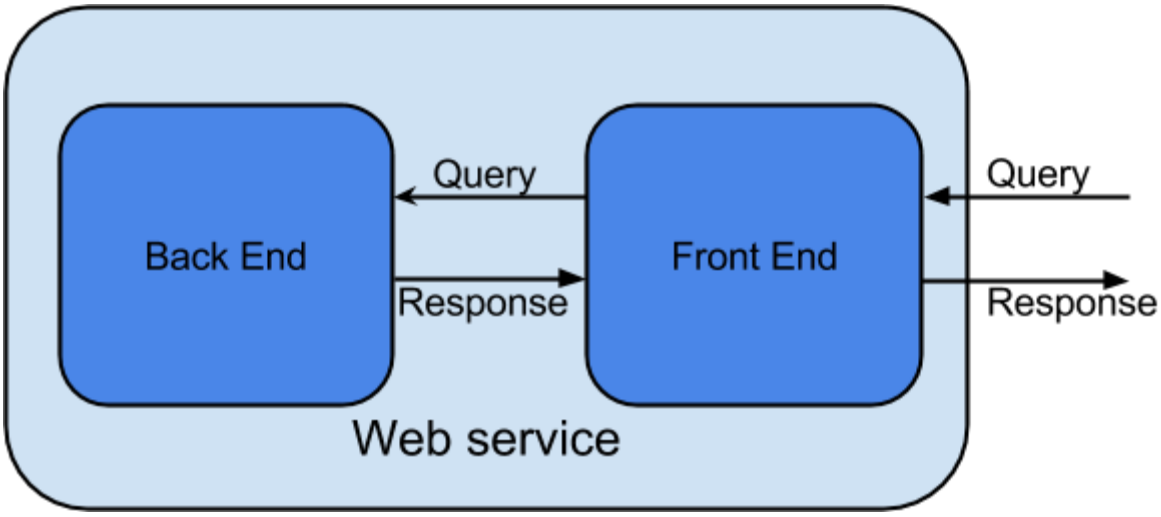
Twitter Analytics Web Service
Project Competition

Introduction
After several weeks of study in the Cloud Computing course, you have studied some of the theories behind various cloud technologies/services and have employed a variety of tools and services provided by the AWS Cloud.

Let us say that you form a cloud-based startup company with one or two colleagues from the cloud computing course.

A client has approached your company and several other companies to compete on a project to build a web service for Twitter data analysis. The client has provided a raw dump of tweets that runs into tens of gigabytes. The dataset will have to be stored within your web service. The web service should be able to handle a specific number of requests per second for several hours. The client has a limited budget for this solution. Your task is to build an effective and cost efficient solution utilizing Amazon Web Services resources.



Guideline:

1. **Front end** should be able to receive and respond to queries. The interface of your service is REST-based. Specifically, the service should handle incoming HTTP requests and provide suitable responses (as defined in the **Query Types** section below).
    a. Users access your service using an HTTP GET request through an endpoint URL. Different URLs are needed for each query type, which are shown in **Query Types** section. Query parameters are included within the URL string of the HTTP request.

   b. An appropriate response should be formulated for each type of query. The response format should be followed exactly otherwise your web service will not test properly with our load generator and testing system.
   c. The web service should run smoothly for the entire test period, which lasts several hours.
   d. The web service must not refuse queries.
2. **Back end** system is used to store the data to be queried.
   a. Millions of tweets are used as the dataset, and you definitely need a capable database to manage the data and handle the queries.
3. Your web service should meet the requirements for throughput and latency for queries for a provided workload.
4. The overall service (development and deployment test period) should cost under a specified budget. Less is generally better, otherwise the competition wins the contract.
5. Your client has a limited budget. So, you can **ONLY** use m1.small, m1.medium and m1.large instances to maintain your front end and back end systems. You can use spot instances for batch jobs and development but not for the deployed web service for the live test period.

Objectives:
The goal of this project is to practice integrating all parts that you have learned from the course in designing, developing and deploying a real working solution. You can use whatever services you have learned so far, configure and build them, make the system work as a solution, and finally optimize for performance and cost. We encourage you to discover and utilize whatever tools you find available to you. If the tools do not appear in the design constraints of this handout, you **MUST** discuss them with the professor or TAs before using them.

Dataset
The dataset collected by the client is available on **S3**. Each file contains roughly 20,000 tweets stored in JSON format.

Input and output
The web service solution should provide data statistics on the twitter dataset. Users can submit queries about tweets based on userids or time. The client has collected millions of tweets through Twitter's streaming API and have collected them as JSON formatted files, which is stored on S3.

The input is in a JSON format. Each line is a JSON object representing a tweet. Twitter's documentation has a good description of the data format for tweets and related entities in this format. https://dev.twitter.com/docs/platform-objects/tweets. Here are some examples of fields that are within the JSON object:

- **user**. A JSON object that contains the information of the user who posted this tweet.

There are many subfields inside this object. For this project we only need "**id**" or "**id_str**".
Example:
`"user":{ … "id_str":"6253282", …, "id":6253282, …}`

- **created_at.** A string**.**
  UTC time when this tweet was created.
  Example:
  `"created_at":"Mon Sep 30 11:10:50 +0000 2013"`
- **retweeted_status.** A JSON object for retweets. Users can amplify the broadcast of tweets authored by other users by retweeting. Retweets can be distinguished from typical tweets by the existence of a **retweeted_status** attribute. This attribute contains a representation of the original tweet that was retweeted. Note that retweets of retweets do not show representations of the intermediary retweet, but only the original tweet.
- **text.** A string.
  The actual UTF-8 text of the tweet.
  Example:
  `"text":"Tweet Button, Follow Button, and Web Intents javascript now support SSL http:\/\/t.co\/9fbA0oYy ^TS"`

Note: There might be some duplicate records in the JSON files. Please ignore duplicate records when you build your system. All records should be unique in your database.

Query types:
Your web service's front end will have to handle the following query types through HTTP GET requests on port 80:

1. **Heartbeat (q1)**
   The query asks about the state of the web service. The front end server responds with the project team id, AWS account id and the current timestamp. It is generally used as a heartbeat mechanism, but it could be abused here to test whether your front end system can handle varying loads.
   - REST format: GET /q1
     - Example: http://webservice_public_dns/q1
   - response format: TEAMID,AWS_ACCOUNT_ID yyyy-MM-dd HH:mm:ss
     - Example: WeCloud,1234-5678-9012 2013-10-02 00:00:00
   - Minimum throughput requirement: 2000 qps

2. **Text of tweets (q2)**
   The query asks for the tweets created at a specific second.
   - REST format: GET /q2?time=2013-10-02+00:00:00

    - Example: http://webservice_public_dns/q2?time=2013-10-02+00:00:00
- response format: TEAMID,AWS_ACCOUNT_ID
  tweetid1:tweet_text1
  tweetid2:tweet_text2
  ...
    - Example: WeCloud,1234-5678-9012
      99963645053419520 0:Happy New Year!
      99963645053421158 4:Merry Christmas!
      ...
- Minimum throughput requirement: 220 qps

3. **Number of tweets (q3)**
   The query asks for the total number of tweets sent by a range of userids. The 5-digit prefix of minimum and maximal userids should be the same.
   - request format: GET /q3?userid_min=133710000&userid_max=137713771
     - Example: http://webservice_public_dns/q3?userid_min=133710000&userid_max=137713771
   - response format: TEAMID,AWS_ACCOUNT_ID
     number_of_userid
     - Example: WeCloud,1234-5678-9012
       1337
   - Minimum throughput requirement: 0.7 qps

4. **Who retweeted a tweet (q4)**
   The query asks for the set of userids who have retweeted any tweet posted by a given userid.
   - request format: GET /q4?userid=133710000
     - Example: http://webservice_public_dns/q4?userid=133710000
   - response format: TEAMID,AWS_ACCOUNT_ID
     userid1
     userid2
     ...
     - Example: WeCloud,1234-5678-9012
       1000205192
       1117007780
       ...
   - Minimum throughput requirement: 250 qps

Load distribution:
During the test period of your solution, the workload will vary. The test will simulate the workload for a real world workload. For example, the number of users of a typical web service changes during a single day. The load generator that will test your web service will increase

the load gradually.

Minimum throughput requirements:
For a given workload, the web service should be able to achieve the minimum throughput requirements specified below.

Query	**q1**	**q2**	**q3**	**q4**
Throughput (qps)	2,000	220	0.7	250

Step 1: Front end
This step requires you to build the front end system of the web service. The front end should accept RESTful requests and send back responses. As the first step, the front end system is only required to handle the HeartBeat (q1) requests, which do not need to interact with the back end (database).

Design constraints:
- Execution model: you can use any web server such as Apache Server, etc.
- Cost budget: The front end configuration should not cost more than $0.2/hour at light load and no higher than $1/hour at maximum load.
- Spot instances are highly recommended during development period, otherwise it is very likely that you will exceed the budget for this step and fail the project. During the live test period, spot instances are not allowed.

Front end submission requirements:
1. The design of the front end system
2. The code to build front end system
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

Recommendation:
You might want to consider using auto-scaling because there could be fluctuations in the load over the test period. To test your front end system, use the Heartbeat query. It will be wise to ensure that your system can satisfy the minimum throughput requirement of heartbeat requests before you move forward. However, as you design the front end, make sure to account for the cost. Write an automatic script, or make a new AMI to configure the whole front end instance. It can help to rebuild the instance quickly, which may happen several times

in the building process. Please terminate your instances when not needed. Save time or your cost will increase and lead to failure.

Step 2: Back end

The step requires you to build the back end system. It should be able to store whatever data you need to satisfy the query requests. You are required to use spot instances for the back end system development. But you should use on-demand instances for the live test. Normally, the dataset storage system is the most crucial part of a web service. People cannot take the risk of losing any bit of data, so you are NOT ALLOWED to use spot instances for the back end system for the live test.

For the database design, you should take the four query types into consideration, which includes the requirement of throughput, latency and cost. First, you need to select what database to use. Your choices are limited between ***HBase*** or ***MySQL***. Then, you need to consider the design of the table structure for the database. The design of the table significantly affects the performance of a database.

Design constraints:

- Storage model and justification: you can use HBase or MySQL, explain your choice.
- Cost budget: The back end system should not cost more than $1/hour (on demand prices) and the development cost should not be higher than $5 total.
- Spot instances are highly recommended during the development period, otherwise it is very likely that you will exceed the budget and fail the project.

Back end submission requirements:

1. The design of the back end system
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

Recommendations:

Test the functionality of the database before the Twitter data is loaded into it. The test ensures that your database can correctly produce the response to the queries. Load a **small part of the data** as the first step. Think carefully about the query types when designing your schema. Test the initial performance capabilities of your back end in terms of throughput and latency.

Step 3: ETL

This step requires you to load the Twitter dataset into the database using the **extract**, **transform** and **load** (ETL) process in data warehousing. In the extract step, you will extract data from an outside source. For this project, the outside source is a large (around 100GB) Twitter dataset of JSON files stored on **S3**. The transform step applies a series of functions on the extracted data to realize the data needed in the target database. The extract step relies on the schema design of the target database. The load phase loads the data into the target database.

You will have to carefully design the ETL process using AWS resources. Considerations include the programming model used for the ETL job, the type and number of instances needed to execute the job. Given the above, you should be able to come up with an expected time to complete the job and hence an expected overall cost of the ETL job.

Once this step is completed, you **MUST** backup your database to save cost. If you use EMR, you can do the Hbase backup on S3 using the command:

```
./elastic-mapreduce --jobflow j-EMR_Job_Flow --hbase-backup --backup-dir
s3://myawsbucket/backups/j-EMR_Job_Flow
```

The backup will run as a hadoop mapreduce job and you can monitor it from both Amazon EMR debug tool or by accessing the http://jobtracker-ip:9100. To learn more about Hbase S3 backup, please refer to the following link http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-hbase-backup-restore.html

Design constraints:

- Programming model: you can use any programming model you see fit for this job.
- AWS resources: You must use SPOT instances for this step otherwise it is very likely that you will exceed the budget and fail the project.
- Cost budget: Your ETL job should not cost more than $1.5/hour.
- The total ETL step should not cost more than $30 in total.

ETL submission requirements:

1. The code for the ETL job
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

Recommendations: The ETL job runs for a long period. Make sure to test the correctness of your system using a small dataset and retest your backend. If your long running ETL job fails or produces wrong results data, you will be burning through your budget. The ETL job depends on you completing Step 2 (especially for the T and L in ETL). After the database is populated with data, it will be wise to test the throughput and latency of your back end system for different types of queries. Always test that your system produces correct query responses for all query types.

Step 4: System Test: Connect the Front End and Back End systems

At this point, all modules of your overall system should have been designed, developed and tested. Now comes the challenge of putting everything together as a single cloud service. At the end of this step you should make sure that your system works, you should know its load, throughput and latency capabilities as well as what might cause it to fail.

Design constraints:

Cost budget: You must not spend more than $1.2/hour during light load and $2/hour during maximum load. The overall system test should not cost more than $30 in total.

System Test submission requirements:

1. The design of the overall system and difficulties encountered
2. The AWS resources utilized to realize your system on AWS
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

Step 5: Optimize for Throughput and Latency

By this stage, your system is functionally operational and you have already tested its performance and understand its limitations. You might want to consider certain optimizations to performance while accounting for cost. Remember, you are competing against other companies to win the contract.

System Optimization submission requirements:

1. The insight from Step 4 to influence optimizations
2. The optimizations utilized and justifications
3. Changes to the overall system design
4. Changes to the overall system implementation
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

Step 6: Provision, Load and Prepare for Live Test

Prepare your system for deployment and evaluation by our automatic load generators. For this step to work, you have to submit your system's IP address, team name and AWS account used to the provided form on the project page.

Design constraints:

1. Use **on-demand** instances during the live test for all parts of your web service
2. Cost Constraints: Your system should not exceed with the following limits:
   Front End:
   $1 / hour
   Back End:
   $1 / hour

Live System submission requirements:

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

Notes:

1. Budget Constraints:
Here are the budget constraints for each step of the project. These constraints are based on on-demand prices. Each step has a per hour price and a per step budget.

	Solution Hourly Cost Constraints (Based on On-Demand Prices)	**Total Cost Constraints**
Step 1: Front End	$0.2 / hour (non peak period) $1 / hour (peak time)	$5
Step 2: Back End	$1 / Hour	$5
Step 3: ETL	$1.5 / Hour (including back end)	$30
Step 4: System Test	Front End: ● $0.2 / hour (non peak period) ● $1 / hour (peak time) Back End: ● $1 / Hour	$30
Step 5: Performance Optimization	Front End: ● $0.2 / hour (non peak period) ● $1 / hour (peak time) Back End: ● $1 / Hour	$30
Step 6: Live Test	Front End: ● $0.2 / hour (non peak period) ● $1 / hour (peak time) Back End: ● $1 / Hour	$20

There are two periods where cost will be evaluated, one for development and another for a live test towards the end of the project. The cost-per-hour Solution Cost Constraints are based on the **on-demand** prices which will be required for the live test period. During your entire development and exploration of configurations you are encouraged to use spot instances to limit your expenditure. Keep in mind that during the live test, the on-demand price will be utilized for the front end and back end. The ETL part will be a batch job that will not contribute to your web-service cost during the live test. Please use spot instances for ETL and test your ETL code using a small dataset to verify its correctness first.

The total cost constraints is a **hard limit**. Spending more than the constraints for any step will result in the heavy penalty of disqualification from the bidding process. This is serious so take it seriously and do not ask for exceptions, they will not be granted. If you forget instances running and exceed the budget, you and your team will fail the project.

2. Using Tags on AWS

The cost of your system will have to be evaluated. Tags enable you to categorize your AWS resources in different ways, for example, by the purpose the resource is being used. Each tag consists of a key and a value, both of which you define. For example, you could define a set of tags for your account's Amazon EC2 instances that helps you track each instance's project and project step. You should devise a tag key according to the table below. Using a tag, it will make it easier for us to evaluate which resource is being used for which project. You can search and filter the resources based on the tags you add. Hence, you are required to tag **every** resource you provision on AWS during the development and test periods using your team name and project step. Use the following tags with your resources for each part of the project:

Tag Key	Tag Value
`15619_Group`	The Name of your Project Group (eg. “1337_team”)
`15619_Step`	The step that your are currently performing in the Project (eg. 1, 2, 3, 4, 5 or 6)

You can read more about AWS Tags here:
http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/Using_Tags.html

We have supplied the AMI **ami-a74b12ce** for you to use. This will be necessary to use so that we can automatically gather the tags described above and logs it at our server so that we can grade your work. In order for you to be able to do this, you will have to grant read-only access for the script to your AWS EC2 Space. In order to do this:

1. Create an IAM Role in your account with EC2 read-only access
    a. In the AWS Management Console, click on **IAM**
    b. Once inside, Click on “**Roles**”
    c. Create a new role, name it “**tagReader**”
    d. When selecting the role type, under AWS Service Roles, select **“Amazon EC2”**
    e. When setting the permissions, select the policy template titled **“Amazon EC2 Read Only Access”**
    f. Once you have created the role, refresh the IAM roles listing and verify that the role has actually been created
2. Grant the IAM role for every Instance you create:
    a. From within the AWS Management Console, specify the “tagReader” role for the instance **before** provisioning
    b. For Autoscaling groups or Elastic MapReduce clusters, make sure that the IAM

role is applied to all your instances in the launch configuration or in the cluster specification

Remember to only use the resources in the design constraints listed above. For example, do not use extra large instances or a database system that is not allowed. Furthermore, the AWS resources utilized for the OLI modules should also be tagged so that we can differentiate cost between OLI modules and this project.

3. Team Dynamic

You are all performing this project as team members in a graduate course. With that said, we expect you to deal with the team dynamic as mature colleagues. We expect all members of a team to contribute equally. We will ask you at random times to inform us of your responsibilities in the team and accomplishments. Teams sometimes face difficulties, when your team does, deal with the difficulties in a mature fashion. Once the teams are formed, we expect you to work together until the completion of the live test period. This project mimics a real world scenario, take full advantage of the experience and learn.

4. Grading

This project is a competition among team members for the top grades. The performance and cost efficiency of your web service will put you above competing designs. Your team's inability to meet the functionality and our minimum performance requirements will lead to a failing grade. Having the best performance but going over the cost budget will lead to a failing grade for the entire team.

5. Active Design Document

This is an active design document, the table below will highlight bug fixes and changes made and their timestamp. It is the team's responsibility to review the document for updates regularly. Hint: things will change.

Date	Modified by	Modification
October 30, 2013	Majd Sakr	Final edit and release to teams.
October 31, 2013	Majd Sakr	Added the AMI ID to be used for this project.
November 16, 2013	Majd Sakr	Note about ignoring duplicate records in data set.

End of Document