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:

1. 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.

Chicago:

● ● ●
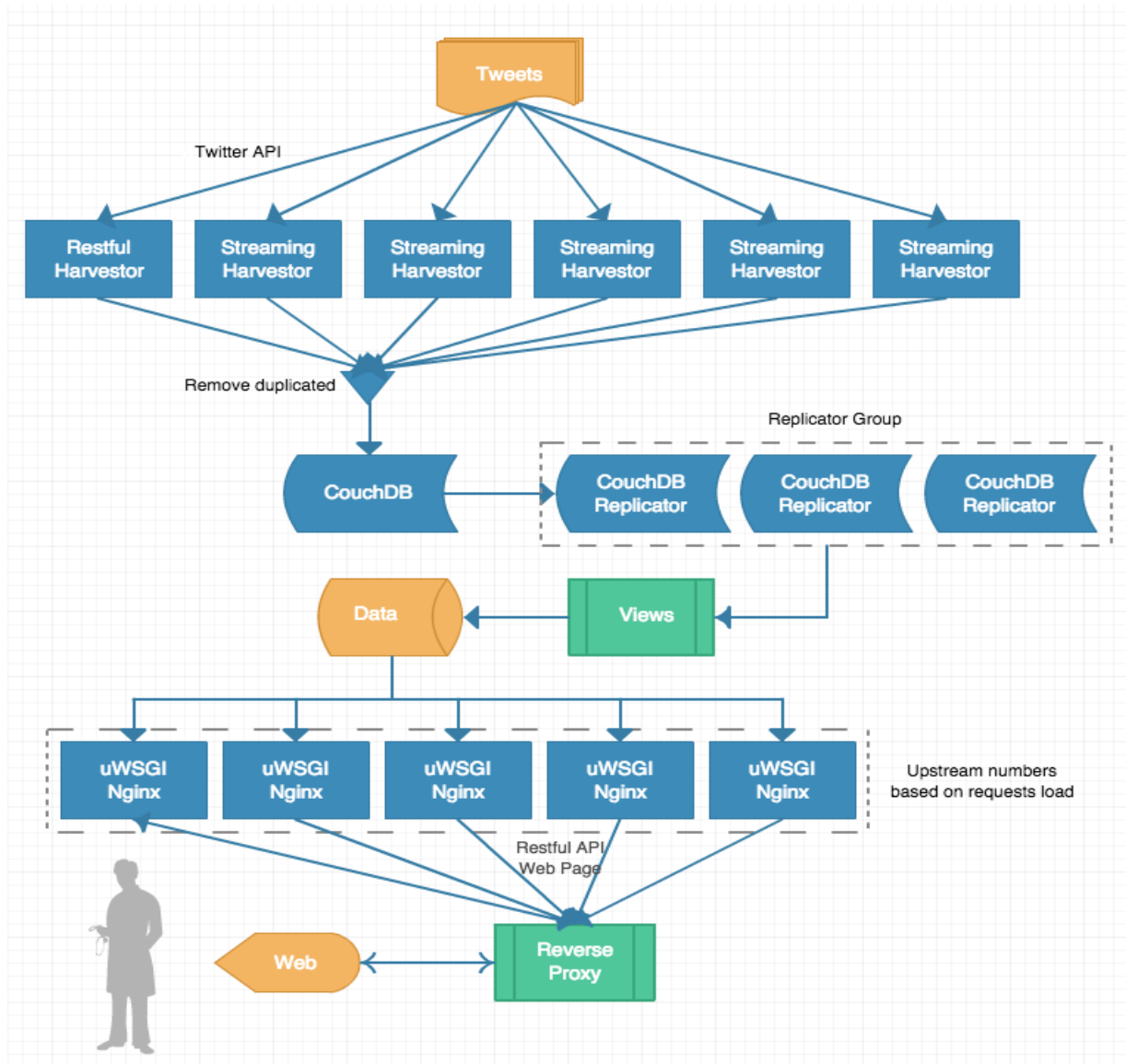
Architecture of global twitter harvester:

Figure 1Architecture

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

2. 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.

3. 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.
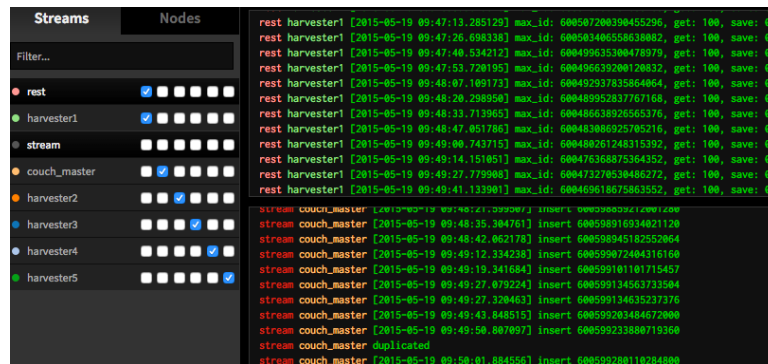
Figure 2: Logger

Removal of duplication

As twitter may return or respond with duplicated tweets to all the crawlers, we need an approach to identify and make sure no redundant tweets are saved to CouchDB. This is one of the key approaches for saving storage space, improving the speed of retrieval of data as well as ease of indexing when requesting views.

The solution for this is quite simple, each tweets from twitter API has a unique id and id_str. To make sure CouchDB does not lose precision of number we will use id_str to replace the build-in _id key. Each document in CouchDB has same definition of _id and twitter's id_str. So whenever a new tweet returns, we can easily identify whether it exists in CouchDB with extremely low time complexity, as _id could be used for searching and indexing.

Fault tolerance

The whole system has three layers of fault tolerance.

- Crawler level: Since we use both REST and Streaming APIs, if one node of Streaming crawler crashes or stops, the rest of the crawlers can still get those tweets from those area as REST crawler is

querying recursively. Since mechanism is designed for the REST crawler, no tweet is missed with both crawlers.

- Database level: Although we have only one CouchDB master server in this implementation, the crawler writes to two different databases. One for data processing and the other for data backup. Moreover, with setting continuous replication, those replicators also store copies of the data. Since there is a very small probability of all nodes dying, safety of data can be guaranteed.

- Web level: Nginx could be configured with multiple up streams. For example, if we have five instances running web services, the main reverse proxy server can have five different options to pass clients' requests. Still, to make sure the upstream is live, a health check module must be configured to poll status of all instances so that no request will be passed to a blocking instance. Also since the uWSGI launches several worker processes, the request could be respond to even if some of them are blocked. Generally if we have enough services providers, they can be calculated as [num_of_instances * num_of_workers * num_of_threads].

System Set Up

1. Ansible Set Up

Ansible is a management tool which provides the ways of creating scripts for automating the deployment and configuration of remote machines. There are two fundamental prerequisites for using Ansible: have an SSH connection with the remote servers which are being managed (the client machines) and, of course, have a server installation of Ansible running. According to its documentation [1], the exclusive usage of OpenSSH leads to a "strong focus on security and reliability".

One of the main advantages of using Ansible is definitely its simplicity, being very easy to convert bash scripts into Ansible scripts, and given its SSH use approach, the commands executed are also essentially the same. Although, Ansible provides a much more elegant way of achieving the final results by pulling together the context from all clients before executing tasks (this context gives relevant information about the current status of a client, e.g. if a program is already installed or not). Given the increasing complexity of a management script, all this information is necessary in order to accomplish a coherent outcome.

Ansible provides an inventory file called *hosts* and which is under the */etc/Ansible/* directory (in the case of our machines that have Ubuntu 14.04 running), which holds a list of remote manageable clients' IPs. These addresses may be grouped. Some other means for adding hosts are available, such as a dynamic way of populating the hosts file (this is not used in our work). The list may look as follow, with the group name in brackets:

Chicago:

● ● ●

[name]

>   192.168.1.2
>
>   192.168.1.3

The language used by Ansible to describe the tasks to be executed on remote systems is called Playbook. They are written in YAML format and deliver a very simple syntax, and should not be mistaken as a programming language, but rather a modelling language. A Playbook is composed essentially of a serial publication of tasks (or module calls) that are executed in a group of remote hosts in a scripted fashion. These tasks may be either synchronous or asynchronous.

IMPLEMENTATION:

The file AnsiblePlaybook.yml (made available through this project submission) contains different tasks to make a remote machine ready to run our application. Below, a brief description on the goals of each task:

1. Install git: A running git installation is required on the machines in order to clone our repository which contains all the application files.
2. Install python-setuptools: used to allow packaging and distribution of Python projects as well as project installation.
3. Install python-dev: for python developer package.
4. Install libffi-dev: provides foreign function interface, which allows code written in one language to call another code written in a different programming language (provided that a handler for conversion of values from both languages is implemented).
5. Install libssl-dev: SSL development libraries.
6. Install pip: Package management system that permits installation of software packages implemented in Python.
7. Set restrictive permission for the ssh private key: this task transfers the SSH key used for accessing the git repository to all remote machines; it also changes its permission as required by OpenSSH.
8. Clone Repository: clones the project repository where all the application files are stored.
9. Requirements: installs all the packages to which our application has dependency. They are: pycouchdb, tweepy, flask, CouchDB, TwitterAPI, whoosh and boto==2.34.0.
10. Requests [security]: install an HTTP library implemented in Python alongside its security extra packet.
11. Run Web-Server: this task executes a command on the remote machine in order to activate the web-server which display all the data resulted from our application execution (this task is asynchronous and the web-server runs for 1 hour, after when it shuts down).

EXECUTION

In order to execute the Ansible Playbook to deploy new remote machines, a series of steps must be performed. Alongside this project submission, a file called AnsiblePlaybook.yml is provided, in which

all the tasks are described (this playbook file must be stored under */home/ubuntu/* directory). The server in which these deployment tasks are based must have a functioning installation of Ansible and also must have an already configured SSH connection with all the remote client machines. To install Ansible, the following commands are employed:

sudo apt-add-repository -y ppa:ansible/ansible

> *sudo apt-get update*

> *sudo apt-get install -y ansible*

After having Ansible installed, is required to populate the inventory file with the clients IPs. This file is called *hosts* and is under */etc/ansible/*. In our project, this file looked like

> *[cloud]*

> *115.146.95.246*

> *115.146.93.141*

> *115.146.95.247*

> *115.146.95.54*

> *115.146.95.64*

where *[cloud]* is the group name.

After that, it is required that the file *id_rsa* (the SSH key which is used for accessing the remote repository) to be under */home/ubuntu/*. This file will be transferred to each remote machine so the cloning task can be executed.

Once all this is done, the Playbook file can be executed. To do so, the command *ansible-playbook -s AnsiblePlaybook.yml -vvvv* is used (the -vvvv argument runs it in verbose mode).

As the Ansible Playbook runs, a bunch of log information is printed. They compose the context of the remote client machine and gives you information about how the execution went. The most important is *{"changed": true}* or *{"changed": false}*: true means the modification on the remote machine was successful (it could a package installation, file transfer, etc); false means that no modification was made (because it was made already by either the Ansible Playbook or that machine user).

The last task, called *Run Web-Server* shows *<job 701862730123.5419> finished on <Remote Machine IP>*. This task executes a blocking command on the remote machine. Thus, to prevent the Playbook execution from hanging indefinitely, this task was made asynchronous (it returns as soon as the

command is executed). It will keep the Web-Server running for 1 hour; after that, the Web-Server is shut down.

2. CouchDB installation and configuration:

Opening port for external web server access

CouchDB is one kind of NoSQL database systems (DBS) for distributed system .The abilities of this database are aligned with project's requirements because of global twittering analysis and distribution-based architecture. Based on this conceptual idea, CouchDB can be implemented in this project and installed in the virtual machine by the command line tool.

Since the CouchDB cannot be accessed by external server the bind address has to be modified to 0.0.0.0 for client access. Regarding the CouchDB configuration file for modifications of bind address, there are two files, storing in etc. directory: local and default.ini files. Only local file needs to be configured .After that, the CouchDB has to be restarted for acquiring the new configuration so the CouchDB can be accessed by external connection. The diagram 3 illustrates the installation of the CouchDB.

```
ubuntu@project-vm4:/etc/couchdb$ sudo cat local.ini
; CouchDB Configuration Settings

; Custom settings should be made in this file. They will override settings
; in default.ini, but unlike changes made to default.ini, this file won't be
; overwritten on server upgrade.

[couchdb]
;max_document_size = 4294967296 ; bytes
uuid = 4c9e85e8dc8dc2213d663b0a0fa1d90f
database_dir = /mnt/data
view_index_dir = /mnt/data/views

[httpd]
;port = 5984
bind_address = 0.0.0.0
; Options for the MochiWeb HTTP server.
```

Figure 3 CouchDB Setup

Accessing CouchDB
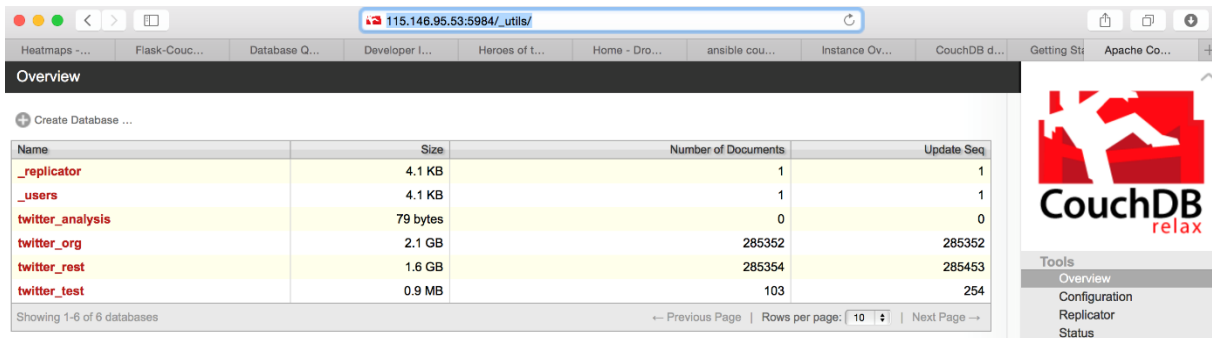
According to the REST approach, the request is communicated by HTTP of GET, POST, PUT, and DELETE [5]. The CouchDB supports REST protocol so that by executing "curl –X Get http://yourdatabaseip:5984/_all_dbs" to the command line we are able to view all database systems in the CouchDB. Another approach through which the database information can be acquired is using a web browser such as http://115.146.95.53:5984/_utils/. This shows the Futon webpage and we can retrieve all of database information and verify whether the CouchDB is working or not. The diagram 4 and 3 depict the process of gathering information from the CouchDB.

• • •

```
ubuntu@project-vm4:~$ curl -X GET http://localhost:5984/_all_dbs
["_replicator","_users","twitter_analysis","twitter_org","twitter_rest","twitter_test"]
```

Figure 4 A command script to check all of DBS in the CouchDB

Figure 5: Futon webpage for verifying all of Databases in the CouchDB

Attachment of volumes

The system also attaches a volume to the DBS because it may be increased incrementally due to inserting the tweet data on a daily basis for purposes of data and sentiment analysis. Based on the NeCTAR research cloud service, the volume can be created as well as attached to the target VM; however, this attachment has a limitation that it can only be from the same availability zone. The availability zone of the VM and volumes are created in the same zone because of that limitation. Once the volume has been attached to the VM, the volume need to be configured to connect to the location of the database files by command. By doing this implementation, the volume has been attached to VM finally. The diagram 6 shows a volume has been attached in CouchDB partition.


```
Disk /dev/vdc: 43.0 GB, 42953867264 bytes
16 heads, 63 sectors/track, 83228 cylinders, total 83894272 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000
```

Figure 6: The volume for the DBS partition

Configurations of the replication

Replication is essential for data backup as well as avoids disaster or loss of node. This disaster recovery in CouchDB can be implemented by web configuration as a continuous mode so that this recovery is

executed constantly. The tweet data is replicated to another CouchDB as a backup database. This plan provides an alternative approach that the CouchDB is crashed by unknown reasons and unable to recover. By doing this implementation, the users have an alternative database for performing the daily job. The diagram 7 depicts the status of this recovery system.

Last updated on	PID	Status
2015-05-18 14:42:50	<0.513.0>	Checkpointed source sequence 312225, current source sequence 312226, progress 100%

Figure 7: the status of the replication

NeCTAR research cloud

NeCTAR is based on the OpenStack™ is an open-source Infrastructure-as-a-Service (IaaS) platform. OpenStack was founded by joint efforts from Rackspace and NASA in 2010.

OpenStack is open of the fastest growing open source communities in the world with more than 18,000 individual contributors and 430 participating companies. The IT industry shows the widest uptake, and adoption is increasing in Academic/Research, Telecommunications, Finance, Media, and more. OpenStack is being widely adopted across many global regions, with an increasing number of OpenStack deployments moving from test/staging environments into production

One of main advantages of OpenStack is availability of a centralized dashboard through which we can manage the computing, storage and networking resources. The project mainly used python BOTO interface for the image creation and volume provisioning. Another advantage is the ability to utilize open source technologies, this allowed us to software which might has been unavailable to us in a closed proprietary platform like Azure.

The Dashboard allows for easy setup and creation of security groups and access to the application. However it is a bit complex for first time users unless they have encountered similar systems before. OpenStack also offers multiple other services such as Orchestration, Block Storage etc. which we have not made use of in our application.

The major issues we did face with Nectar was with its stability. We had instances of down time as well as networking errors while trying to provision new instances and volumes. This could be attributed to the fact that Nectar is used by a large number of research groups from across Australia and availability of resources per zone. This was compounded by the reboot required due to Venom, we could not access the master node containing the main CouchDB instance and the volume attached was missing for a few hours.

We had implemented another backup database as a failsafe, which helped to diminish any panic we felt during this issue.

The scenario of 1, 3, 4, 5 and 6 mainly use the tweets retrieved form Map/Reduce directly, while scenario 2 and 7 needs more processes on tweets. That is pro-processing and sentiment analysis.

Pro-Processing:

Pro-processing on tweets particularly means to parse the 'text' in tweets. Since natural language is significant complicated for a machine to understand and process, certain necessary pre-processing on 'text' are indispensable. In this project, this function is implemented in 'TextParser.py'. The 'TextParser.py' is responsible for the following tasks.

1. Delete meaningless words appearing in 'text'. The meaningless word means that this word is not useful and helpful to determine the sentiment of a 'text'. In this project, these words are named as 'stop words'. Stop words are various. For instance, the majority of them could be a pronoun (e.g. 'she', 'they'…), a preposition (e.g. 'about', 'to'…). Some of them could be an adverb, such as 'how', 'ever' and so on. In this project, all the stop words are collected manually and stored in a data file named 'stop_words.txt'.
2. A special scenario is also been considered that some people would like to repeat certain characters in a string to express their emphasizing. (E.g. instead of using 'bad', people might use 'baddddddddd' to address their feelings.) In this case, deleting those repeated characters is necessary.
3. Deleting mentioned tweeters (@), hashtags (#) and URLs in 'text'. Since the mentioned tweeters and hashtags show only very limited help in sentiment analysis, they are deleted in the implementation. Because basically, they are just nouns of topics or names which could be used to detect what this tweet is talking about. However, the most general way to obtain topic relevant tweets is by searching whether a certain term or string of this topic appears in the 'text' using Map/Reduce. Thus, these tweets are already associated with the topic, which makes the fields of mentioned tweeters and hashtags are no longer useful for sentiment analysis.
4. After the processes above, a new 'text' is generated and ready to be tagged its sentiment.

Sentiment Tagging:

The last step is tagging the sentiment field based on the analyzed 'text' of each tweet. TextBlob is used in this project as the tool to tag each tweet's sentiment. TextBlob is a Python library for processing textual data. It provides a simple API for diving into common natural language processing (NLP) tasks such as noun phrase extraction, sentiment analysis etc. [2] Both the build-in classifiers 'Pattern Analyzer' and 'Naïve Bayes Analyzer' are experimented in this project. Because 'Pattern Analyzer' shows more accuracy, it is chosen as the final classifier to do sentiment analysis.

Using this analysis method, tweets retrieved from the database will be add another filed named 'sentiment', with the values of its sentiment classification (positive/negative/neutral) and sentiment score (where [-1,0) represents negative, 0 represents neutral, and (0, 1] represents positive).

Source codes of pro-processing and sentiment tagging are under the directory of 'sentiment'.
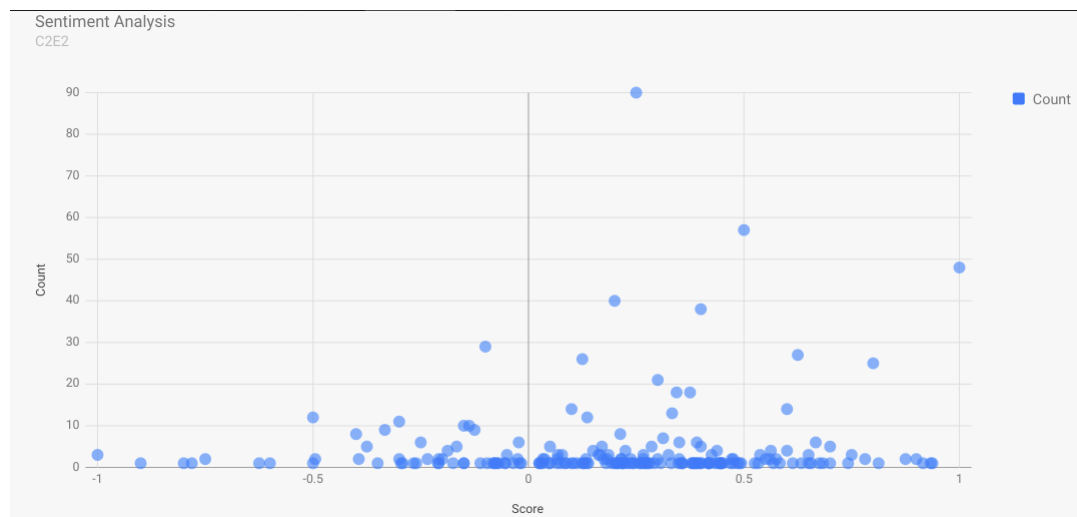
Chicago:

● ● ●

Scenario Analysis 1: Hot Topic

The hot topics are statistical hashtags in tweets that users mentioned most frequently. Figure 1 illustrates the most popular 10 topics in Chicago. It shows that three topics (They are Job, Jobs, and hiring respectively) are highly associated with finding jobs among the top five ones. This phenomenon could probably reflect a fact that the employment market in Chicago is still weak and certain amount of people are experiencing the difficulties to find a job.
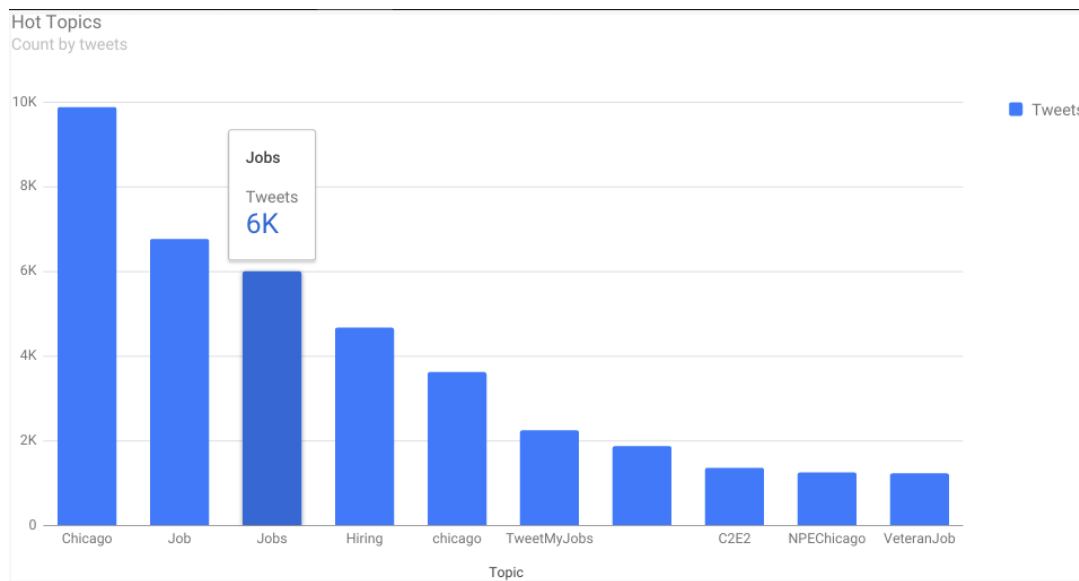
Scenario Analysis 2: Topic of C2E2

2E2 is an example of a specific topic chosen from the hot topics above. C2E2 is the abbreviation of Chicago Comic & Entertainment Expo. Figure 2 shows the sentiment distribution of tweeters regarding C2E2. Overall, it can be seen that people in Chicago hold more positive attitudes towards C2E2 than negative one, which expresses that they love Comic and the Comic-relevant things.

Based on the overall sentiment results, a further analysis is conducted to see what specific opinions hold by users to C2E2. This function is implemented in 'text_analysis.py' and the result is shown below. The result presents that lots of users have feelings such as 'thanks', 'love', 'great', 'best' to C2E2.

[(u'c2e2', 694), (u'chicago', 219), (u'mccormick', 189), (u'comic', 114), (u'table', 90), (u'entertainment', 67), (u'expo', 63), (u'panel', 60), (u'thanks', 59), (u'time', 54), (u'booth', 51), (u'love', 44), (u'move', 44), (u'great', 43), (u'best', 41), (u'convention', 38), (u'trooper', 35), (u'storm', 35), (u'clone', 34), (u'chicks', 34)]

Scenarios Analysis 3: Device Distribution

As the development of industry technology, devices used to access Internet are more and more various. In aspect of product popularity or other analysis, it is significant to detect what devices are used most by people. Figure 2 demonstrates the details of the distribution of devices people used to post tweets. As shown in the figure, iPhone is the most popular device with 43.7% users using it, following by android device that owns 20.9% users. An interesting phenomenon that only 3.3% Twitter users post tweets from the Twitter Web Client can be found from this figure. Compared with the percentages of either Twitter for iPhone or Twitter for Android, this number could be an evidence that mobile device is the overwhelming tendency in the future's IT industry development.
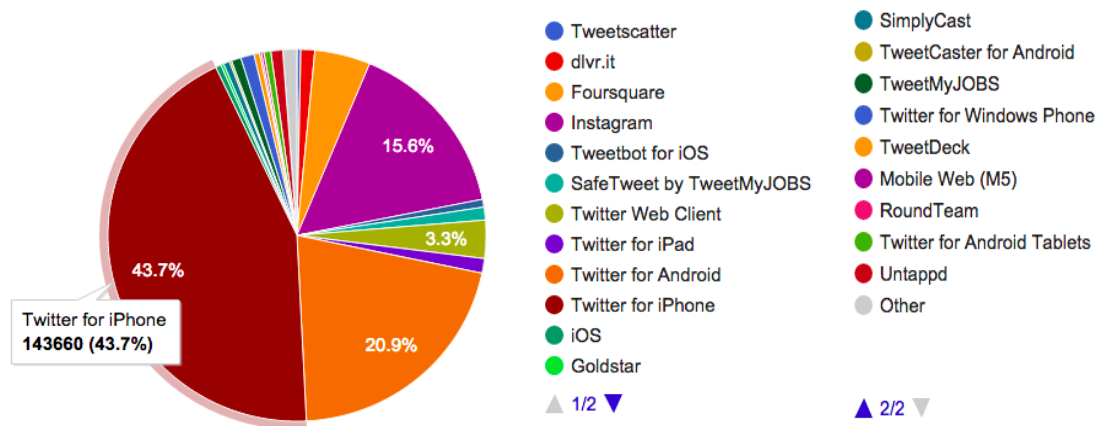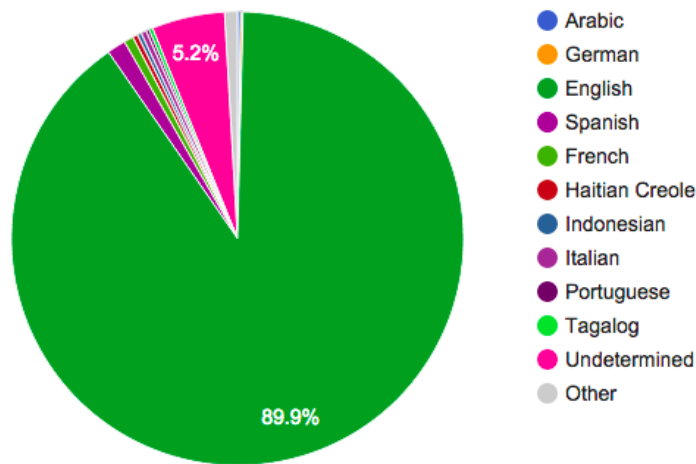

Scenario Analysis 4: Language Distribution

Since Chicago is a famous metropolis, people from around the globe bring their cultures here and throw them into this big pot. Thus, an analysis based on the language distribution is also conducted to see what kinds of language are daily used by people in Chicago. The figure shows the language distribution of those tweets. Instead of English, the most widely used language is Italian (1.3%), then followed by French (0.7%), which means lots of Italian and French are living in Chicago. However, there are 5.2% users cannot be determined using which language. The reason could lies in that these

users probably are Chinese, Japanese, Korean or other foreigners that their languages are dramatically different with English and therefore are difficult to be detected and recognized.

Scenario Analysis 5: Followers Distribution

According to this follower distribution, the MLB is the most famous person and has 4.5 million followers. The reason why MLB has heaps of followers is that MLB is a popular sport in US and this MLB account may periodically announce MLB latest news and highlight replays so that these information attract followers. The second person who has most follower is Steve Harvey. He is a comedian, author and celebrity. He usually uploads video and shares something fun with audiences so people love to review what he performs because of feeling happiness by these videos. The third one is Chicago Bulls which is the famous basketball team so it is a similar features regarding to MLB. The diagram XXX depicts the results of the people who have most followers in Chicago.
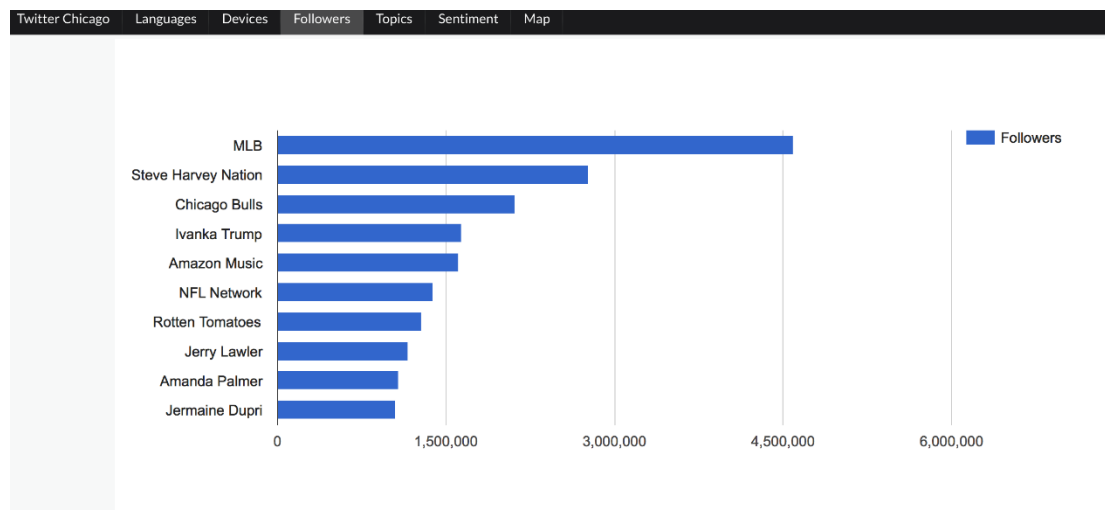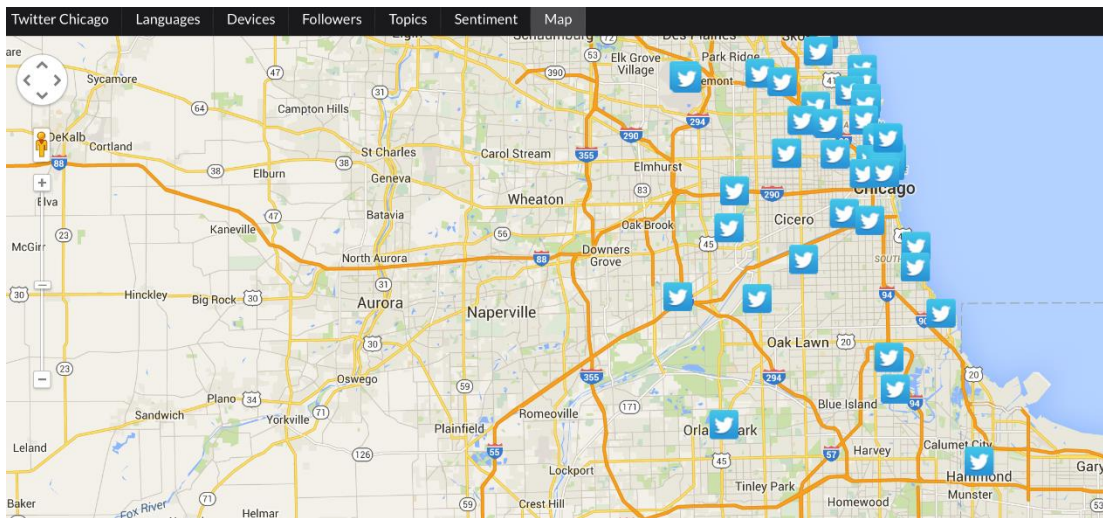
Scenario Analysis 6: Latest Tweets

The majority of tweet data are sent by the main areas in Chicago, representing urban people typical are more enjoyable to utilize social network app. In the urban area, perhaps, most of people take the train to the workplace so that they may have free time to surf the Internet during the transportation period as well as post the information by twitter. Urban lifestyle also is strongly connected with the social network, building the connection with each other. Based on this, this may enable urban people who frequently utilize twitter more than rural people. The diagram illustrates the result of locations of the latest tweet data.

.

## Scenario Analysis 7: Sentiment Timeline for Chicago Bulls

Chicago Bulls is an American professional basketball team based in Chicago. The NBA playoffs is currently going on in America and should expect a large number of tweets regarding their home basketball team by the people of Chicago. Unfortunately it was hard to distinguish the tweets from generic ones regarding pit bull dogs, red bull drinks etc. Hence we collected and analyzed the tweets regarding Chicago Bulls.

This time line traces the changes in sentiment over time. We had only around 300-400 tweets to do a sentiment analysis and this sample may not be true for the collective whole. We can compare this against the game schedule of Chicago Bulls and how they fared.

.

Change in sentiment for Chicago Bulls
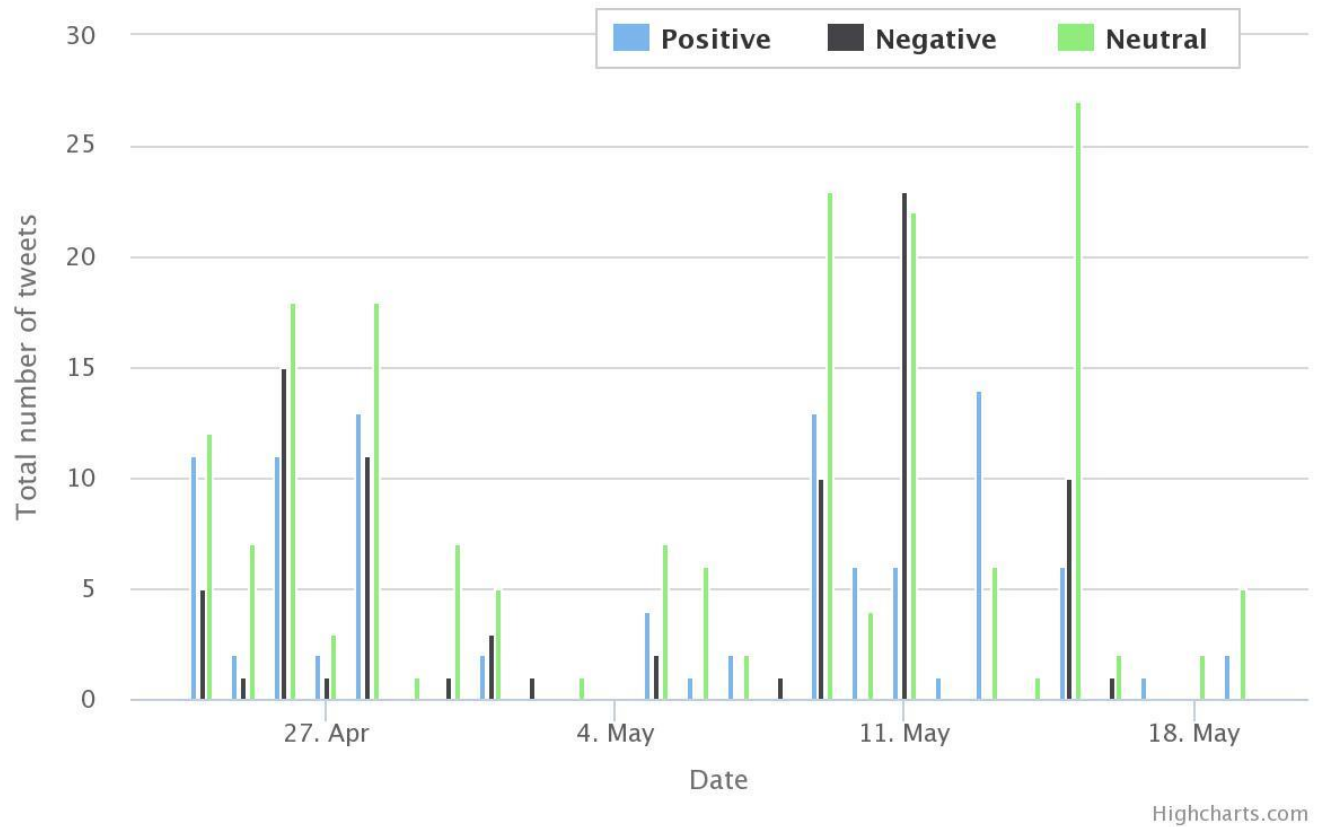
Schedule of Chicago Bulls Semifinals:

Date	Opposing Team	Win/Lost
April 23rd	Milwaukee Bucks	Win
April 25th	Milwaukee Bucks	Lost
April 27th	Milwaukee Bucks	Lost
Apr 30th	Milwaukee Bucks	Win

Schedule for Chicago Bulls Quarter Finals:

Date	Opposing Team	Win/Loss
May 4th	Cleveland Cavaliers	Win
May 6th	Cleveland Cavaliers	Lost
May 8th	Cleveland Cavaliers	Win
May 10th	Cleveland Cavaliers	Lost
May 12th	Cleveland Cavaliers	Lost
May 14th	Cleveland Cavaliers	Lost

• • •

Most of the tweets were neutral, however we can see a spike in the negative tweets. This could be attributed to losses faced by the Chicago bulls. The number of positive tweets peaks during April 23rd which is when Bulls won against Milwaukee Bucks, but gets more negative on April 26th as they lose.

We again see a spike in negative tweets after Bulls lost against Cleveland Cavaliers on May 10th. But it does seem they became neutral about their team as Chicago Bulls close this season with a loss on May 14th.

Conclusion

A stable fault tolerant cloud based twitter harvester was built and the harvested data was analyzed. Around 40,000 unique tweets from Chicago over a period of around 4weeks have been collected and analyzed.

References:

[1]http:/en.wikipedia.org/wiki/MapReduce

[2] http://textblob.readthedocs.org/en/dev/

[3] DZone RefCardz 208: Getting Started with OpenStack

[4] http://docs.ansible.com

[5] http://www.ibm.com/developerworks/library/ws-restful/

[6] https://dev.twitter.com/overview/documentation

[7] http://flask.pocoo.org/

[8] https://flask-restless.readthedocs.org/en/latest/

[9] https://uwsgi-docs.readthedocs.org/en/latest/

[10] http://nginx.org/en/