

COMP90024 Cluster and Cloud Computing
Assignment 2

Melbourne Tweets Analysis



Team 58

Zisu Geng - 989095

Wenbin Cao - 1008253

Haopeng Yan - 962332

Zhijun Chen - 1060606

Zemin Yu - 989980

Contents

1. Introduction.....	3
2. System Design in General.....	4
2.1 Nectar Cloud Resources	4
2.2 System Architecture	5
3. System Design in Detail	6
3.1 Automation	6
3.1.1 Introduction	6
3.1.2 How to Work	6
3.1.3 Playbook Structure	6
3.1.4 Specific Process.....	7
3.1.5 Apply Instance	7
3.1.6 Instance Variables	7
3.1.7 Render Template.....	8
3.1.8 Install Docker	9
3.1.9 Setup CouchDB Cluster	9
3.1.10 Docker Port Problem	10
3.1.11 Outcome	10
3.1.12 Setup Apache Server	10
3.2 Data Harvester	11
3.2.1 Tweet Object.....	11
3.2.2 Search and Stream	11
3.2.3 Remove Duplication.....	12
3.2.4 Maximum the limitation.....	12
3.2.5 Filter Geolocated Tweets.....	13
3.2.6 Suburb Name Normalization:	14
3.2.7 Final Decision	14
3.3 Data Analysis Pipeline.....	14
3.3.1 Query Terms (By WordNet)	14
3.3.2 Text Tokenization and Lemmatization.....	15
3.3.3 Sentiment Analysis	16
3.4 CouchDB Cluster Backend	17
3.4.1 Single Point Failure Discussion	17
3.4.2 CouchDB Cross-origin Access	17
3.4.3 Pre-defined MapReduce Views (Javascript format).....	18
3.4.4 Using Node.js Grunt to Upload Views	18
3.4.5 All Tasks in One Script.....	18
3.5 Visualization and Scenarios Discussion	19
3.5.1 The Front-end Design	19
3.5.2 Data Visualization	21
3.5.3 The Main Four Aspects of Data Analysis:	21
4. Conclusion.....	24
5. Tools and Packages References	25

1. Introduction

Twitter is popular social networking site allowing people to post short text messages, ranks the eighth from all social networking sites all over the world. Its API is well-built and free to use within some limitations, and by using that, a huge amounts of tweets data could be extracted and stored to do an analysis, about certain trends and phenomenon appearing in a city (in our cases, Melbourne will be focused).

We mainly aim on three different scenarios, which including twitters about food, twitters about shopping and twitters about works and jobs. These topics have close relations with people's daily life and so called "Seven Sins". Meanwhile, our system is easily to be extended to many other potential scenarios or research topics.

Our system is made up five parts:

System Component	Description	Responsible teammate
Cluster Configuration	The script package for one-click deployment of the whole system.	Zhijun Chen
Data Harvester	To search, stream, and filter geolocated data into CouchDB	Zisu Geng
Data Analysis Pipeline	Keyword and sentiment analysis	Wenbin Cao
CouchDB Cluster Backend	Configure CouchDB cluster, compile and push MapReduce designs.	Wenbin Cao
Web-based Visualization	Illustrate the finished data in the front web page.	Zemin Yu and Haopeng Yan

GitHub repository:

<https://github.com/AlanChaw/90024>

YouTube video of web-based frontend:

https://www.youtube.com/watch?v=BDCcR_rYeBE&feature=youtu.be

YouTube video of Ansible deployment:

<https://www.youtube.com/watch?v=Re2F-W4HsJc&feature=youtu.be>

Backup frontend server address (Only allows access from Unimelb):

<http://115.146.92.183/homepage.html>

2. System Design in General

2.1 Nectar Cloud Resources

For the allocated Nectar Cloud resources [1], we applied 4 instances with “m1.small” flavor, which has 1 virtual CPU and 4GB RAM resources for each instance. These resources are enough to be qualified for the tasks.

The Nectar Cloud instances can be applied by Ansible playbook, which makes them easy to manage and deploy as a cluster to work together (like a CouchDB cluster). However, objectively speaking, the Nectar Cloud is not stable enough comparing with the cloud platforms for commercial use. For example, sometimes it takes a long time to response the requests and there are some unpredictable failures happen.

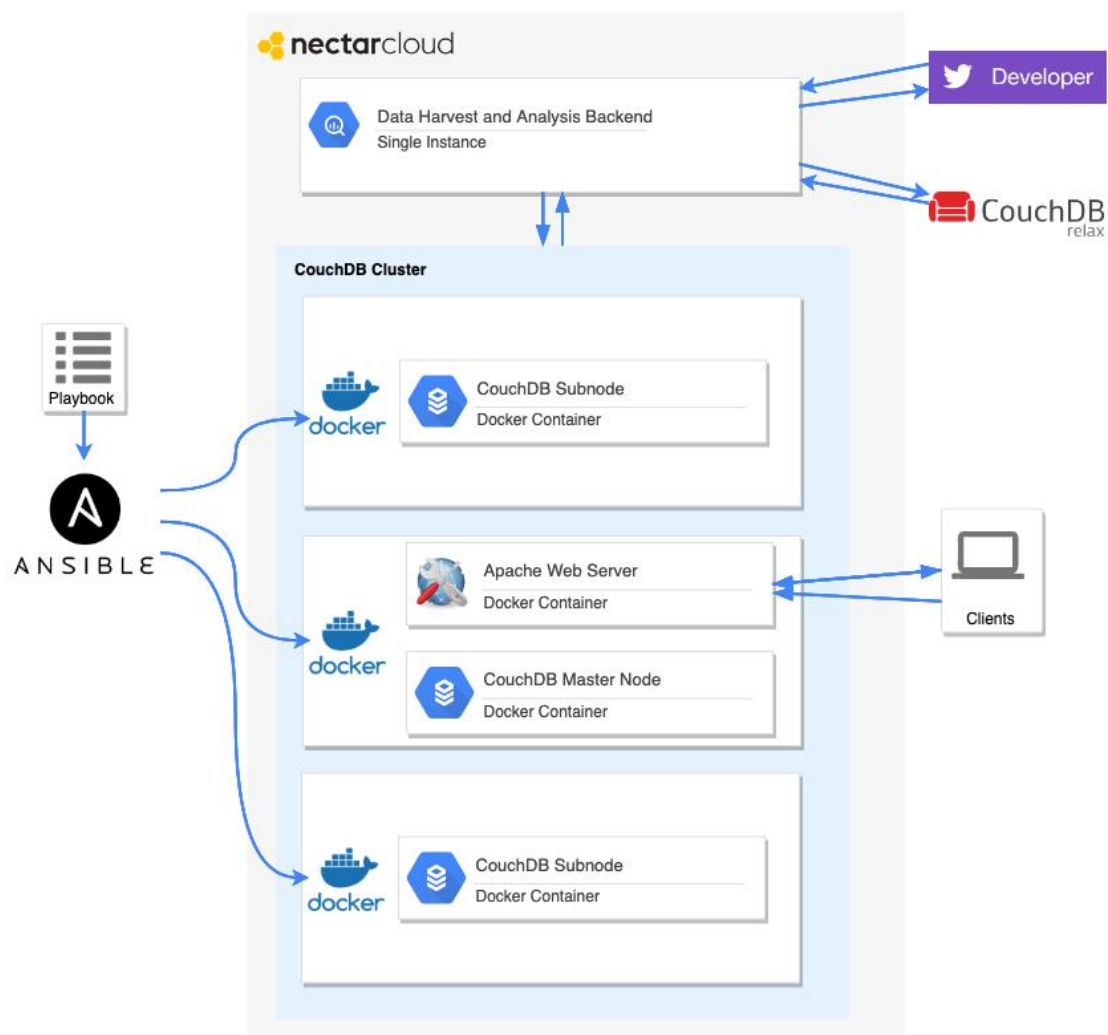


Figure 1: System architecture

2.2 System Architecture

Figure 1 displays our whole system architecture. Firstly, one single instance is applied for the **Data Harvest End** and **Data Analysis Pipeline**. The Data Harvest End gathers data from two sources: Twitter Developer API and existing CouchDB on IP address 45.133.232.90, which is provided in the task instructions. After the raw data are gathered, they are feed to the Data Analysis Pipeline to filter the twitters which are related to our scenarios and analysis the sentiment of the twitter texts. The analyzed data is stored at a stand-alone CouchDB on the same instance.

Secondly, the **Cluster Configuration End** is executed on a local machine. It automatically applies three instances on Nectar Cloud and deploys a ready-to-use CouchDB cluster by a single Ansible playbook script. All the CouchDB nodes are running on independent Docker Containers, which makes the instances more stable and easier to manage. Meanwhile, the **Web-based Visualization End** is also deployed on the same instance with the CouchDB master node and running on a Docker Container with Apache Web Server.

Finally, the analyzed twitter data is replicated from the previous single instance to the CouchDB cluster. Then, several CouchDB MapReduce views will be compiled and pushed to the CouchDB cluster by a script automatically. These pre-defined MapReduce views supply efficient queries to the database with RESTful APIs, these APIs can be requested directly by the frontend without any middleware.

3. System Design in Detail

3.1 Cluster Configuration (Ansible Playbook)

3.1.1 Introduction

In this assignment, we use Ansible [2] to develop the function of automation. With one script, we can apply the instances from Nectar, then setup a CouchDB cluster among the virtual machines and make the webpage ready to run (but the databases are empty at this moment, to populate them, we need to log into the master node server and run a bash script which is uploaded by Ansible).

3.1.2 How to Work

We just need to execute run.sh, then input the password of openstack API and localhost, then the script will automatically complete the tasks mentioned above.

Flexible username and password for CouchDB cluster are supported. If the user wants to use the customized user information, just modify them in the variables/variables.yml.

```
# couchdb user and password
username: admin
password: 123456
```

3.1.3 Playbook Structure

- |— Group58.pem
- |— ansible.cfg
- |— apply-instance.yml
- |— background-configuration.yml
- |— couchdb-cluster-setup.yml
- |— docker-configuration.yml
- |— environment-reset.yml
- |— inventory.ini
- |— openstack-api-password-Group58.txt
- |— roles/
- |— run.sh
- |— unimelb-comp90024-group-58-openrc.sh

└─ variables/
└─ web/

3.1.4 Specific Process

1. Apply Instance
2. Render Templates
3. Install Docker
4. Setup CouchDB Cluster
5. Setup Apache Server

3.1.5 Apply Instance

In this step, we reference the demo in Lecture 5, which shows how to use Ansible to apply one instance. We apply 3 instances in this assignment, set one instance as the master node of CouchDB cluster and the other two as the sub nodes. An exclusive instance (IP address is 115.146.92.183) has been applied in advance for harvest program, data collecting and visualization display.

3.1.6 Instance Variables

Image: NeCTAR Ubuntu 16.04 LTS (Xenial) amd64

Flavor: m1.small

Availability Zone: melbourne-qh2

Security: We open the port 22 for ssh connection, port 80 for http access, and port 5984 for CouchDB's operations.

<input type="checkbox"/>	Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Time since created	Actions
<input type="checkbox"/>	instance 2	NeCTAR Ubuntu 16.04 LTS (Xenial) amd64	115.146.84.198	m1.small	Group58	Active	melbourne-qh2	None	Running	22 hours, 24 minutes	Create Snapshot ▼
<input type="checkbox"/>	instance 1	NeCTAR Ubuntu 16.04 LTS (Xenial) amd64	115.146.84.106	m1.small	Group58	Active	melbourne-qh2	None	Running	22 hours, 25 minutes	Create Snapshot ▼
<input type="checkbox"/>	instance 0	NeCTAR Ubuntu 16.04 LTS (Xenial) amd64	115.146.84.163	m1.small	Group58	Active	melbourne-qh2	None	Running	22 hours, 26 minutes	Create Snapshot ▼
<input type="checkbox"/>	Preprocess	NeCTAR Ubuntu 16.04 LTS (Xenial) amd64 [v35]	115.146.92.183	m1.small	myKKKEY	Active	melbourne-np	None	Running	2 weeks, 1 day	Create Snapshot ▼

Displaying 4 items

We set a task for the instances to wait 240 seconds since when they have been created. The reason why we do this is to ensure the enough time before the port 22 of each instance has been opened. If the waiting time is too short, the ssh connection to the virtual machines may fail.

```
# wait for port 22 open
```

```
- name: wait for the port 22  
  become: yes  
  command: sleep 240
```

And we solve the problem that if we use Ansible to connect several instances by ssh for the first time, the prompts from several virtual machines to the user to input "yes/no" will conflict to each other, then fail the connection. We create an `ansible.cfg` file in the playbook directory, which can skip the prompt when connect instances via ssh.

```
[defaults]  
host_key_checking=False
```

3.1.7 Render Template

A very important step for the automation is that we need use some bash scripts to complete the operations. So we need put the IP address into these scripts. In addition, Ansible also need the IP address to tell which servers should complete the current tasks.

```
echo "== Set variables =="  
declare -a nodes=(115.146.84.163 115.146.84.106 115.146.84.198)  
export masternode=115.146.84.163  
export user=admin  
export password=123456
```

Since the floating IP address is allocated by the application of instances, we use some variables to record them and make some Jinja2 templates, after the applications of instance, the templates will be rendered as the real scripts, JavaScript codes and configuration files to relevant locations.

```
|— roles  
|   |— template-rendering  
|   |   |— tasks  
|   |   |   |— main.yml  
|   |   |— templates  
|   |       |— data-populate.sh.j2  
|   |       |— food_data.js.j2  
|   |       |— inventory.ini.j2  
|   |       |— job_data.js.j2  
|   |       |— masternode.sh.j2  
|   |       |— shopping_data.js.j2  
|   |       |— subnode1.sh.j2  
|   |       |— subnode2.sh.j2
```


3.1.8 Install Docker

We use Docker [3] as a supporting tool in this assignment. The most useful characteristics of Docker is that we can reset the all the dependent environment of the image by killing the container, which is very convenient for developing. With the help of Docker, we do not need to uninstall the software or delete the instance repeatedly if the environment has been "messy", and it allows us to run multiple services on one host and avoid the conflict between them.

In this step, we reference the demo in Lecture 5, which shows how to launch a WordPress service by Docker via Ansible. And for the later step, we add three tasks for every instance in this role: add docker users, pull the image of CouchDB, pull the image of Apache.

```
- name: adding docker users (for use without sudo)
```

```
  user:
```

```
    name: "ubuntu"
```

```
    append: yes
```

```
    groups: docker
```

```
  become: true
```

```
  with_items: ["ubuntu"]
```

```
- name: pull the image of couchdb 2.3.0
```

```
  become: yes
```

```
  command: docker pull couchdb:2.3.0
```

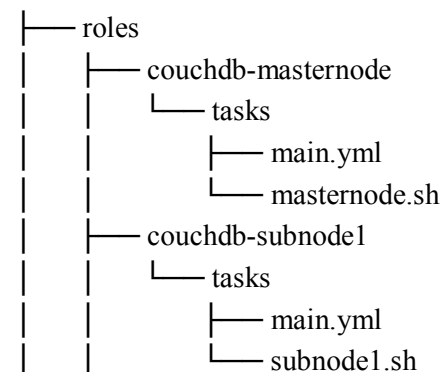
```
- name: pull the image of apache
```

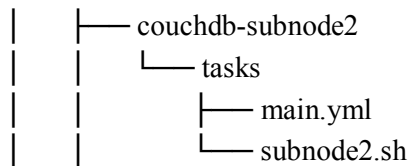
```
  become: yes
```

```
  command: docker pull httpd
```

3.1.9 Setup CouchDB Cluster

At first, we set up a demo of CouchDB cluster on two instances manually as a prototype, then extend it to 3 instances. At last, we try to use Ansible to complete the configuration of CouchDB cluster. Since we can use the bash scripts in the early steps, so our choice is to use Ansible to load the bash scripts to the database services, then execute them.





The order of the execution of these 3 scripts is described as follows: first we use Jinja2 templates to render the floating IP to the scripts and upload them to corresponding servers, then we start the scripts on the sub nodes. After that, the script on master node will be executed.

3.1.10 Docker Port Problem

The necessary port should be mapped to the corresponding port of the host machine. We get a bug that when the bash script which is uploaded by Ansible whose function is to run the master node of the CouchDB cluster is executed, it has a probability that the step to sign up the username and password may fail. This is step is the first step after the container of CouchDB is restarted. But if we log in the server and execute this script manually, this bug will not appear.

We notice that every time we get this bug, the node whose user and password can not be signed up is always the master node. So we think it may be the reason that the master node is the last node to be restarted, so it does not get enough time to set up the container environment which is necessary for the continuing operation. So we make the bash script sleep for 15 seconds to ensure the time for Docker container, then the bug is fixed.

3.1.11 Outcome

After finishing the configuration of the cluster, we can check the membership of the cluster via any IP of the instances. If the scripts are executed correctly, we can see that three nodes are in a same cluster.

```
ubuntu@instance0:~$ curl http://admin:123456@115.146.84.163:5984/_membership
{"all_nodes": ["couchdb@115.146.84.106", "couchdb@115.146.84.163", "couchdb@115.146.84.198"], "cluster_nodes": ["couchdb@115.146.84.106", "couchdb@115.146.84.163", "couchdb@115.146.84.198"]}
```

3.1.12 Setup Apache Server

We set the Apache server [4] as our web server to process the ask from front end. A Docker container is used for this background end configuration. The container is running on master node, and the characteristic of Docker promises that it will not conflict to other processes on the instance. Then we set an Ansible task to upload the frontend codes and resources (the reason why we do not use GitHub to download these code is that some JavaScript code depends on the floating IP, so we need to use Jinja2 to render it).

We map the port 80 to allow the HTTP access and map the web front end code to apache/htdocs which is the DocumentRoot of apache server.

```
- name: run the apache webserver
  become: yes
  command: docker run -d -p 80:80 -v
/home/ubuntu/web/visualization:/usr/local/apache2/htdocs --name webserver httpd
```

At last, we upload the data-populate.sh to the web server, then run it manually. The script's function is to solve the cross-domain access problem.

```
- name: upload a script to populate the data
  become: yes
  copy:
    src: ./data-populate.sh
    dest: /home/ubuntu/
    mode: 0777
```

3.2 Data Harvester

There are two ways to harvest tweets into are database: searching and streaming, by using the 'tweepy' API [5]. Searching is to get the tweets from the past, and streaming is to get the tweets in the future.

3.2.1 Tweet Object

When using Python to search and stream tweets by the tweepy API, each returned tweet object is a Python dictionary having an unique ID number. The newer a tweet is, the bigger the ID of the tweet is.

3.2.2 Search and Stream

Narrowing parameters: The search function of the tweepy API must have at least one specified narrowing parameter (keyword, location or others) in order to give some results. Since using 'keyword' to search is too arbitrary - an exact search which filters out too many results that may be related to our future analysis. We choose to specify the 'location' parameter.

Limitation: We can set the value for the 'count' parameter of the search function, and the function will return the minimum value between the value we give and 100, thus we can search a maximum amount of 100 tweets each time by calling the function. The returned result of the search function is a Python list of tweet dictionaries in chronological order where the first element is the most current tweet which has the

biggest ID number, down to about 100 tweets or when reaching the 10-day limit (we can only get tweets within previous 10 days). When reaching the 10-day limit, the search function will return the earliest element we can get in a list of length one.

3.2.3 Remove Duplication

Since the search function of the tweepy API returns the recent 100 tweets each time, we develop a method, in order to get 100 distinct and non-overlapping previous tweets consecutively (gradually tracing back by 100 tweets each time). This is achieved by specifying the 'max_id' parameter of the function.

1. When specifying the 'max_id' parameter of the search function, the api will search from the tweet having the ID (inclusively), back to about 100 previous tweets.
2. Firstly, we search once, and get the ID of the most current tweet (the biggest ID in our program), and store it in the variable 'new_max_ID'
3. Then, we repeatedly do the following operations until reaching the 10-day limit:
 - a) search from the 'new_max_ID' down to 100 previous tweets, which gives us a list.
 - b) Jump the first element of the list, deal with the rest tweets in the list.
 - c) set the value of the 'new_max_ID' to the least ID we get in the list (the ID of 'list[-1]') and use it as the max ID for the next call of the search function.
4. Since we jump the first tweet in the list each time of calling the search function, duplication could be prevented, and will be gradually tracing back without overlapping.

3.2.4 Maximum the limitation

Generally, people use access token authentication to create the API object:

```
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token_key, access_token_secret)
api = tweepy.API(auth, wait_on_rate_limit=True,
                  wait_on_rate_limit_notify=True)
```

This way of searching the past tweets has a limit of around 18000 tweets/15 minutes, per authenticated API object. However, if we do not set the access token, which changes the Access Token Authentication into application Authentication only, we can reach the limit of around 45000 tweets/15 minutes:

```
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
api = tweepy.API(auth, wait_on_rate_limit=True,
                  wait_on_rate_limit_notify=True)
```

However, this is not enough, because the time to search the 45000 tweets, filter the desired tweets that have coordinates or suburb name, and save the desired tweets into our database take about 350s. As a result, we have to wait about 550s in every 15 minutes(900 seconds). This is not efficient enough:

```
[144.9676666, -37.7980385, 'Carlton']
Rate limit reached. Sleeping for: 545
```

By solving this, we gathered three pairs of consumer key and consumer secret from our group. In each 15 minutes, we use the three pairs of the authentications one by one. Since 350s *3 is bigger than 900s, we can guarantee that the sleeping action will not be triggered when running our program:

```
# initialize apis
key_secret_pairs=[];
key_secret_pairs.append((key1,secret1))
key_secret_pairs.append((key2,secret2))
key_secret_pairs.append((key3,secret3))
api_s=[]
for key_secret_pair in key_secret_pairs:
    auth = tweepy.OAuthHandler(key_secret_pair[0],
                                key_secret_pair[1])
    api = tweepy.API(auth, wait_on_rate_limit=True,
                    wait_on_rate_limit_notify=True)
    api_s.append(api)
```

3.2.5 Filter Geolocated Tweets

There are two dictionary-keys describing a tweet's geolocation information: 'coordinates' and 'place', and both of the keys could have None or existing value. When the 'place' value is not None, the 'place_type' value of a tweet must fall into five categories: "poi" - location is a specific point (such as a scenic spot on the map), "neighborhood"-location is a specific suburb, "city" - location is a specific city, "admin" - location is a specific province, and "country" - location is a specific country. Since we aim to analyze tweets by suburb. We extract only the tweets having values for the dictionary-key 'coordinates' and the tweets whose key-value of 'place_type' equals 'neighborhood'.

3.2.6 Suburb Name Normalization:

Because we have a json file containing 309 suburb names in the Greater area of Melbourne with related boundary data, and our front-end web page uses the json file to do the data visualization, it is important to normalize the suburb name of an raw tweet that we get, into the range of the 309 suburb names.

When getting a tweet which has coordinates, we do not use google API to find the suburb from the coordinates, because it is too slow to return a result. Instead, we use the additional package 'Shapely', with the help of the json boundary file, to return the right suburb name for the coordinates, or 'None' if the point is not in any of the 309 suburbs.

When getting a tweet which has a suburb name, the suburb name may refer to a suburb in our boundary json file, but they have some tiny difference, such as getting 'Melbourne West' but it is 'Melbourne - West' in the boundary json file. In this case, we use the degree of similarity to normalize the raw suburb name into the 309 range.

3.2.7 Final Decision

We find the fact that we can get only the tweets within 10 days (about 800,000 tweets), only a few parts of them (about 5000) are in our need - containing coordinate information or suburb name, and Aurin has little recent data which is grouped by suburb. As a result, we decide to use the most recent data provided by the professor's CouchDB database - the tweets from January 2018 to Apr 2018 - about 10,000,000 tweets with the size of around 40g. After filtering the tweets having coordinates and suburb information, we end up with about 74,000 geolocated tweets for doing our analyzation.

3.3 Data Analysis Pipeline

The analysis end is deployed on the same single instance with the data harvest end. It is executed after the harvest end finishes its job.

After the raw data gathered by the data harvest end, a pipeline is implemented to tokenize, lemmatize the raw data and the related data is filtered by seed words. After the data processed by the pipeline, the results will be output to the local CouchDB [6] on the instance for further use.

3.3.1 Query Terms (By WordNet)

To filter data, which is related to the exact scenarios, we generated several query terms for each scenario. Only the twitters contain the query terms are believed related to the specific scenario.

To generate the query terms, firstly, a set of seed words are picked manually for each scenario (food, job, shopping). For instance, the seed words like “buy”, “purchase”, “consume”, are picked for the “shopping” scenario. The seed words like “employment”, “work”, “colleague”, are chosen for the “job” scenario. Secondly, the seed words are input to WordNet respectively to find more synonyms to expand the term set. Finally, all the seed words and their synonyms together constitute the query terms set for each scenario. WordNet is a well-used lexical database for English.

All the query terms can be generated by one single script. Before executing this script, the NLTK (Natural Language Toolkit) [7] for Python is required to be installed and the wordnet component must be downloaded. This can be done pip command. Then, run python3 in command line and open NLTK Downloader by:

```
$ pip3 install nltk
$ python3
```

```
>>> import nltk
>>> nltk.download()
```

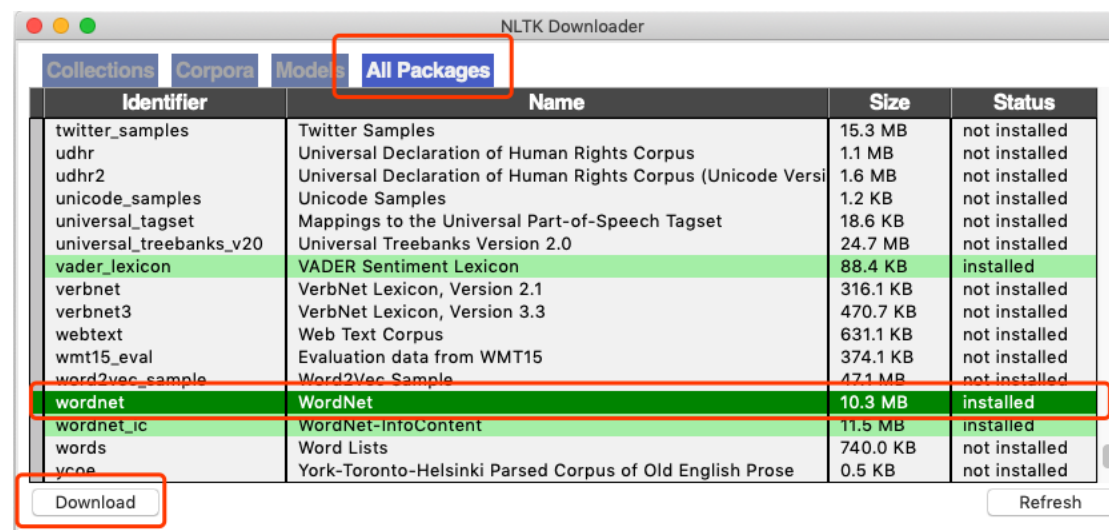


Figure 2: Download wordnet package for NLTK

After installed run the follow commands to generate the query terms for three scenarios.

```
$ cd ${Project_Path}/Analyze/FindWords
$ ./find_words.sh
```

3.3.2 Text Tokenization and Lemmatization

After the query terms created, the raw data goes through the pipeline. The first step is tokenizing the twitter text into words. Next, all the words are lower-cased and lemmatized to the original form, so as all the query terms. These two steps are both

implemented by NLTK built-in tokenizer and lemmatizer.

For the third step, the texts are filtered by query terms, only the texts contain the exact query terms are kept and others are discarded.

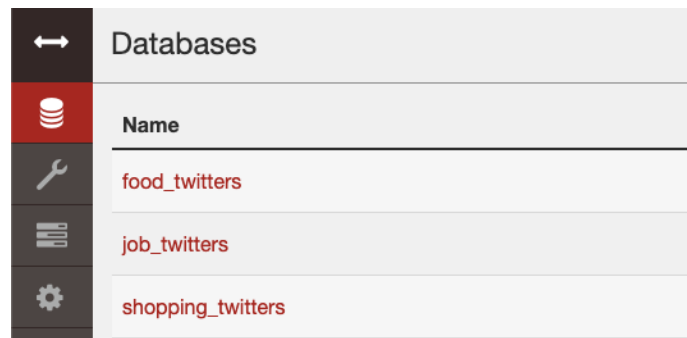
3.3.3 Sentiment Analysis

For the filtered twitters, the VADER (Valence Aware Dictionary for sEntiment Reasoning) sentiment analyzer in NLTK is employed to analysis the sentiment of the texts automatically. The texts are classified into “positive”, “negative”, and “neutral”, respectively and a tag will be added to the twitters. The processed data then will be stored in the local CouchDB.

All the data analysis steps can be executed by **one script**:

```
$ cd ${Project_Path}/Analyze/  
$ ./process_all.sh
```

After execution of this analysis script, three databases will be created on local CouchDB of the instance. The figure below displays the deserved result in CouchDB:



↔	Databases
🗄️	Name
🔧	food_twitters
📄	job_twitters
⚙️	shopping_twitters

Figure 3: Databases generated by analysis end.

The twitters generated by the pipeline have a format like:

```
1 {  
2   "_id": "71dca58df477703bc8151e824f01e69e",  
3   "_rev": "1-75ee2ce28d29bee2cfed4135e85f557e",  
4   "suburb": "St Kilda",  
5   "text": "snack stokehousetkilda on a saturday night be you eat off quality plat tonight stokey",  
6   "related": 1,  
7   "sentiment": "neu"  
8 }
```

Figure 4: Twitters generated by analysis pipeline

The “text” field is already lower-cased and lemmatized, the “related” tag is used to differentiate if the text is related to the specific scenario. In this example, “sentiment” field is “neu”, which represents that this piece of twitter has neutral sentiment.

3.4 CouchDB Cluster Backend

After the raw data goes through the analysis pipeline and stored at a single node CouchDB, we replicate the analyzed data to the CouchDB cluster, upload several pre-defined MapReduce views with Grunt (A JavaScript task runner), then the views with RESTful style API will be supplied by CouchDB and ready to be accessed by the frontend.

Since CouchDB supplies RESTful APIs to access the MapReduce views, there is no need to implement a stand-alone backend (like Flask or Django web backend). The APIs of CouchDB views will be directly accessed by the frontend and return the data which is going to display to the web frontend.

All these steps including replicate data to CouchDB cluster, open CouchDB Cross-origin access, upload MapReduce views will be executed by **one single script**.

The details of these steps are discussed below.

3.4.1 Single Point Failure Discussion

The analyzed twitter data on the single node is replicated to the CouchDB Cluster automatically by script. The CouchDB processes in the cluster are all deployed in Docker containers. Therefore, if one CouchDB node fails, on the one hand, it will not affect other processes in other Docker containers. On the other hand, the other CouchDB nodes will also not be affected by the single point failure.

3.4.2 CouchDB Cross-origin Access

CouchDB is required to be configured to allow Cross-origin access from other origins according to the CORS mechanism, which can be simply implemented by following commands:

```
curl -X PUT
http://${user}:${password}@${masternode}:5984/_node/couchdb@${masternode}/_config/httpd/
enable_cors -d "true"

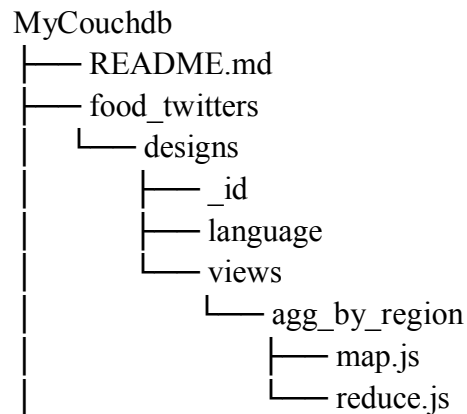
curl -X PUT
http://${user}:${password}@${masternode}:5984/_node/couchdb@${masternode}/_config/cors/
origins -d "*"

curl -X PUT
http://${user}:${password}@${masternode}:5984/_node/couchdb@${masternode}/_config/cors/
methods -d "GET, PUT, POST, HEAD, DELETE"
```

After these configurations, CouchDB will allowed the access from any arbitrary domain.

3.4.3 Pre-defined MapReduce Views (Javascript format)

The MapReduce query views are pre-defined as Javascript files in the paths similar as below in Background/MyCouchdb file path:



In this example, this “agg_by_region” view implements a MapReduce query across the cluster. This query calculates the counts of twitters with different sentiments and aggregated by regions in Melbourne.

The map function emits (Region name: 1) for each piece of twitter in the database. Then the reduce function gathers the data from map function and calculate the count of different sentiments for each region and return the aggregated results.

There is a question of MapReduce in CouchDB is that if the mapped data is huge, then these data will be stored as a B-Tree data structure, and just arbitrarily adopting the reduce function could cause failures. This problem can to be solved by implementing the “**reduce**” function and “**rereduce**” function respectively. The details of build the reduce functions will not be discussed here.

3.4.4 Using Node.js Grunt to Upload Views

The CouchDB MapReduce designs can be simply pushed to the databases by Grunt [8], which is a Javascript task runner. The configurations of Grunt are written in the Gruntfile.js in the Background path. The dependencies including Node.js and npm (Node Package Manager for Node.js) are required to be installed prior. These steps will be processed by our single “click-and-play” script and manual install is not required here.

3.4.5 All Tasks in One Script

The tasks of deploying this CouchDB cluster backend includes upload geometry JSON file, replicate pre-processed data from the analysis instance, open Cross-origin access for CouchDB, install and configure Grunt environment, compile and push MapReduce

views to CouchDB cluster. All these tasks can be executed by **one script**. Just run following command:

```
cd ${Project_path}/Background/  
./data-populate.sh
```

This script can also make sure the MapReduce designs are consistent with the remote version on GitHub. Therefore, when the designs are modified, we can simply run this script and the new designs will be updated.

After the execution of this script, several MapReduce views will be available to access. For example, by access the RESTful API:

[http://\\${instance_ip_address}:5984/food_twitter/design/designs/view/agg_by_region?group=true](http://${instance_ip_address}:5984/food_twitter/design/designs/view/agg_by_region?group=true)

The JSON results like following:

```
{  
  "key": {  
    "name": "Abbotsford"  
  },  
  "value": {  
    "positive": 98,  
    "negative": 21,  
    "neutral": 65,  
    "total": 184  
  }  
}
```

For each region in Melbourne, the counts of twitters with different sentiments will be shown in the JSON stream. These results are directly access by the frontend for data visualization tasks.

3.5 Visualization and Scenarios Discussion

3.5.1 The Front-end Design

There are two main components of our web: the homepage and the data display page. The core functions are described below.

1. Homepage:

This page shows the team members and there is a button to enter the data page. The main purpose of this page is not let users directly enter the data page, which make the website looks too stiff. Hence, we design this page to improve the user experience.

2. Data type page:

Entering this page will spend a few seconds to load, because a large amount of data will be loaded, so we designed a loading interface to make users feel more comfortable when using the page.

In order to compare the differences in twitter data between different regions, we used leaflet, which is an open source JavaScript library for interactive maps for the Web, to display geographic information on the map. Meanwhile, we use ajax to achieve asynchronously read json. The difficulty here is matching the homeless data with the corresponding regions. There are three different ways to display data on the map:

Firstly, the map directly shows the number of tweets in different regions, the depth of the color indicates the different rates of positive tweets.

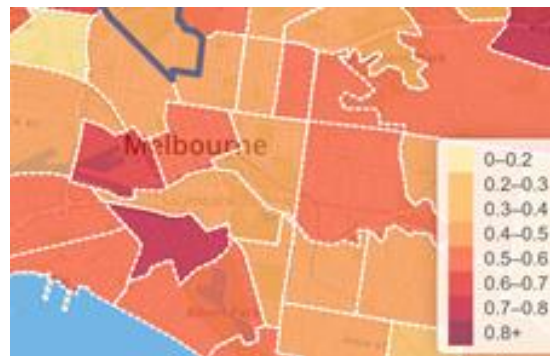


Figure 5: Rate of positive tweets

Secondly, when the mouse hovers over a region, the top right corner of the page will show the type of data, the name of region, total number of tweets, the number of positive tweets, the number of negative tweets and the number of neutral tweets sent by people in that region.



Figure 6: Details of the region data

Thirdly, when users click on the area which they want to know, a bar chart will appear that visually shows the number of positive, negative and neutral tweets, which make it easier for users to get the valid information from the chart.

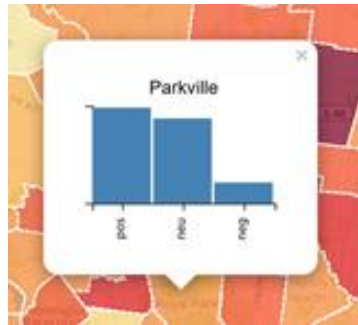
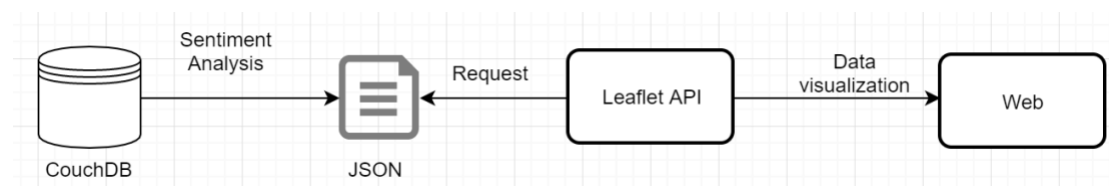


Figure 7: The bar chart of the number of tweets for suburbs

3.5.2 Data Visualization

Data visualization is an important part to display the data we have collected, processed and analyzed. In this project, we plan to use the rates of positive tweets on different topics in different areas of Melbourne to estimate the satisfaction of Melbourne residents with shopping, work and diet. Users can view relevant data in the front-end website. In the process of visualization, we retrieve the data stored in CouchDB, which is the json file generated after sentiment analysis. we used three pages of maps with different colors scheme (red, blue and violet) to visualize result of three different objects. The basic visualization process like below:



3.5.3 The Main Four Aspects of Data Analysis:

1. Overall situation

The number of tweets is basically in line with the population distribution in Melbourne. The number of tweets sent in Melbourne central is the largest, and the most of tweets coming from Melbourne central, south Melbourne, East Melbourne, Richmond, Southbank, Kensington, and Melbourne airport and other densely populated areas.

2. Food data

From the figure 4, the residents of Melbourne have high satisfaction for food services. Take the example of inner Melbourne, which has a higher total number of tweets, the number of positive tweets in most areas is around 50%, and the number of neutral tweets is almost half. Only a handful of twitter users have expressed dissatisfaction with food, like for instance, in Melbourne central, only 689 of 11380 tweets expressed

dissatisfaction. In the northeast Melbourne region, like Healesville - Yarra Glen, Yarra Valley and Wandin – Seville, Positive tweets about food reached more than 70%. To explain the data of food with deadly sins, a phenomenon can be found, people living in the city center are greedier about food than those living in the surrounding area. This may be because the quality of life in the city center is higher and there are more food choices, so residents have higher dietary requirements. Comparing with people living outside of the inner Melbourne, they are easy to satisfy with food, they don't have such many choices as the people living in the inner Melbourne.

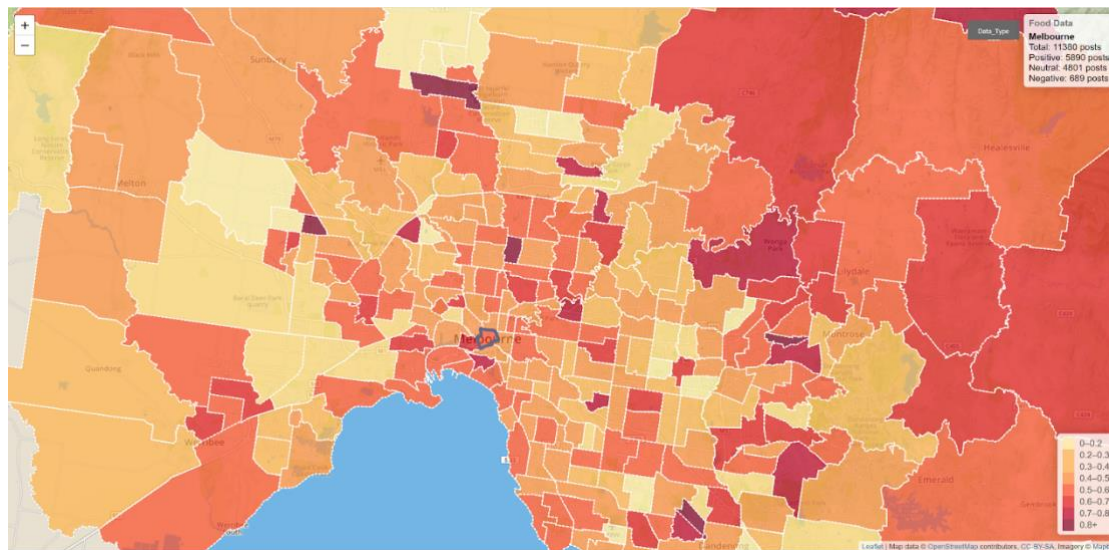


Figure 8: Food data

3. Shopping data

After analyzing tweets data leads to conclusion: Firstly, it can be concluded that the satisfaction level of shopping in the whole Melbourne region remains at 40% to 60% without some extreme data (such as some regions with only one tweet). This conclusion is better proved in the central area of Melbourne, where there is more data, since that there are more residents and more shops. Secondly, people living in the city center have a stronger desire to go to shopping. The areas with large shopping malls, such as the South Melbourne and Melbourne Center, has more number of positives posts, because the city can give more shopping choices, better service and experience.

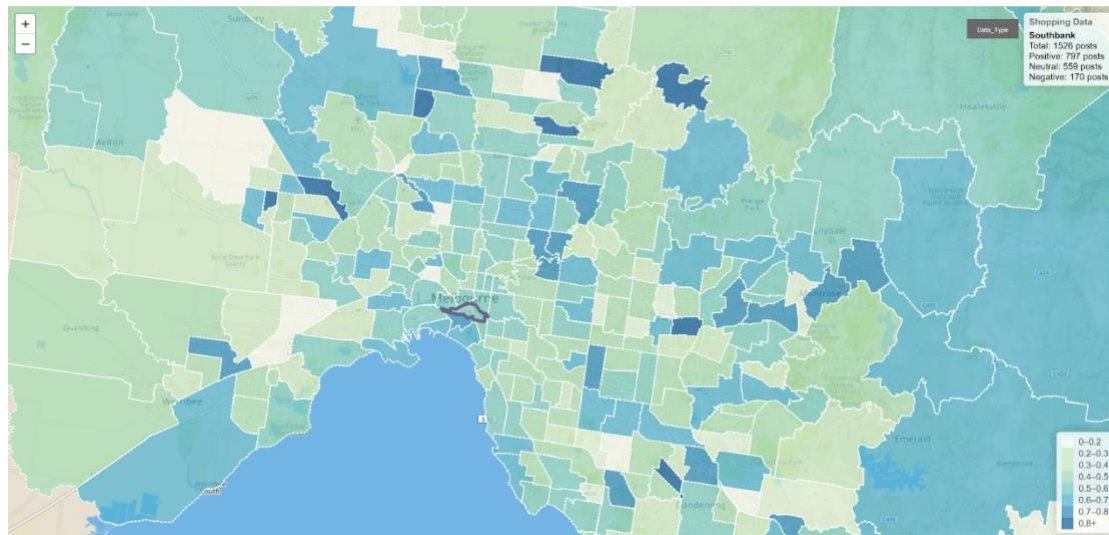


Figure 9: Shopping data

4. Job data

From the Figure 10, Job satisfaction in the central areas of Melbourne, such as Melbourne center and south bank, remains at the level of 40% to 50%. Meanwhile job satisfaction in the surrounding areas of Melbourne, such as Mount dandenong-olinda, is as high as 60% and even 80% in some areas. Although people in the inner Melbourne have higher income levels and life condition, but they have much more living cost and more stress of working for living in the inner Melbourne, so they would be less satisfied with their jobs comparing with the people living in the outside of Melbourne.

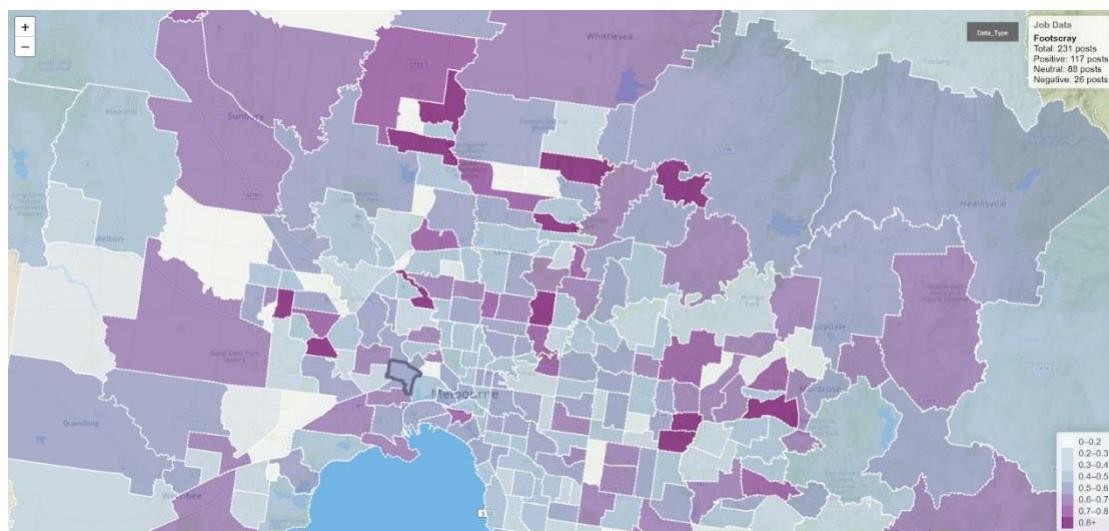


Figure 10: Job data

4. Conclusion

In this project, our team build a clustered system for analyzing the tweets in Melbourne area. Specifically, we chose the tweets related to food, shopping and jobs to do the analysis. The related tweets are filtered by several sets of key words. This makes this system is easy to extend by simply modifying these key words and this system can be deployed for research on many other scenarios and topics.

For our project, we build a three-node CouchDB cluster on three Nectar Cloud instances to provide fast MapReduce APIs to the frontend. However, the Data Harvest and Analysis part still run on single instance. We are planning transfer these two parts to a Spark cluster for more efficient data gathering and computes.

5. Tools and Packages References

- [1] "Home - Nectar Cloud", Nectar Cloud, 2019. [Online]. Available: <http://cloud.nectar.org.au/>.
- [2] R. Ansible, "Ansible is Simple IT Automation", Ansible.com, 2019. [Online]. Available: <https://www.ansible.com/>.
- [3] "Enterprise Application Container Platform | Docker", Docker, 2019. [Online]. Available: <https://www.docker.com/>.
- [4] D. Group, "Welcome! - The Apache HTTP Server Project", Httpd.apache.org, 2019. [Online]. Available: <https://httpd.apache.org/>.
- [5] "Twitter Developer Platform", Developer.twitter.com, 2019. [Online]. Available: <https://developer.twitter.com/>.
- [6] "Apache CouchDB", Couchdb.apache.org, 2019. [Online]. Available: <http://couchdb.apache.org/>.
- [7] "Natural Language Toolkit — NLTK 3.4.1 documentation", Nltk.org, 2019. [Online]. Available: <https://www.nltk.org/>.
- [8] "Grunt: The JavaScript Task Runner", Gruntjs.com, 2019. [Online]. Available: <https://gruntjs.com/>.