1).

Optimization Algorithm

How to get the data in O(1) time for q3 and q4?

The first think is HashMap, because we can use userId as key and find the record based on userId, in O(1) time. However, HashMap will take too large memory, because as to HashMap, it will takes 8*Rawdata + 4*Capacity due to the HashMap load factor. So, it will take at least 4GB heap space to load the whole data into memory. But even m1.large instance has only 7.5GB and m1.medium: 3.75GB. To load the mysql and tomcat service stable and robust which all take some of heap space, we need to find a way to decrease the heap space.

In search engine technology, we know index is a good way find denote data in almost O(1) time without using HashMap. So we design an algorithm based on index.

First, we find there are only 18185738 users, but the userId can up to 2148717488. So, there will be 1 user in 10 userId slots on average. Based on this, we split the user into 2148718 groups, so that there will be only 10 users in one group on average. Then we create our index based on this. There is an array with 2148718 integers. Each element will record the first element's index in record table (which is also an array) of the latter groups. For example, the index[1342] will record the first index of the [1342000, userMax] user groups in the record table.

But how to store our record efficiently since we have an index which can just get the first element of one user group? First, we need an offset stored in each record which can help us to find specific user in a user group. And we set offset as "Short" type, which can decrease the space used. Then, to accelerate the response time of q3 which needs a range of users feature, we store the number of tweets from userId 0 to the current userId in records. So when we user_max and user_min, we just need to user_max.tweets – (user_min-1).tweets. Then for q4, we shore the retweeted userId list in bytes array mode and decode to String when return back the result.

So, the final record structure for q3, q4 is: an index integer array to keep track of the user group location and a record array to store each record. The query will be [number of tweet of user 0 to now: long, retweetIdList: bytes[], Idoffset: short]

For specific q3, we basically need to find "user_max.tweets – (user_min-1).tweets". But there are some times that we cannot find out the specific userId when that user is not record in our index. For this case, we need to find the userId that no larger than this userId. We first find the offset and the index number of this user, named "basic" and "offset". Then we compare index[basic] with index[basic-1]. If index[basic] = index[basic-1], the user group [basic*1000, basic*1000+1000] will have no user. For instance, index[1342] = 20= index[1341] which meacn [1342000, uMax]'s first user index will be 20 and equals to which in [1343000, uMax]. So, the [1342000, 1343000] will have no user. And the greatest user which no larger than current user whose index will be 19 which is index[basic]-1. However, if these two numbers are not equal, we can go through the group and find the highest id which no larger than current id combined with user offset. If all of the offset is larger, we return the index[basic]-1, to indicate this

element is the smallest one in his user group. After we find the two, we can get the number of tweets in nearly O(1) time.

For specific q4, we need to find whether that userId is in the record table. First we still get the index base and the offset. Then we are trying to see whether index[basic] = index[basic-1], if it is, we directly return null because there is no user in this user group. If it not, we will go through the user group and find the user. If we find the user successfully, we will decode the retweet bytes string into string and return.

After implementing this algorithm, we successfully decrease the heap size of our server from 4.5GB to 1G, which can even run on m1.small instance and get the q3,q4 result almost the same as q1

System After Optimization

After all of design and optimization as we have discussed above, our final design is following:

Front end:

Tomcat Server with Jersey Restful web service. Each class file handles a kind of request.

Back end:

For q1: There is a global variable to record the q1 answer string. On the other hand, there is a thread which updates this sting every 500 milliseconds. So, whenever a request comes, we directly print the global string onto the screen.

For q2: use MySQL database and make all data directly in front-end machine. The schema of the database is “time:content”. The content is what we should return back. As a result, whenever a request comes, we select time from the database, get the content and directly print onto screen.

For q3 and q4: we store all q3 and q4 results in memory instead of the database since the total size of the q3, q4 result is only 312MB. Also, according to the optimized algorithm we have describe above, we successfully decrease the data store memory from 4.5GB to 1GB and get the result almost in O(1) time.

As for AWS resources, we have used EC2 for instances launching, ELB for request load balance, EMR to load data from S3 to instances that we used and S3 to store intermediate result and backup database.

For instance view, we launched 9 m1.medium instances and connected with ELB. The choice of 4 m1.large instances and 9 m1.medium instances will be described in following section. The total on demand cost of 9 m1.medium + ELB will be $0.12 * 9 + 0.025 = 1.105 < 1.2$

Optimization Result

No ASG

After several tests, we decide not using ASG because of a high rate of error. The error rate can be up to 20% when the amount of queries varies a lot, which is unbearable. We have tried many configurations for our ELB including health check, but it’s still not clear why errors happen when auto-scaling group scale up or down. As a result, we gave up using ASG and re-designed our system based on the minimum budget. The final design of our system is shown as below:

Plan	Front-end & Back-end	Cost
1	M1.medium/9 + ELB/1	1.105
2	M1.large/4 + ELB/1	0.985

Test Result of Plan #1 (9 m1.medium)

q1	q2	q3	q4
4362.7 (id 3386, 11ms)	1979.4 (id 3390, 10ms)	2947.5 (id 3452, 3ms)	2473.9 (id 3454, 7ms)
2300.3 (error 12.86) (id 3503, 20ms)	5226.1(error 2.94) (id 3504, 3ms)	2952.4 (id 3542, 3ms)	6100 (id 3553, 3ms)
2149.9(error 16.02) (id 3580, 21ms)	6239.5 (id 3587, 3ms)	2800.6 (id: 3624, 3ms)	5849 (id 3634, 3ms)
		2865.7 (id : 3724, 3ms)	

Test Result of Plan #2 (4 m1.large)

q1	q2	q3	q4
2312.4 (error 14.64) (id 3513, 19ms)	2706.5 (id 3388, 7ms)	2454 (id 3477, 3ms)	5401.4c (id 3456, 3ms)
3119.3 (error 9.04) (id 3570, 15ms)	5190.0 (id 3571, 3ms)	2522.5 (id 3524, 3ms)	5633.9 (id 3611, 3ms)
		2786.7 (id 3612, 3ms)	

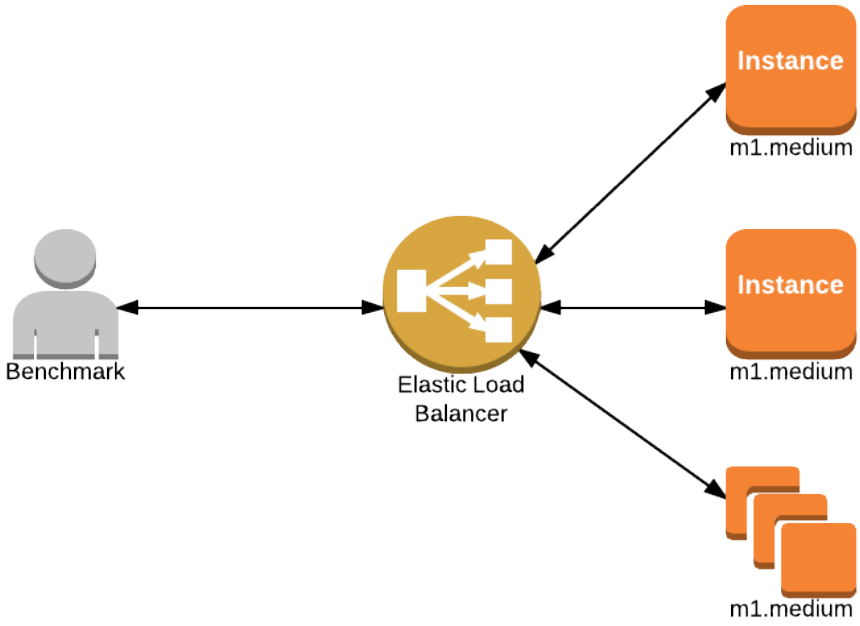
Based on the test results, we chose plan #1. This is because first, plan #1 has better performance except q1; Second, plan #1 has 9 instances which means it has larger capacity of transmission (m1.medium and m1.large has the same level of transmission capacity). Third, plan #1 is more robust because it has more instances. If one instance fails, we will lose 1/9 capacity in plan #1, but 1/4 capacity in plan #2.

3.6 Provision, Load and Prepare for Live Test

Our final configuration is shown as below:

Front-end & Back-end	Cost
M1.medium/9 + ELB/1	1.105

The structure of our system is shown below:



4 Live Test Performance Analysis

id	time	team name	URL						
	query	dur	throughput	latency	err.r	success	email		correct
req_id									
588	2013-11-27 23:02:25	Up_in_the_Air	UpInTheAir-1939154938.us-east-1.elb.amazonaws.com						
	q1	3600	980.4	4	1.52	1	yinsuc@andrew.cmu.edu		100
1220									
604	2013-11-28 00:02:32	Up_in_the_Air	UpInTheAir-1939154938.us-east-1.elb.amazonaws.com						
	q1	3600	11465.5	4	0	1	yinsuc@andrew.cmu.edu		100
1221									
632	2013-11-28 01:02:36	Up_in_the_Air	UpInTheAir-1939154938.us-east-1.elb.amazonaws.com						
	q1	3600	9381.4	5	0	1	yinsuc@andrew.cmu.edu		100
1222									
634	2013-11-28 01:02:36	Up_in_the_Air	UpInTheAir-1939154938.us-east-1.elb.amazonaws.com						
	q3	3600	1383.8	6	0	1	yinsuc@andrew.cmu.edu		100
1222									
635	2013-11-28 01:02:36	Up_in_the_Air	UpInTheAir-1939154938.us-east-1.elb.amazonaws.com						
	q4	3600	3479.2	5	0	1	yinsuc@andrew.cmu.edu		100
1222									
633	2013-11-28 01:02:36	Up_in_the_Air	UpInTheAir-1939154938.us-east-1.elb.amazonaws.com						
	q2	3600	3513.1	5	0	1	yinsuc@andrew.cmu.edu		100
1222									
685	2013-11-28 02:03:07	Up_in_the_Air	UpInTheAir-1939154938.us-east-1.elb.amazonaws.com						
	q2	3600	5890.2	3	0	1	yinsuc@andrew.cmu.edu		100
1223									
702	2013-11-28 03:03:10	Up_in_the_Air	UpInTheAir-1939154938.us-east-1.elb.amazonaws.com						
	q3	3600	2975.9	3	0	1	yinsuc@andrew.cmu.edu		100
1224									
722	2013-11-28 04:03:13	Up_in_the_Air	UpInTheAir-1939154938.us-east-1.elb.amazonaws.com						
	q4	3600	5947.7	3	0	1	yinsuc@andrew.cmu.edu		100
1225									

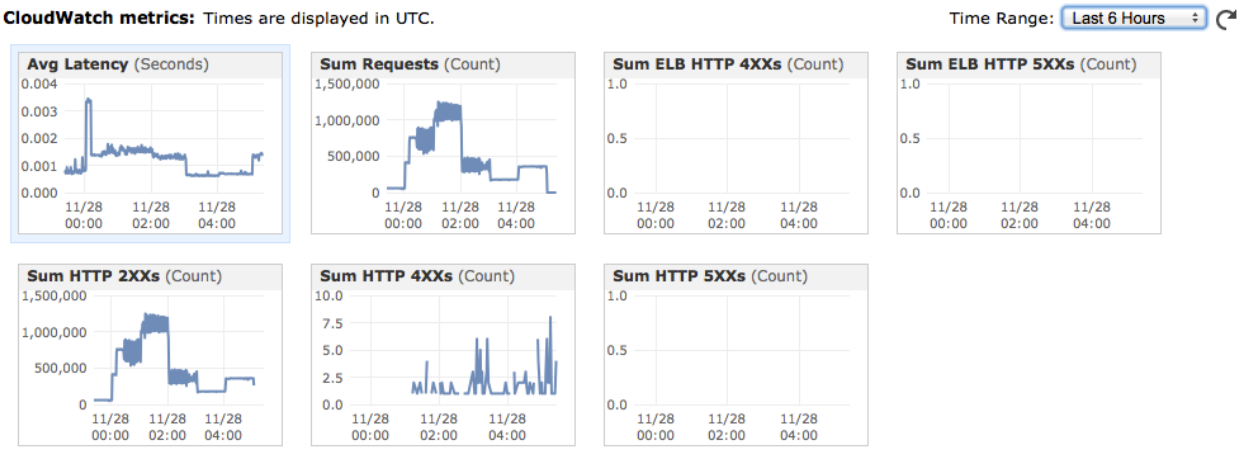
For the final result, some query types exceed our expectation while some others did not.

For q1, the first result is 980, which is apparently far below our capacity, we guess this is probably not peak time. The second and third value (11465.5 and 9381.4) is higher than our testing results. The error rate is %1.52, which is less than our test result but the weird thing is that it appears in the non-peak time.

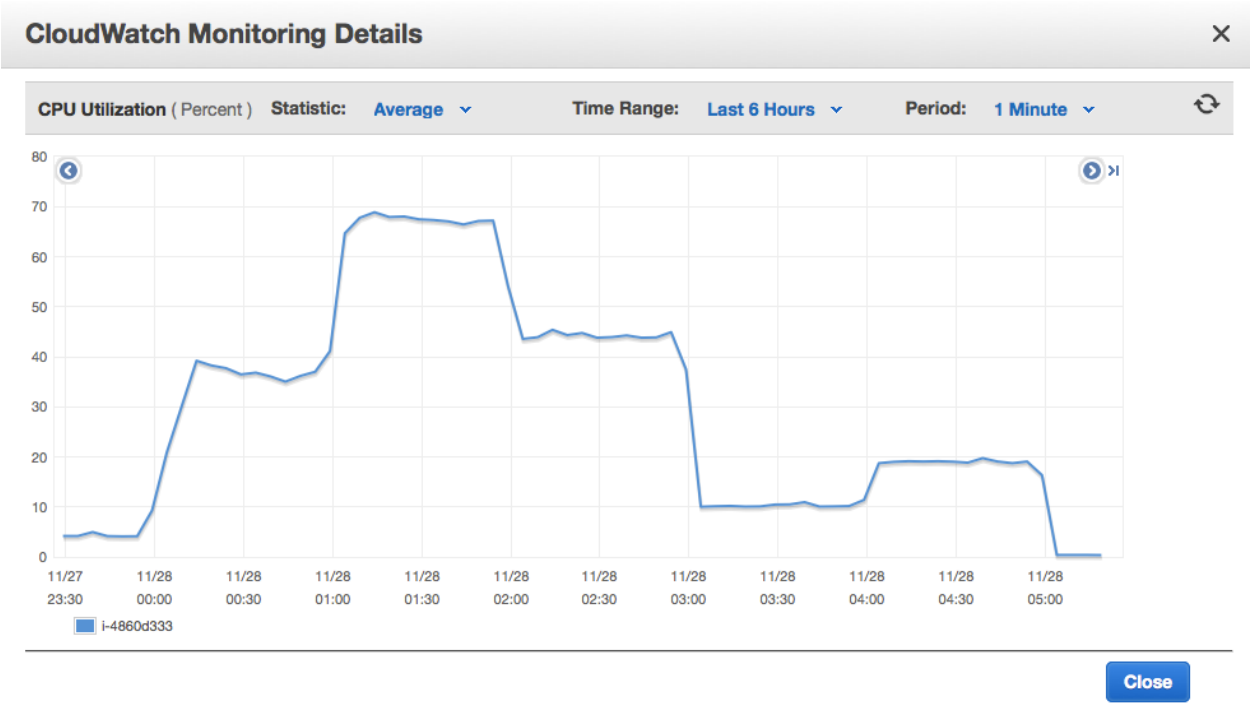
The performance of q2 is 3513 and 5890, the second value is roughly the same as our tests. The first one is a little below probably because non-peak time.

For q3, the results did not exceed 3000, which is roughly the same as our test results. For q4, one is 3479.2 (maybe non-peak time) and the other is 5947.7. The second one is roughly the same as our test result.

We suppose that each query there are peak-time value and non-peak time value, as a result, the actual result is basically the same as our expectation (except q1 is much higher).



For reference, the above is the CloudWatch of our ELB. The peak time comes roughly after more than one hours after the live test begins.



This is the CPU Utilization (each instance is roughly the same graph). As can be seen from the graph, even at peak time, the utilization did not exceed 70%, which is good because in this case there is no over-loading in peak time. This is because we use 9 m1.medium, the traffic is divided among them instead of focusing on a single or a smaller number of machine. If we use 4 m1.large instead, the CPU utilization is likely to be higher.

Analysis:

1) q1 is much higher than our expectation, we think this is because in live test, the test server will send much higher traffic than the online test service. In this situation, we have 9 m1.medium plus one ELB,

which is largely parallelize the traffic, reduce the load of each m1.medium, resulting in a higher throughput of q1.

2) For q2, we think the bottleneck is at MySQL, the peak performance has already been reached when we are doing test using online test service. Even when in live test the traffic could be larger, but MySQL can only handle about 5000 qps maximum.

3) For q3 and q4, the reasoning is the same. The bottleneck is in memory I/O, q3 is about 3000 qps and q4 is about 6000 qps maximum.

5 Review and Suggestions

Suggestion 1: The real live test's traffic is much larger than online test service. This surprises a lot of students including us. It is better provide same test service as will be used in real live test.

Suggestion 2: The amount of data is very small. The purpose of this class is to let us get a feel and know how to deal with "big data". So it would make more sense if even after the parsing, the data set will still be large (like 100GB, etc.).

Suggestion 3: The backend cost constraint is very tight if we use HBase, only 1 master and 2 cores. This is actually one factor that discourages us to use HBase.

Appendix

Source Code Structure

ETL/LoadHBase.java

This parses q2 data from S3 to HBase. Even in our final solution HBase is not used, this code is still useful because we can first load q2 data into HBase and then export to .csv file and then import this .csv file to MySQL

ETL/parser_json_csv.py

Parse JSON from S3 to extract q3 and q4 data.

back_end/

Our back_end/ directory is empty because our data is merged to front_end. When importing data into MySQL, we did not use any script, just import .csv file into MySQL.

front_end/air_grizzly/*

This is our experimental frontend server using Grizzly

front_end/air_tomcat/*

This is our frontend implementation. After comparing with Grizzly, we decided to use Tomcat.

utils/Record.java, utils/FormTable.java

These are used to generate q3 and q4 data to memory.

utils/ExternHelper.java, utils/Merge.java

These are used to process raw data (such as eliminate duplication, etc.)

utils/FrontEndASG.java

This is used to create ASG (not used as final solution).

utils/PreprocessHBaseCSV.java

This is used to process csv files into key-value pairs to be conveniently loaded into HBase, this is no longer needed since we did not use HBase in the end.

utils/RandomRequestTest.java

To test our optimized algorithm to store q3 and q4 data in memory.

utils/TestReadHBase.java

To test read HBase data from front end.

Note: in the project we have a lot of dependent JAR library files, they are too big to be placed in the submission, so we deleted them.