|  |
| --- |
| Chicago: |
| Twitter Harvester and Analyzer |
|  |
| Description and analysis on the cloud based solution that harvest tweets from city of Chicago and the sentiment analysis done on the harvested tweets. The geotagging provided by twitter was used to identify the tweets from Chicago and sentiment analysis was done on the topics unique to Chicago. |
| ***Group 8:***  Felipe Cista (706972)  Nikki Vinayan (646621)  Hung, Te-Tu (662240)  Xin Huang (685269)  Rongzuo Liu (645452) |

Chicago:

# Introduction

A simple cloud based solution was developed using Nectar research cloud facilities. A simple architecture with multiple twitter harvester with centralized CouchDB database node was chosen to design the solution. We have included multiple replicators in case of failure of central node. This harvested data is analyzed with help of the built in Map Reduce capabilities of CouchDB.

Around 2.4 GB of data has been collected with help of our Twitter harvesters, this is exclusive of the data used for backup. This is a continuous process, the harvesters are continuously running and pushing data to the database.

The multiple views for the data analysis has been created from with CouchDB and the sentiment analyzer can be run over any provided view. The sentiment analyzer add the sentiment and sentiment score to the document present in the provided view.

A simple FLASK web application has been created to display the data analytic scenarios pertaining to Chicago city. We have identified and displayed five generic scenarios as well as two scenarios specific to Chicago city.

Error detection was supported by a real-time logging tool log.io. Log Io uses a simple stateless TCP API to receive the log messages. You can activate it for the different nodes or cores in the solution. Nginx and uWSGI is used as a solution to handle large number of requests or load from user end.

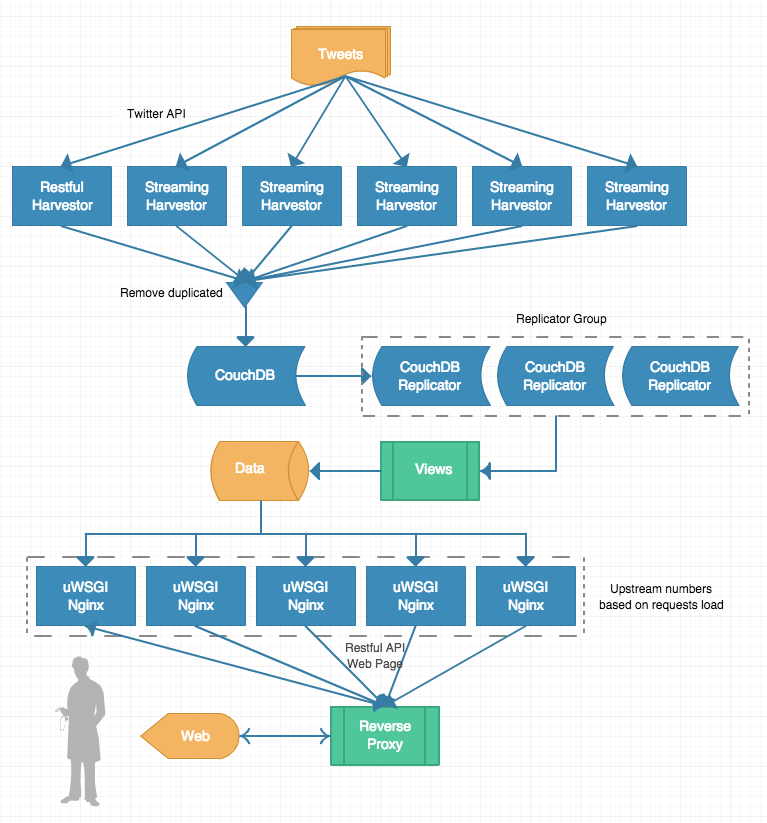
The system software is installed using Python Boto interface and Ansible.

The different component and architecture of the system is described below:

# System Components:

## Data Collector (crawler):

As a very large number of tweets is required to do analysis, two kinds of crawlers were created to access both REST and Streaming APIs provided by twitter. Both of them were used to fetch tweets continuously from twitter. The duplicates were recognized with help of twitter id and unique ones were added to the CouchDB application.

Architecture of global twitter harvester:

### REST API

In order to other tweets exclusively from Chicago, the requests to Twitter Search API were submitted with the geocode parameter. Unlike the streaming API which supports bounding box parameter, for limiting the location, we can only specify a central latitude and longitude and radius value. We provided the geocode value of Chicago with a radius of 15 miles, but there is a possibility we may cover areas outside of Chicago city.

To retrieve as many tweets a possible, our script recursively sends http requests with a result type and max id as parameters. The result type is set to the most recent, but in that scenario the API only retrieves the most recent results to the response received. In order to make sure we get the older tweets, the max id has to be set. Each time when script retrieves 100 tweets, the minimum id of the obtained tweets will be used as max id for next request .Thus we can make sure the crawler could go back and retrieve as many past tweets that the API can provide. After a few experiments, we found that the API can return tweets one week before the request day at the most. Once the crawler is not able to get the old tweets, the script will then fetch latest tweets, resets the max id and then repeat requests to make sure we do not miss any one of the latest responses.

There are limits enforced by the twitter API which does not let us harvest a large number of tweets. We handled two of the main limits set by Twitter. To resolve the ’15 minute window’, we only need to add adéquate interval between each request to not trigger this issue. Another limit we encountered is ‘Over Capacity’, Twitter API reduce the capacity or number of tweets that can be harvested by current requesting user. If we use one IP address and same app authentication for a long time, we get this faulty response. No new tweets will be obtained irrespective of how long you wait or how many times you request. In such situation, we had to restart the restful crawler on another virtual machine with new IP address, and a new twitter app had to be configured as well.

### Streaming APIs

The Streaming APIs gives developers low latency access to twitter’s global stream of Tweet data. Connecting to the streaming API requires keeping a persistent HTTP connection open. We use streaming API to maintain the completeness of tweets as it can retrieve much more tweets than the REST API.

Chicago city is divided into five bounding boxes based on its geographical area. These 5 different bounding boxes were used to create five apps to run on five different virtual machines. This locations parameter was provided to get correct tweets from specified area. However, according to our logging system, twitter seemed to push the same tweets to our five different harvesters.

On closer inspection we found that those tweets usually contains no coordinate information, but the user’s profile showing the location is Chicago. This might be the design of the Streaming API as a kind of compromise to provide tweet without accurate geo location using the same location as the user’s profile.

|  | **REST** | **Streaming** |
| --- | --- | --- |
| **Completeness** | No | Yes |
| **Accuracy** | 90% | < 50% |
| **Speed** | Low | High |
| **Persistent Connection** | No | Yes |

*Table1: compare rest and streaming*

## Web Services:

For data visualization, a simple web server was created with flask, which is a micro web framework written by Python. It includes html templates for browser viewing as well as RESTful APIs to access data. All charts or maps objects are drawn using JavaScript libraries which will do asynchronous requests to backend RESTful API.

All data for the visualization is retrieved from CouchDB map reduce views which were generated to get the desired data for various charts. For example, we add a sorting to CouchDB list functions. So when the data need to be ranked, we called the sorting list function from Couch app rather than fetch the entire data from specific view and sorting them locally on web server. This saves a lot of time for transmitting data. With large amounts of data, transmission load can be extremely heavy when there are lot of requests. Reducing this time can result in enhanced user experience.

To pick a suitable application server container, we used Vegeta which is an HTTP load testing tool to simulate 50 requests per second. The results are illustrated in the bar chart below:

From the tests it is obvious that pure python cannot handle concurrent requests very well, as the native multithreads cannot utilize all computing resources. A single uWSGI has much more better performance running this application than pure Python. Even though the combination of one Nginx and one uWSGI slightly slower than pure uWSGI, this situation will be changed once the load goes much higher and add more up streams to Nginx. Some articles have mentioned that the performance is better when Nginx running together with multiple uWSGI master processes, but we haven’t test that approach yet.

As shown in the system architecture diagram once the request arrives, the reverse proxy server will retrieve resources on behalf of a client from servers. These resources are then returned to the clients. On every instance, server application is started by uWSGI with 1 master process and 8 worker processes, to utilize all the processing cores of server. The local Nginx on each instance could be used as reverse proxy as well as http server handling other kind of requests, for example, static files, redirection and so forth.

The main reverse proxy server also plays the role of load balancer. It has the following three ways of handling loads:

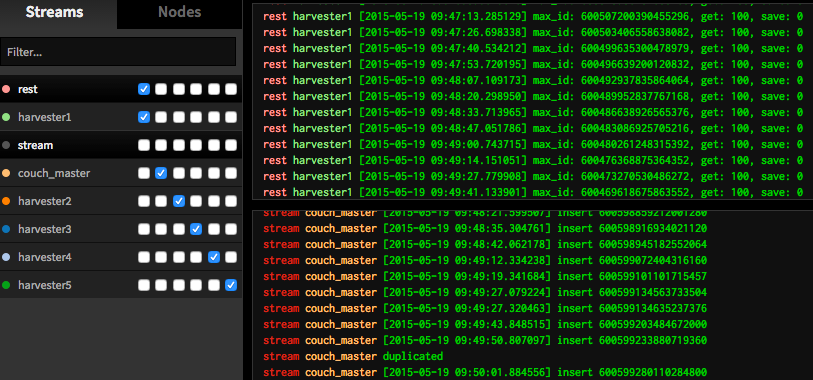
* + round-robin — requests to the application servers are distributed in a round-robin fashion
  + least-connected — next request is assigned to the server with the least number of active connections
  + ip-hash — a hash-function is used to determine what server should be selected for the next request (based on the client’s IP address)

This server model should enhance the performance very well and give a better fault tolerance.

## Error Handling:

### Real time log monitor system

The crawlers are deployed to six virtual machines on Nectar. Although we can use Boto and Ansible to secure the environment and for easy install, we need to monitor all the crawlers to make sure there are no blocking errors. However logging into each instance, checking the processes and the logs are a waste of extra time when automated solutions are available. We use log.io to monitor all instances via web interface. All logs on different instances will be streamed to the web browser, all six machines at one time. With log.io, we could identify if the crawler is running correctly or not.

.