# Cloud Computing Project Report

# Choice of Front End

## Grizzly or Tomcat zhuolinl :

Grizzly is a light-weight server which can directly be launched by running java project with grizzly jar file. And it is commonly used in Junit test. However, Tomcat is more big 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 a m1.small instance and for Tomcat, we get the q1 qps for 1800. Since the performance does not have a too large 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, we use Jersey as our back package. We implement a listener to do some operations when initiating servlet, such as initiating database connection and parsing data file. Then for each query, we have a separate class file to handle the query and return the result.

## Load Balance Choice yinsuc (asg, not good, data using only q2:hbase)

1 code location

3. The type of instances used and justification

4. The number of instances used and justification

5. Other configuration parameters used

6. The cost per hour for the front end system

7. The total development cost of the front end system

8. The throughput of your front end for the given workload of q1 queries.

# Backend Choice

## Basic design (HBase) yinsuc (based on intuition data )

Assume we put all data in the backend and get the not so good result

1 The design of the back end system (why hbase)

2. The table structure of the database, justify your design decisions

3. The type of instances used and justification

4. The number of instances used and justification

5. The cost per hour for the back end system

6. The spot cost for all instances used

7. The total development cost of the back end system

# ETL method yinsuc

1 code location

2 The programming model used for the ETL job and justification

3. The type of instances used and justification

4. The number of instances used and justification

5. The spot cost for all instances used

6. The execution time for the entire ETL process

7. The overall cost of the ETL process

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

9. Discuss difficulties encountered

10. The size of the resulting database and reasoning

11. The time required to backup the database on S3

12. The size of S3 backup

# Optimize for Throughput and Latency

## Large or Small qianm (also change from hbase to mysql, data is not so large enough with high requirement of throughput)

data size: backend, we first put all our data in backend and find we actually don’t need so much data

1. The insight from Step 4 to influence optimizations

2. The optimizations utilized and justifications

3. Changes to the overall system design

## Large data choice (mysql qianm data)

code location

The design of the back end system

The table structure of the database, justify your design decisions

The type of instances used and justification

The number of instances used and justification

The cost per hour for the back end system

The spot cost for all instances used

The total development cost of the back end system

## Small data ETL zhuolinl

As what we have 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 will go on to the second round parsing to make file can be read fast. So, we read userid, retweet\_userid file line by line and form other file whose schema is userid, num\_of\_tweets, retweet\_list.

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

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

## Small data choice zhuolinl

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 zhuolinl

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.

## Optimization Result qianm

5. For the given workload

a. The maximum throughput of the optimized system for q1, q2, q3 & q4

b. The latency of your optimized system for q1, q2, q3 & q4

6. The cost per hour of your optimized system at low and high load

# System Test zhuolinl

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.

For instance view, we will launch 4

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.

3. For the given workload

a. The maximum throughput of your overall system for q1, q2, q3 & q4

b. The latency of your overall system for q1, q2, q3 & q4

4. The cost per hour of your system at low and high load

# Provision, Load and Prepare for Live Test yinsuc

1. The IP address of your system

2. Final configuration of each part (instance type and number)

3. Estimated per hour cost of the web service during the test period