COMP90024
Cluster and Cloud Computing

Assignment 2 Report

Team 45

Name	ID	Email	City
Hong Ngoc Nguyen	1113959	hongngocn@student.unimelb.edu.au	Melbourne, Vic., Australia
Nhan Kiet To	1043668	nhankiett@student.unimelb.edu.au	Melbourne, Vic., Australia
Ting-Yu Lin	1100175	tllin1@student.unimelb.edu.au	Taipei, Taiwan
Mayank Sharma	936970	mayanks1@student.unimelb.edu.au	Melbourne, Vic., Australia
Duc Trung Nguyen	1069760	ductrungn@student.unimelb.edu.au	Melbourne, Vic., Australia

Table of Contents

1. Introduction

As social media has become extremely popular for the past decade, people tend to share their life and emotions on platforms such as Twitter and Facebook. As social media is getting popular over the years, people tend to share their experience and emotions on platforms such as Twitter and Facebook. This makes enormous data be generated on a daily basis. Resulting in a huge opportunity for researchers to discover different kinds of scenarios in real life.

Yet, many challenges arise due to the characteristics of big data, the amount of data to be processed is gigantic, on top of that, the data throughput is rather too fast while the variety of data is tremendous. There are three Vs properties of Big Data: Volume, Velocity, Variance (Fong et al., 2015), making conventional computing methods to handle them inefficiently. Hence, a Cloud-based solution to deal with big data is proposed in this report.

We utilized the Melbourne Research Cloud as the deployment platform to build our cloud-based system. The data from Twitter were harvested into our database. The real-time sentimental analysis and data visualization is implemented in our system as well. This helping us to have a better understanding of human behaviors such as emotion dynamics, mobility during the COVID-19 pandemic via our application.

In the following sections, we are going to introduce our system architecture including harvester, cloud-based system deployment, and web application in section 2. In section 3, we provide a user deployment guide to make it easier for users to deploy our system by themselves. In section 4, the data scenarios and results will be elaborated. And in the last section, we will discuss the advantages and disadvantages to use the University of Melbourne Research Cloud and how we handle the error.

2. System Architecture and Description

2.1. System Architecture Overview

As we utilize the portion of storage and computing power of our group in Spartan HPC, Ansible is being used to deploy and control our virtual machines (VMs) and Docker Image of the system, these are the deployment platform of our system.

First, we gather the tweets from Twitter using python library, tweepy, to access twitter API, there are 8 access tokens in use for this solution, the arrival of the tweets, they will be preprocessing with VADER Lexicon (Hutto et al., 2020), a sentiment analysis module from nltk library (Bird et al., 2009).

Then the data is stored in CouchDB with JSON format, the Back-end consists of Django-REST framework in Python3 with a small SQLite server for storing setting/config, the RESTful service is used to retrieve the data from CouchDB and push it to the Front-end which is handled by ReactJS, from here the data will be visualized into the charts and maps

and facilitate us to have a better understanding of human behavior such as emotion dynamic and movement.

The system works continuously in real-time, new tweets upon collecting can run through the whole flow until it or its results are visualized. Thus, the data in the Front-end is real-time updated. In the following paragraphs, we will introduce (1) Twitter Harvester (2) Cloud-base System Deployment, and (3) Application (Databases, Frontend, and Backend).

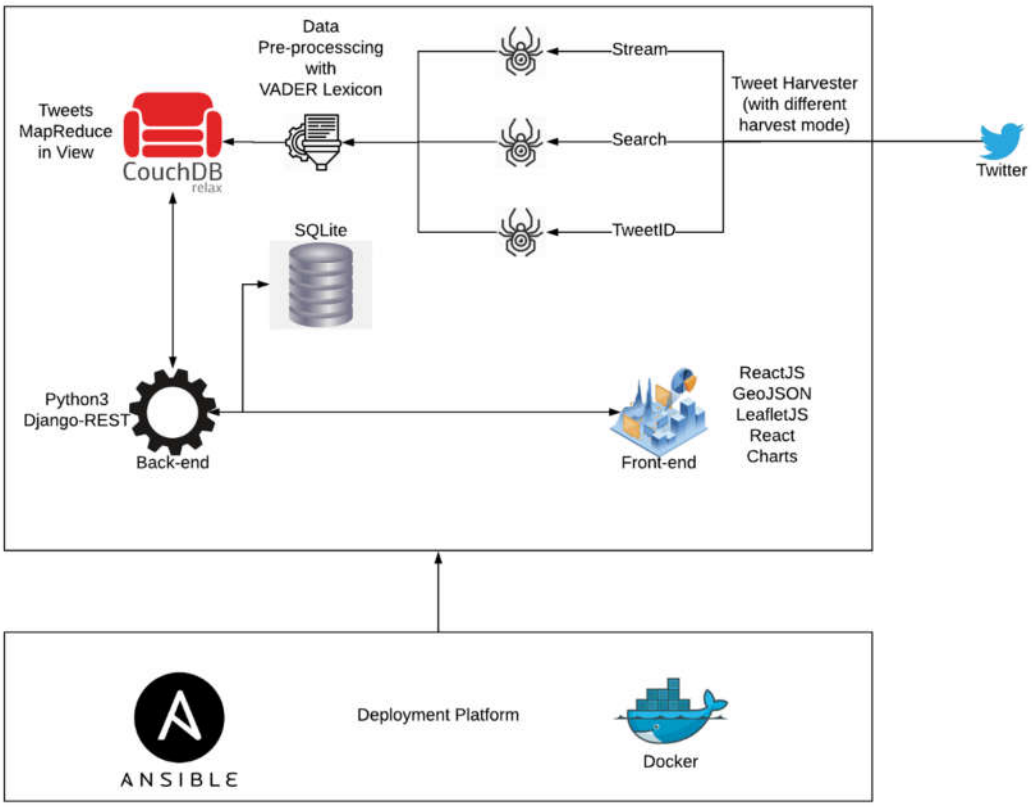

Figure 1 System architecture

2.2. Twitter Harvester

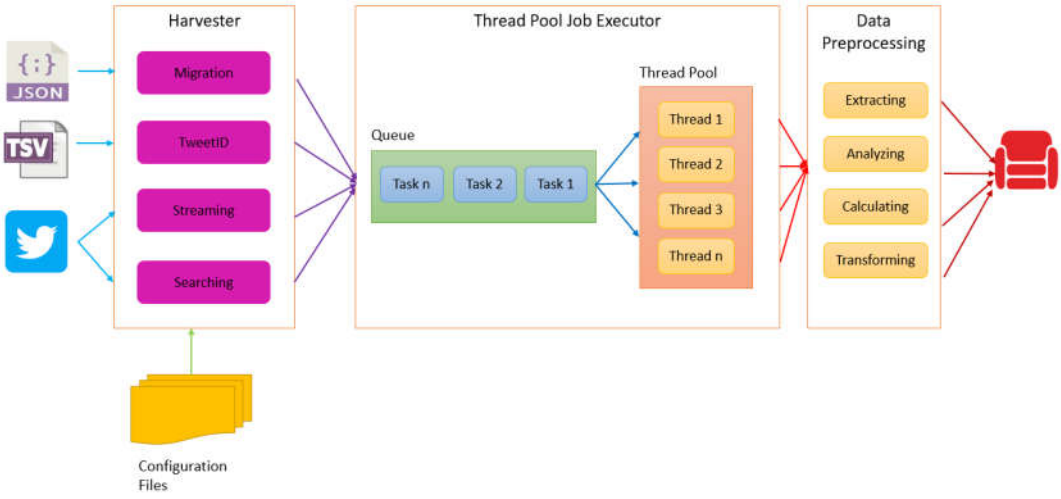

Figure 2 The harvester architecture

The twitter harvester component is responsible for collecting tweets in various sources, performing data pre-processing, and storing proceed data to the Apache CouchDB database. It supports the following harvest modes. Users might choose to run each mode separately or run all harvesters in All mode.

- **Migration**: performs data pre-processing to draw twitter messages exported from the Apache CouchDB database in JSON format. For instance, the 10 GB JSON file provided by Professor Richard O. Sinnott.
- **Streaming**: queries twitter messages in real-time (Figure 3).
- **Searching**: queries twitter messages from the list of users specified in the configuration file (Figure 4).
- **TweetID**: lookup twitter messages from the tweet id Covid19-related data set in TSV format provided by Banda et al. (2020) (Figure 5).

The configuration files allow users to configure twitter authentication keys, twitter messages filters, thread pool job executor, CouchDB connection. It comprises of the following JSON files:

- **authenConfig.json**: contains the common twitter authentication keys. The harvester component will use these configured keys to make a connection to Twitter if there are no authentication keys specified in harvester mode.
- **databaseConfig.json**: contains configured values for the database name, username, password, host, and port of the Apache CouchDB database.
- **auCoordinates.json**: contains geometry coordinates of Australia exported from AURIN.
- **filterConfig.json**: contains the following configured sections:
    - **streaming**: contains the approximate bounding boxes of Australia. The streaming mode only retrieves the twitter messages in real-time from this configured bounding boxes.

- **searching:** contains the twitter screen name list that needs to be harvested, and the twitter authentication keys used in searching harvest mode.
- **ruling politicians**: contains the twitter screen name list of interested politicians from the ruling party.
- **opposition politicians**: contains the twitter screen name list of interested politicians from the opposition party.
- **track**: contains all interesting keywords used to filter the content of received twitter messages.
- **user location filter**: contains interesting user locations in Australia. They are used to filter received tweets when the harvest component cannot extract coordinates from geo, coordinates, and place elements in tweet content.
- **tweet ids**: contains configured tweet id data set related to Covid19 from April 22 to May 18. Each data set is stored in a separate folder. It contains around 1 million tweets collected from all users around the world. The harvester component will maximize parallel processing by initializing 3 threads for each of the two processes in Virtual Machine. It means that the total 6 data sets will be handled at the same time. The thread number on each process, thread name, and corresponding twitter authentication keys are also configured in this section.

The harvest component also implements thread pool job executor (Figure 6) to process collected tweets. As a result, there is no bottleneck in the Apache CouchDB writing processing. Furthermore, the database writing processing does not block the harvest processing. So, the number of harvested tweets is considerably increased eventually.

The collected tweets are pre-processed before storing it to the Apache CouchDB database. The backend and frontend components can take advantage of these pre-processed data. The following steps are executed when pre-processing tweet data:

- **Extracting:** tweet id, created date-time, full tweet text, and user screen name are extracted from raw tweet data.
- **Analyzing:** the sentiment and pronoun analysis are performed based on the full tweet text. We applied the Sentiment Intensity Analyzer in VADER python module to extract the intensity of negative, positive, neutral emotion, and the compound score. For the pronoun analysis, we removed the special characters and tokenized the tweet content. After that, we classify every pronoun into four categories: first person singular ("i", "me", "my", "mine", "myself"), first-person plural ("we", "us", "our", "ours", "ourselves"), second-person pronoun ("you", "your", "yours", "yourselves"), and third-person pronoun ("he", "his", "himself", "she", "her", "hers", "herself", "it", "its", "itself").
- **Calculating:** the tweet coordinates are extracted from geo or coordinates elements from raw tweet content if applicable. If only place elements are included in tweet data, the middle point is calculated based on this place bounding boxes.
- **Transforming:** match_track_filter and politician_type flags are added to the collected data.

- If the tweet full text matches with the configured track in filterConfig.json, match_track_filter is set to True. Otherwise, it will be set to False.
- If the user screen name of a tweet is in configured ruling politicians in filterConfig.json, the politician_type will be set to "ruling". If this screen name matches with configured opposition politicians, the politician_type will be set to "opposition". Otherwise, the empty string will be set to politician_type.

In the end, we got over 1 million tweets in our application. The following table shows the main fields for each document we harvest into our CouchDB database (Table 1).

Field	Description
id	The integer representation of the unique identifier for the tweet.
created_at	Date time in "Y-m-d H:M:Sz" format when the tweet was created.
full_text	The tweet full text.
user	The user screen name.
match_track_filter	True: matches with configured track in filterConfig.json. Otherwise, False.
politician_type	Ruling, Opposition, None (empty string). Check the previous section for further information.
calculated_coordinates	The coordinates extracted from geo or coordinates or place elements of the twitter message.
coordinates_source	geo or coordinates or place.
emotions	an object with four keys: neg, neu, pos, compound. (neg: negative emotion score, pos: positive emotion score, neu: neutral emotion score, compound: compound score)
tweet_wordcount	The tweet text word count.
pronoun_count	an object with four keys: fps_cnt, fpp_cnt, sp_cnt, tp_cnt. (fps_cnt: first person singular count, fpp_cnt: first person plural count, sp_cnt: second person pronoun count, tp_cnt: thrid person pronoun count)
raw_data	The original twitter message.

Table 1 The Collected Data Structure

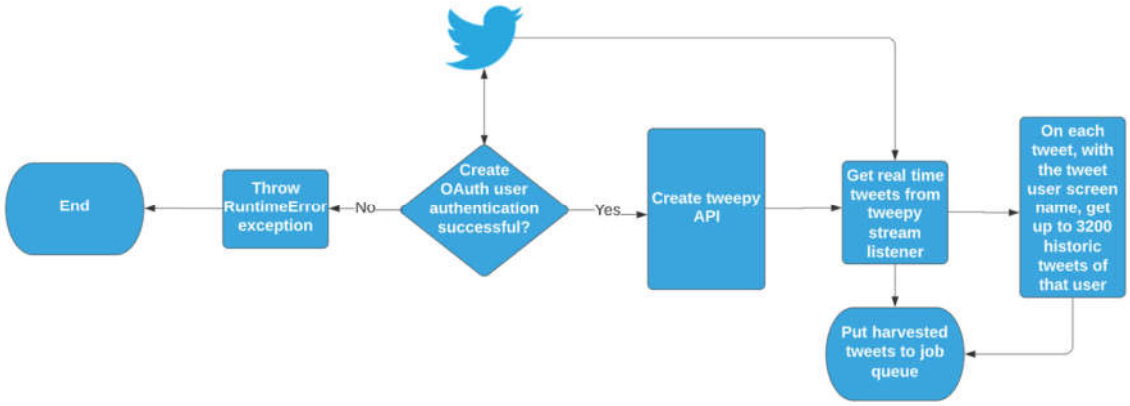

Figure 3 Stream Mode

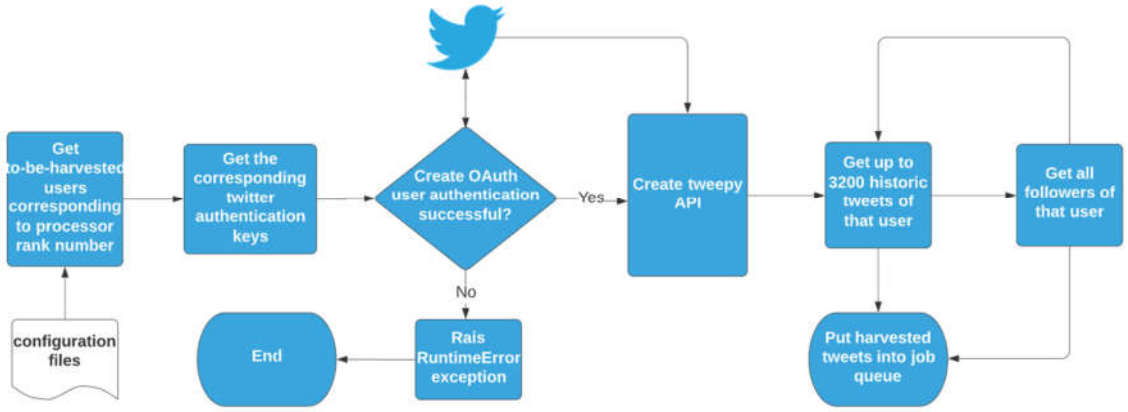

Figure 4 Search Mode

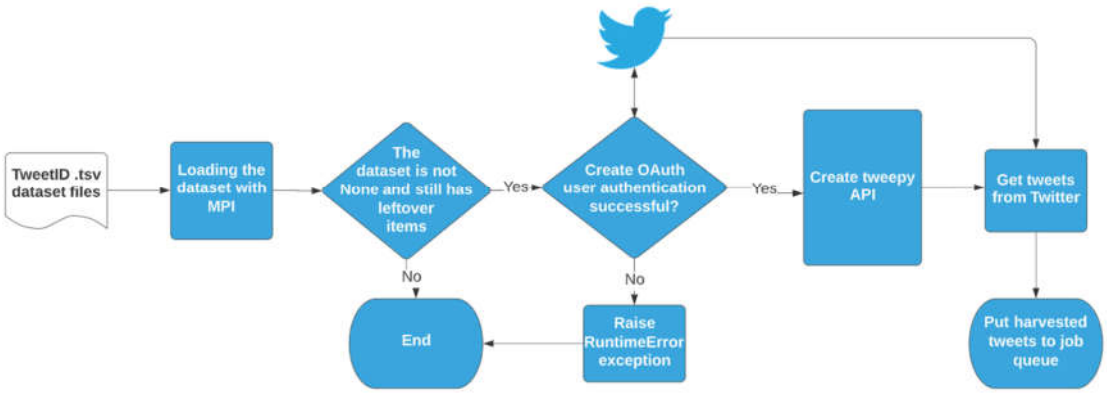

Figure 5 TweetID Mode

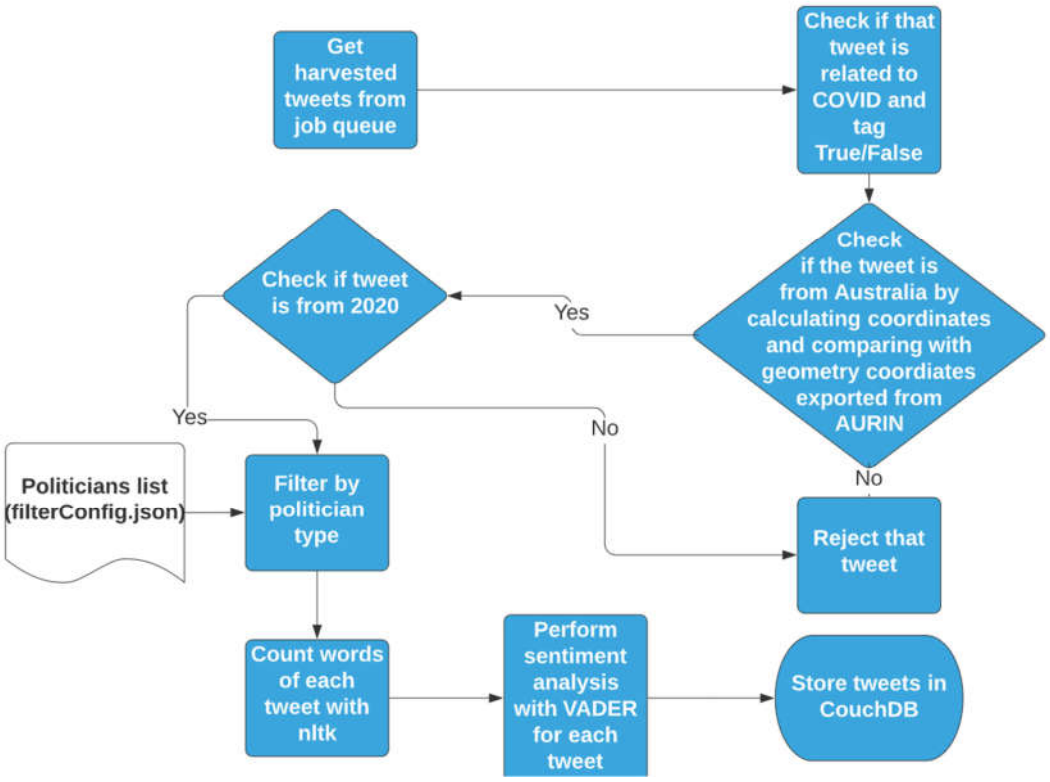

Figure 6 CouchDB writing process

2.3. Cloud-Based System Deployment

For our deployment we used **Ansible** and **Docker** to facilitate dynamic deployment. Ansible ("Ansible Documentation", 2020) was used as the configuration management and deployment tool and Docker ("Docker Documentation", 2020) was used to containerize specific parts of the application. Docker compose was used along with Ansible to deploy web-frontend and web-backend. Four instances were used from Melbourne Research Cloud (MRC) out of which

three were assigned to CouchDB clusters and harvesters for harvesting tweets. A single instance was used for the application containing both the backend and the frontend.

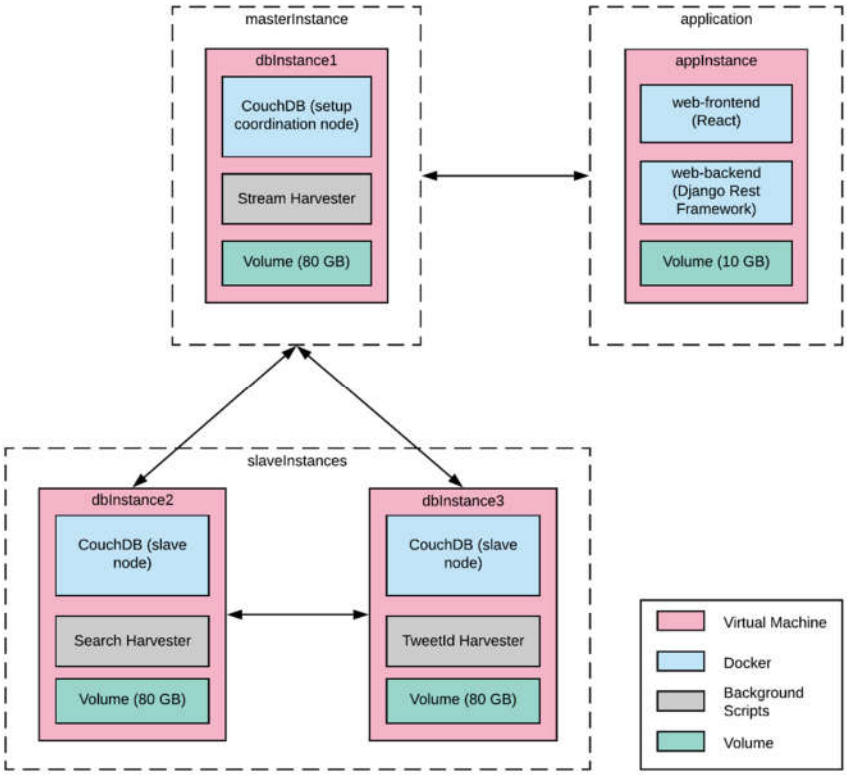

Figure 7 Cloud-Based System Architecture

2.3.1. Instant Deployment

The instances were created and deployed using ansible with host as localhost. Our system is divided into 4 different groups. The groups are as follows:

1) appInstance
2) masterInstance
3) slaveInstances
4) allInstances

As hosts are allocated to the group project, the hosts are grouped dynamically during runtime to the different groups according to the instance names into the host inventory files. Here **instance** creation **security groups**, **remote security group** for VM to VM networking, **volume** allocation, and common dependencies installation takes place.

As shown in Figure 7 each instance was assigned a volume. The master and slave instances were each assigned a volume of **80 GB** each to accommodate the increasing size of CouchDB tweeter data. The main application instance is also assigned a volume of **10 GB**.

2.3.2 CouchDB Deployment

Since the localhost is not required anymore after instance creation, the proxy settings, volume mount, and installation of packages for instances are also taking place here besides CouchDB deployment. For proxy, proxy links are added to the **/etc/environment path**. Volumes are mounted on the path /data for each instance and dependency for docker, docker compose and basic instance modules are installed here.

As shown in Figure 7 above, the CouchDB docker created using docker. Ansible module **docker_container** (Cove S. et al., 2020) for previous container deletion, creation of new containers, and restarting of containers. As problems were arising in bind mount volume creation using the **ibmcom/couchdb3** image, the official CouchDB image **couchdb:3.0.0** was used, and a bind mount volume was used between the **80 GB** instance volume and the CouchDB data path for each backup and storage. Each docker container was assigned the following parameters:

```
# Couchdb cluster variables
couchdb_name: "couchdb{{ ansible_default_ipv4.address }}"
couchdb_image: couchdb:3.0.0
couchdb_user_name: admin
couchdb_password: password
couchdb_cookie: '0e0cfc55cf5cde25799c547bdf0008ea'
couchdb_secret: '0e0cfc55cf5cde25799c547bdf001633'
couchdb_uuid: '0e0cfc55cf5cde25799c547bdf0008ea'
couchdb_nodes: "{{ groups['slaveInstances'] }}"
couchdb_master_node: "{{ groups['masterInstance'] }}"
```

Figure 8 CouchDB Variables

The **vm.args** and the **local.ini** files were also updated to add the cookie, update the bind address, and adding the secret. Also, the ports, **5984**, **4369** and **9100** were opened to all communication between the clusters and the master node to the outside network. The way CouchDB was configured was to ensure that more clusters could be added dynamically with a single deployment script. Since VMs were added dynamically by the previous create_instances.sh script based on the number of instances defined in the **localhost.yaml** file, any IP address defined under **dbInstance** will automatically have the docker CouchDB deployed provided the values of **q** and **n** are changed in the **deployment/roles/couchdb-environment/tasks** folder.

Figure 9 Curl query with q and n value

To make sure the setup is running in each instance, ansible **loop_control** extension ("Loop", 2020) is also used with **ansible_loop.index** variable to count the number of nodes dynamically to add nodes to clusters. This ensures that the exact number of nodes are deployed as dynamically allocated by the create_instances.sh script. The application then scales accordingly, and all the **dbInstance** VMs contain the CouchDB container with the specific **nodename**. In addition, containers are used as it is very easy to set up CouchDB in containers and we do not require to install any extra dependencies for the containers to run.

2.3.3 Harvester Deployment

There are 3 types of harvesters deployed in our system. All are developed for general purpose deployment. Adding a new VM and adding a specific role out of three harvester roles in file path **deployment/deployment_harvester.yaml** will deploy the harvester to that VM. The harvesters harvest data and communicate with the local CouchDB node available in the specific VM deployed in. Therefore, they are strictly deployed only on **dbInstances** and not in-app instances. During deployment, harvester files and python script files are copied to the **/data** path of the instances, and pip dependencies are installed to install **mpi4py**. Then depending on the harvester mode, bash scripts are invoked and run as background scripts using async and poll ("Asynchronous Actions and Polling", 2020) in ansible. The setting for **poll** is set to **0** so that the Ansible moves on to other tasks for deployment and shifts the bash file as a background task. This is in the file path **/deployment/roles/harvester-search/tasks/main.yaml.**

The reason why this was chosen over docker was that since bash scripts are easier to invoke in the terminal, it was decided to just run them in background considering the time constraints. Also, it is much faster to copy the files and run them rather than copying them again into docker.

2.3.4 Application Deployment

The application is contained in a single VM. As shown in the figure, the application contains both the **web-frontend** and **web-backend**. The application is set up using **docker-compose.yaml** file, **Dockerfile.frontend** and **Dockerfile.backend** files. The files are first copied into the VM. Then Ansible builds the containers and runs then using the docker_compose module in ansible. The deployment also uses Jinja2 templates for the following files: (1) **backendUrl.js** and (2) **Settings.py**

These files contain the IP addresses of the VM being deployed to and the remote address of the CouchDB **masterInstance**. To make them dynamic, these files are rendered using Jinja2 files to copy the remote address and the VM address of **appInstance**. In this way, the app will always be deployed to the VMs under the group **appInstance** and connect to the **masterInstance** CouchDB as shown in the figure.

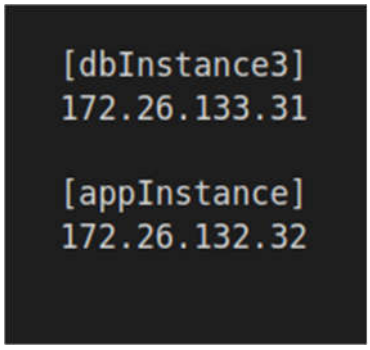

Figure 10 appInstance IP Address

Since our frontend is in **React** ("Reactjs Documentation", 2020) and the backend is in **Djangorestframework**, we need to set these up in containers and not as a background task. **Djangorestframework** needs a python **virtualenv** ("Quickstart", 2020) environment. Running them in container prevents the need to have a virtual environment setup and our backend can run smoothly. To make react work in a container, a special parameter **tty** is set to **true**. Otherwise, the container running React will stop after a few seconds.

2.3.5 How the Application Integrates with the deployment

As the application is deployed, the Instances are created first. After while the CouchDB is deployed and configured in the cluster. The Harvesters are deployed next and the Database is populated with tweets. Since it takes time for the data to be processed in CouchDB, the data indexed under views takes time to show. After that the application is deployed. The application is deployed in a single instance and contains both the frontend and the backend.

2.3.6 Pros and Cons of Ansible and Docker

Pros:

- Ansible is very easy to work with as it uses a **yaml** syntax.
- Dockerfile uses its own syntax which is very easy to understand.
- Docker-compose uses **yaml** syntax and is easy to grasp.
- Both have well-defined documentation available online and it is easy to look for different modules and parameters.
- There are several ways to code for docker, docker-compose in Ansible.

Cons:

- One must have a good understanding of networking to configure networks and proxies in Docker.
- Ansible is hard to debug if **shell** or **cmd** modules are used as changes are not registered by Ansible. The **-vvv** flag also does not show all the details of playbook failure.
- It is hard to configure volumes as it requires an understanding of each image. For example, bind mount works in **couchdb:3.0.0** but not in **ibmcom/couchdb3**.

2.4. MapReduce

To query data from the Apache CouchDB database, we implemented map and reduce functions. For example, the photo below shows a demonstration of how we query the movement data of the people in Australia during the outbreak.

Map function

```
function (doc) {
   if (doc.user && doc.calculated_coordinates) {
    if (doc.calculated_coordinates.length == 2) {
        emit(doc.user, [doc.calculated_coordinates[1].toFixed(3), doc.calculated_coordinates[0].toFixed(3)]);
    }
   }
}
```

Custom Reduce function

```
function (keys, values, rereduce) {
  if (rereduce) {
    var result = [];
    values.forEach(function (value) {
      value.forEach(function (point) {
          result.push(point);
        }
      )
    });

    var uniquePoints = [];
    result.forEach(function (point) {
        var existing = false;
        for (var i = 0; i < uniquePoints.length; i++) {
            if (uniquePoints[i][0] === point[0] && uniquePoints[i][1] === point[1]) {
                existing = true;
                break;
            }
        }

        if (existing === false) {
            uniquePoints.push(point);
        }
    });

    if (uniquePoints.length > 1) {
      return uniquePoints;
    } else {
      return [];
    }
  } else {
    return values;
  }
}
```

2.5. Application

Figure 11 illustrates the architecture of the application. Overall, the application is a single-page web app comprised of three key components which are:

(1) Storage component including CouchDB (a NoSQL solution) and SQLite (a small database).
(2) Front-end: implemented by ReactJS.
(3) RESTful API Back-end: implemented by Django Rest Framework.

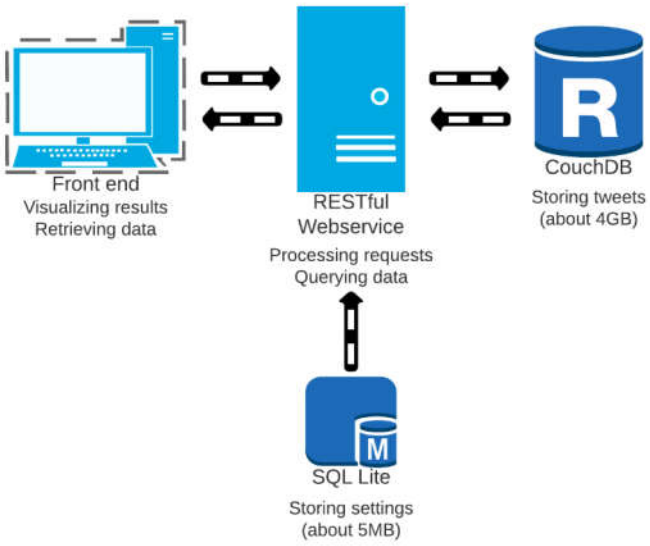

Figure 11 Application Architecture

The first component is the storage component. In this component, the CouchDB solution is the main database which stores the tweets we harvested. On the other hand, the SQLite database contains configuration data and static data such as vector data (used in some features such as rendering the heat map).

The second component of the application is the frontend implemented by ReactJS along with several libraries for visualizing data such as Recharts and D3JS, and libraries for making modern web maps such as Leaflet. Using ReactJS brings a lot of benefits for building a frontend such as performance, scalability. In fact, developing the frontend by ReactJS also granted us unlimited access to available resources. As a result, the quality of our frontend is guaranteed, and the software development time is reduced significantly. Figure 12 shows the flow process of a simple React application:

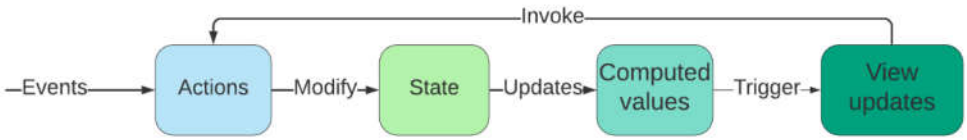

Figure 12 Process of React Application

The third component of the application is the REST API backend implemented by Django Rest framework. It is important to realise that although the NoSQL solution (CouchDB) provides RESTful APIs, building a backend like a "middleware" between the frontend and the NoSQL solution is still truly necessary because it makes the development team easier and more efficient in customizing APIs. In addition, a backend also improves the **scalability** of the system when the system has more access to other resources such as a SQL database or configuration files. In fact, we used a SQLite database for storing some static settings for the application as mentioned above. Moreover, the backend is also a layer hiding the complexity of accessing directly to the database. Eventually, building a backend also helps increase the **security** of the application via establishing security mechanisms such as CORS, OAuth.

The design of the backend follows the ideal of the n-layer design pattern Figure 13. Thereby, the web service becomes more **scalable**, **secure**, and better **performance**. Most importantly, this design improves **speed of development**. Therefore, it is an extremely suitable design for the backend.

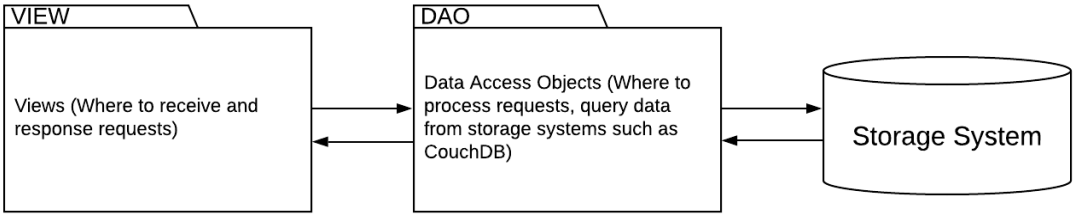

Figure 13 2-layer backend

In conclusion, the four key factors made the team to choose the approaches mentioned above areas:

- Speed of development
- Scalability
- Security
- Performance

Map function

```
function (doc) {
  if (doc.user && doc.calculated_coordinates) {
   if (doc.calculated_coordinates.length == 2) {
       emit(doc.user, [doc.calculated_coordinates[1].toFixed(3), doc.calculated_coordinates[0].toFixed(3)]);
   }
  }
}
```

3. Deployment Guide

Navigate to the deployment folder by typing:

bash$ cd deployment

The instance key **mayank-sharma-key.pem** has been added to the deployment folder.

3.1. Deployment Instances (Step1)

Type the following command to deploy the instances:

bash$ ./create_instances.sh

Enter the **Openstack Password** as: NjljNWFmMDgwNTI5ZDc2
Enter your **root password** of your **localhost**.

Please deploy this *ONLY* if you do not have any valuable data in CouchDB.
Make sure to back up the data before if required.

To scale the system, add or remove an instance under instances and volume parameters in the **localhost.yaml** file in **deployment/host_vars** folder. Also, add the instance name under any group as required in the **"Write inventory file"** section on the file path **deployment/roles/instance/tasks/main.yaml**.

3.2. Deploying CouchDB cluster (Step2)

Type the following command to deploy the couchDB cluster:

bash$ ./deploy_couchdb.sh

You will be prompted to enter Openstack Password. Here you can just press enter.
You will be prompted to enter the root password of your localhost, but you simply press enter.

These prompts have been added in case future deployment through the localhost is added to this part.

Please deploy this *ONLY* if you do not have any valuable data in CouchDB.
Make sure to back up the data before if required.

To scale the database cluster, make sure an instance has been added or removed during instance creation and the **q** & **n** values in **main.yaml** in file path **deployment/roles/application-environment/tasks** is updated.

3.3. Deploying Harvesters (Step3)

Type the following command to deploy the Harvesters:

bash$ ./deploy_harvester.sh

You will be prompted to enter Openstack Password. Here you can just press enter.
You will be prompted to enter the root password of your localhost, but you simply press enter.

These prompts have been added in case future deployment through the localhost is added to this part.

To scale the harvesters, make sure an instance has been added or removed during the instance creation. In that way the deployment code will automatically deploy the harvester to the VM with the specific harvester. The harvester must be part of a specific CouchDB instance.

3.4. Deploying Application (Step4)

Type the following command to deploy the Application:

bash$ ./deploy_app.sh

You will be prompted to enter Openstack Password here you can just press enter.
You will be prompted to enter the root password of your localhost, but you simply press enter.

These prompts have been added in case future deployment through the localhost is added to this part.

To scale the app, make sure an instance has been added or removed during the instance creation and the name is updated in the **main.yaml** file **"Write Inventory file"** section under **deployment/roles/instance/tasks**. In that way the application can be deployed to that VM as well.

4. Data Scenarios and Results

Based on the harvested data during the Covid-19 pandemic, we have created some visualization to be able to have a more specific understanding about the feelings and habits of the people in Australia over the difficult period. Specifically, we have classified the data visualization into 7 different tabs on the website that are Home, Sentiment, Sentiment Analysis, Movement, User Tracker, Language, and Politics.

Firstly, the Home tab shows us descriptive statistics about the harvested data as well as gives us an overview about the activity of the Twitter users from Australia during the outbreak. It can be seen of the charts, while the proportion of tweets which do not mention Covid-19 is 92%, just about 8% of tweets contain information about the virus. The total tweets which do not contain geographic information (Figure 14) are nearly double the number of tweets with location information. Moreover, we found out that the residents in Australia were inclined to tweet during the morning and do less frequently in the evening (Figure 15). Furthermore, the Twitter users were likely to be more active on weekends Figure 16.

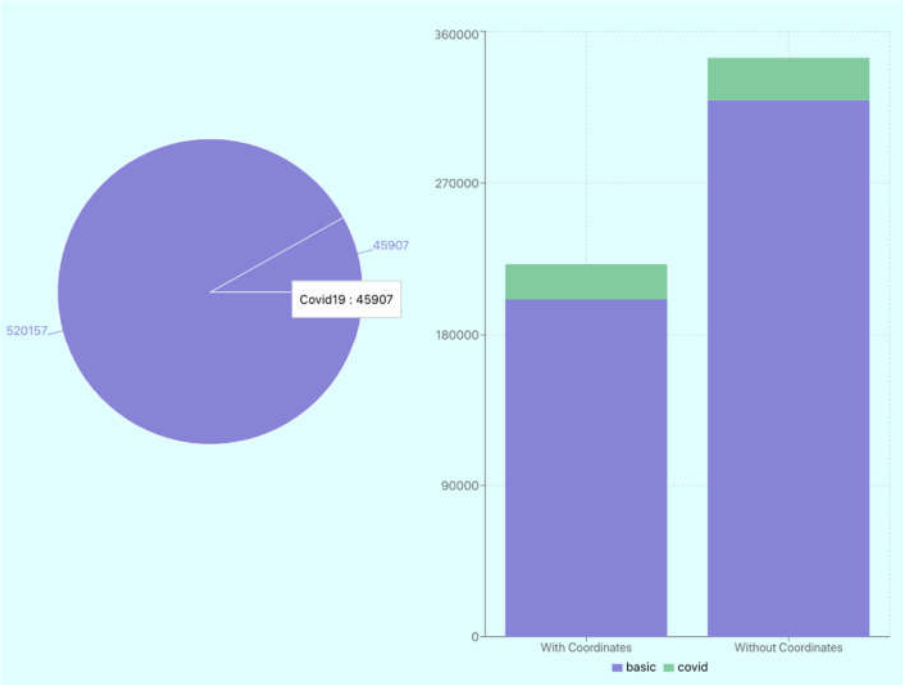

Figure 14 covid tweet vs. non-covid tweets

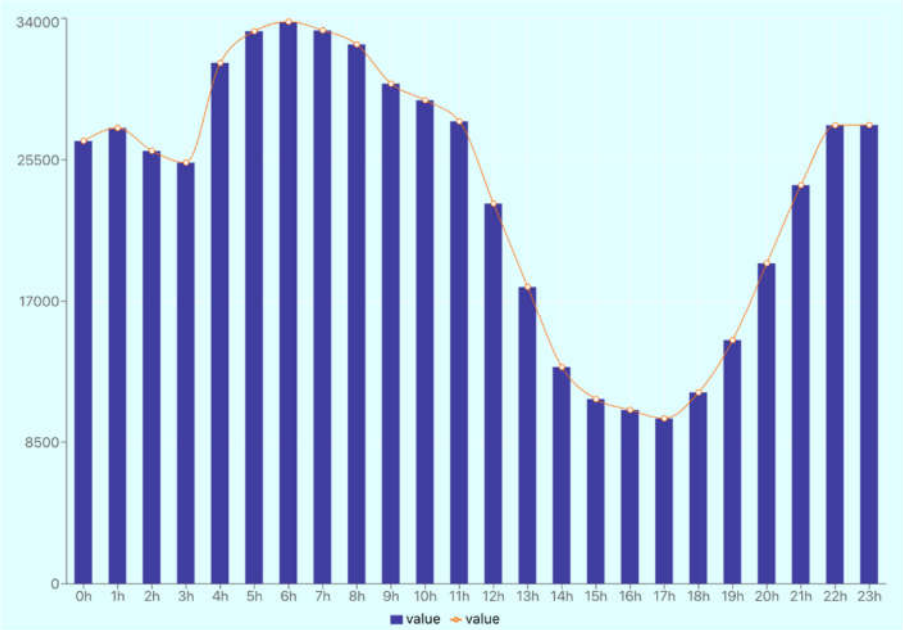

Figure 15 Number of Tweets per hour on Home tab

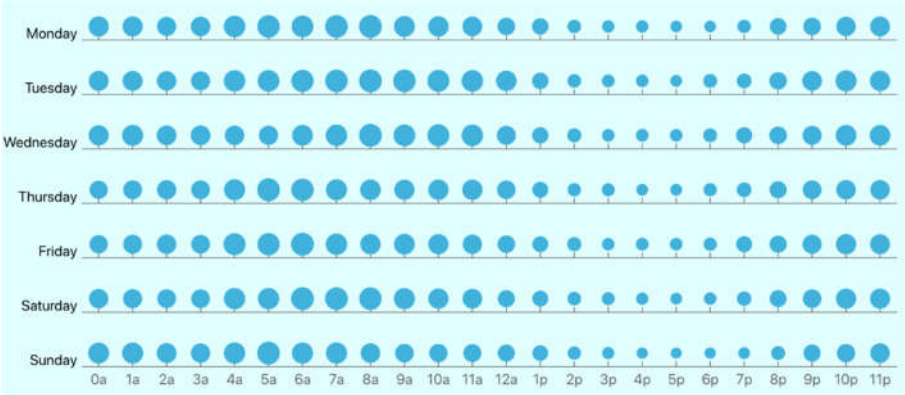

Figure 16 User Activities Pattern Within a Week

The heat map on the Sentiment tab shows a demonstration of different levels of emotions for some regions in Australia (Figure 17). In particular, the red areas on the map below are regions having the number of negative tweets is greater than the number of positive tweets, meanwhile, the green regions show the opposite meaning. With some polygon maps imported from Aurin listed in the tree menu on the right, a user could choose from the tree menu a specific area that they want to study. For example, when looking at the photo below, it is clear to state that the citizens living by the east coast were likely to be happier than the residents living near in the adjacent areas. Furthermore, the people locating in the west coast did not seem to be happy during the outbreak.

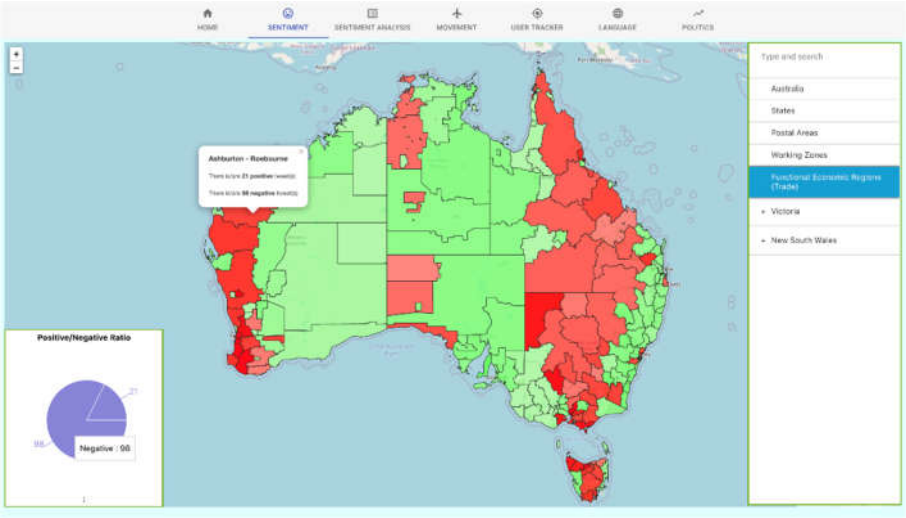

Figure 17 Emotion Heatmap in Australia (Functional Economic Regions)

Next, the Sentiment Analysis tab gives us more details about the emotions of the people in Australia during Covid-19. Particularly, in the pie charts below (Figure 18**)**, the neutral tweets constitute the biggest proportion (in blue), the red part illustrates the number of negative feelings and the green is for positive emotions. As the charts show, the residents in Australia remained extremely neutral over the Covid-19 period. The virus did not seem to be a severe problem for them because they likely did not feel negative about it. Regarding the emotion level of the people during the day, we found out that people did not show many negative thoughts about Covid-19 as they thought of the other topics.

In Figure 19, we also see that the residents often felt most negative during the night and in the early morning. Figure 20 shows that the people were happier during non-working hours.

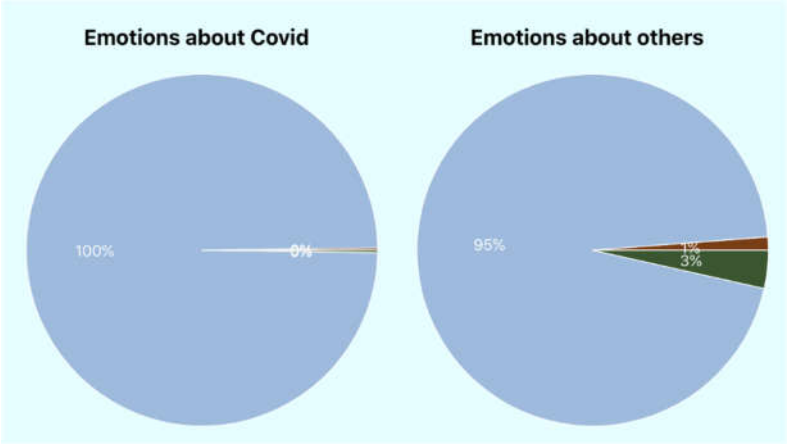

Figure 18 The Pie Charts of Neutral, Negative, and Positive Emotion

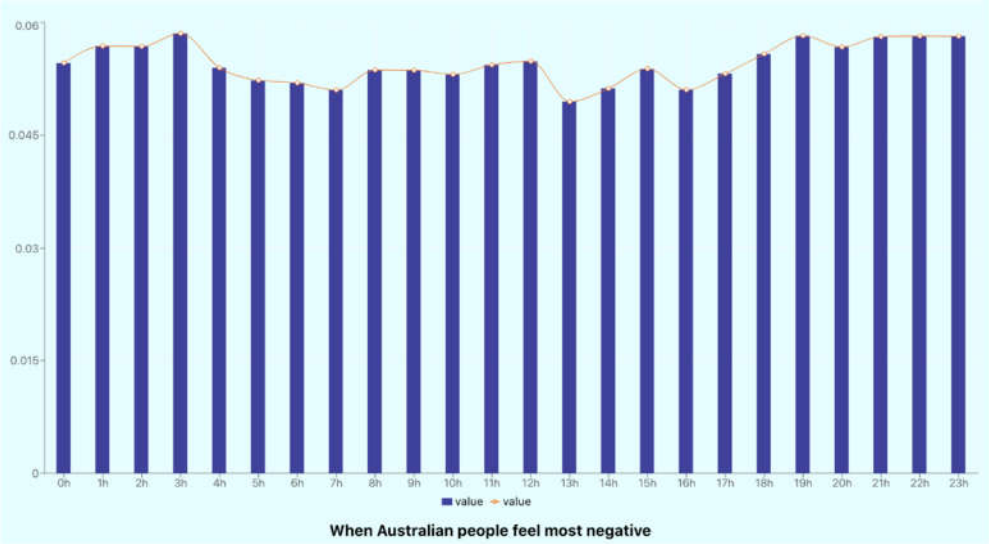

Figure 19 Negative Emotion Dynamic Within a Day

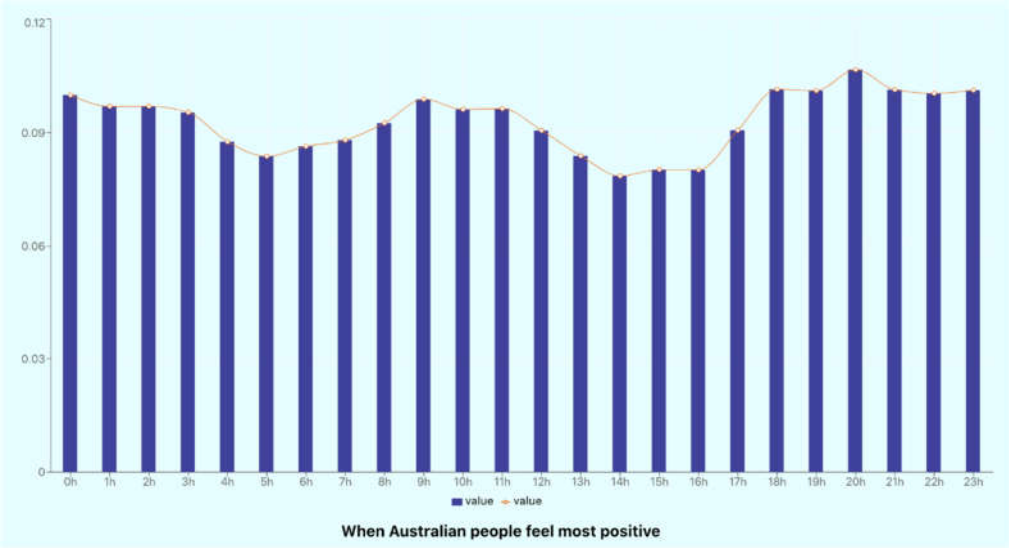

Figure 20 Positive Emotion Dynamic Within a Day

The traffic map is displayed in the Movement tab. As a user, they can select one of the three transport volume levels that are 1000, 2000, and 3000 citizens. The outcome Figure 21 shows us an interesting detail that the eastern states of Australia had much more commuting if compared to the western states. The most visited cities during the epidemic were Canberra and Sydney.

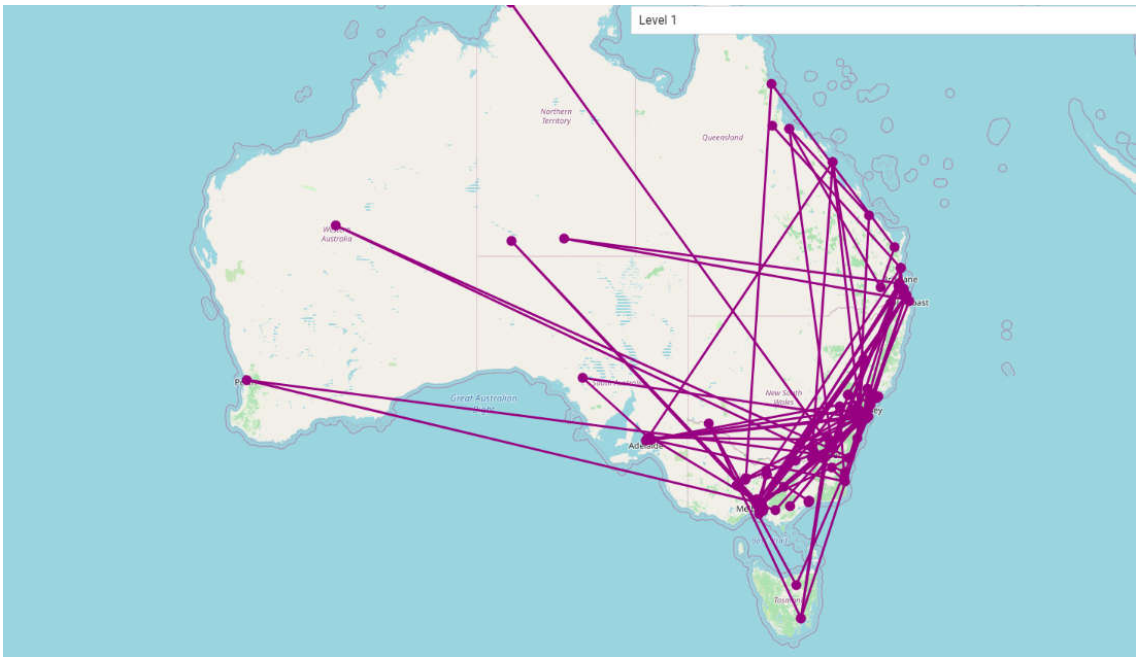

Figure 21 Traffic Map

In the User Tracker tab, we learn about the movement of the residents during the outbreak. Particularly, on the right of the screen, we could select a twitter user to learn about their visited points as well as their profile at the left bottom corner. The outcome tells us the truth that the people did not move around during the outbreak. Take user unixbigot below as an example, he never left Brisbane and stayed there over the period (Figure 22). It is understandable because most of the people were likely to try to stay home as much as possible

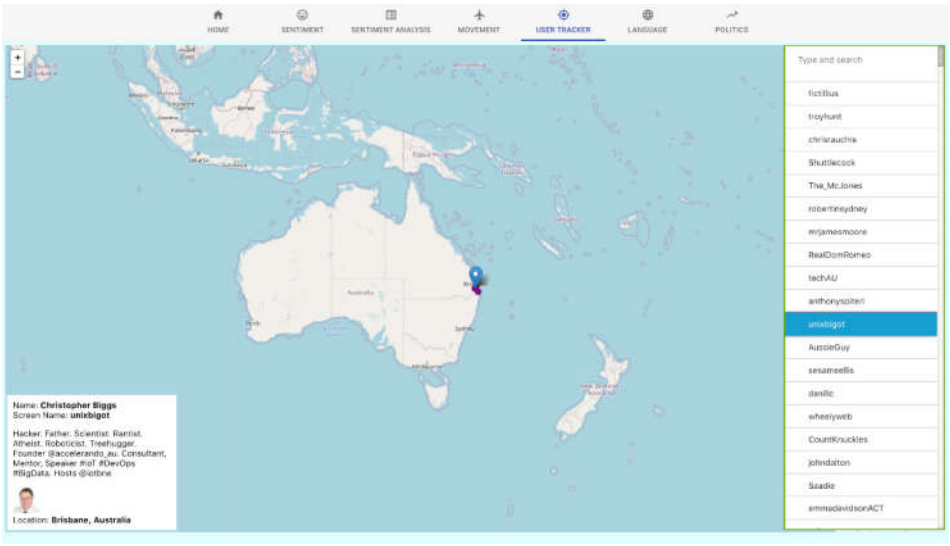

Figure 22 User Tracker

In the Language tab, the most used languages for tweeting are shown. Based on the harvested data, we realized that the top 5 languages used in Australia to tweet are English, Indonesian, Spanish, Portuguese, and France (Figure 23). According to Statista (Felix Richter, 2013), the

top language used in tweets are English, Japanese, Spanish, Malay, Portuguese, Arabic, French, Turkish, Thai, and Korean. Comparing the two sources, it is reasonable that English is still the dominant language on Twitter in Australia. However, Japanese is not the second most common language. As a result, we tried to find more information about the language in Australia. We used the NCRIS-enabled Australian Urban Research Infrastructure Network (AURIN) Portal e-Infrastructure to access "SA3-P13 Language Spoken at Home by Sex-Census 2016" (Richard et al., 2014). After aggregating the data, we found out the top 3 languages spoken other than English in Australia are Indo Aryan (61,9197), Mandarin (596,566), and Arabic (321,749) (AU_Govt_ABS_Census, 2017). In summary, this phenomenon shows that Twitter is popular in certain language populations in Australia. The community on Twitter is not aligned with real-world demography. Researchers should be cautious when connecting twitter data to the real world in Australia.

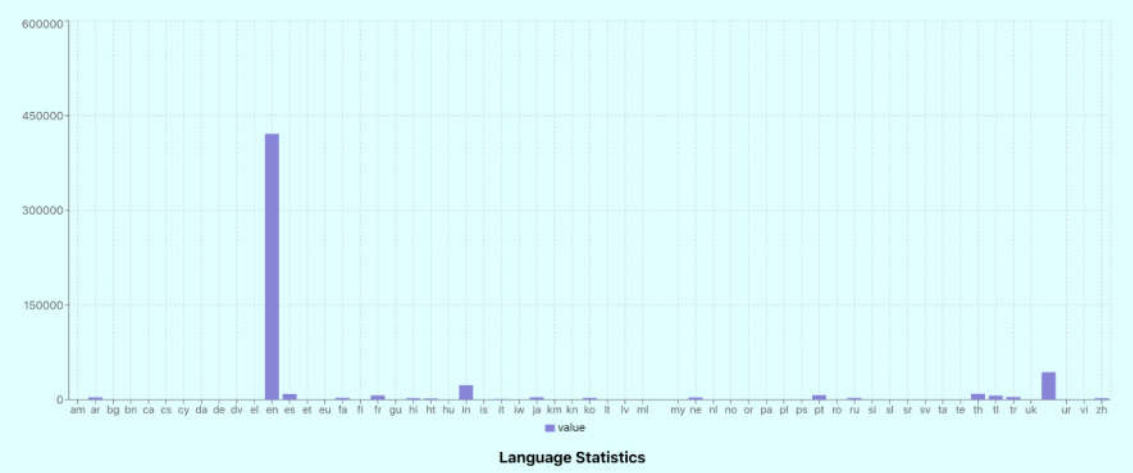

Figure 23 Language Statistics

The Politics tab is the last window showing how Australian politicians responded to Covid-19. Using tweets collected from a given list of politicians from the two main Australian political parties (the Labor party and the Liberal party), we found out how often they mentioned the Covid-19. In Figure 24, the orange part is tweets related to Covid-19 and the green is not. The ruling party (the Liberal party and the allied parties) cared about Covid-19 more than the opposition. Furthermore, we can see the details of each politician in Figure 25.

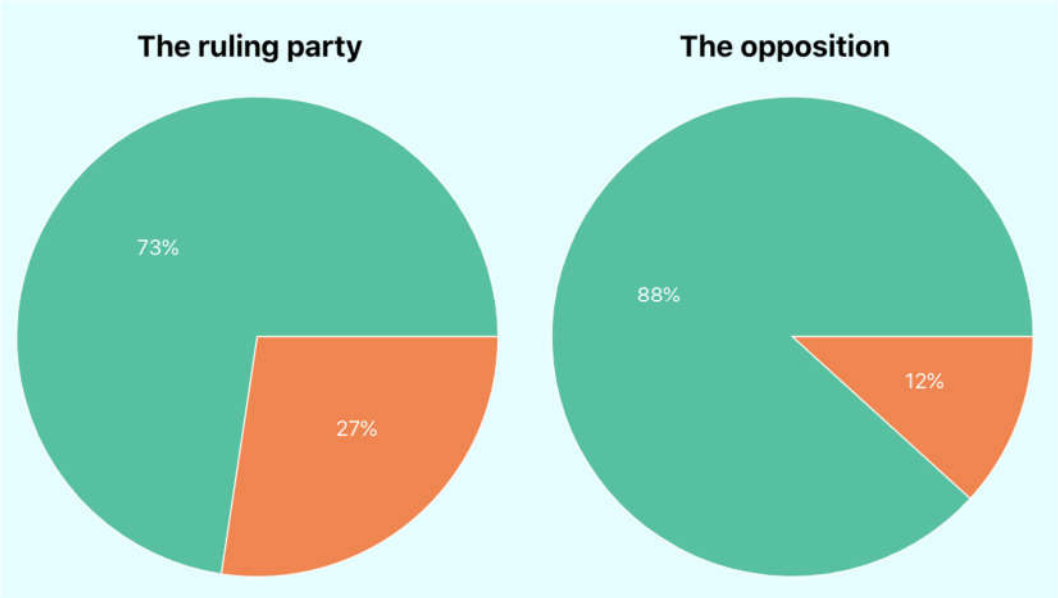

Figure 24 Covid tweets proportion of ruling party and opposition party

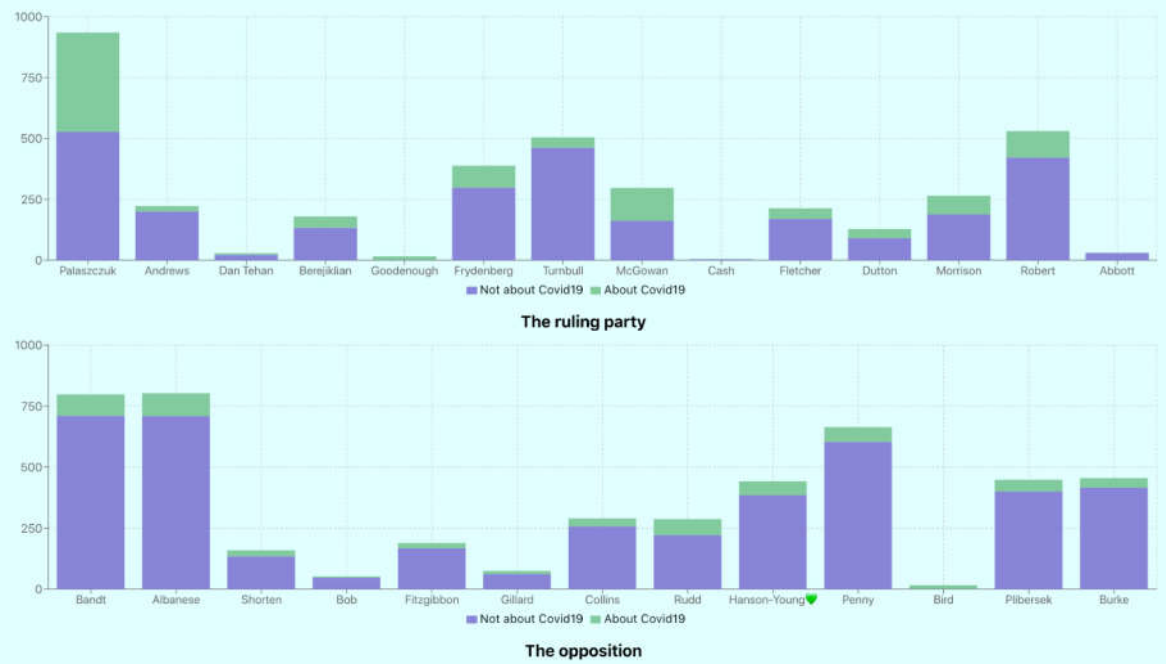

Figure 25 Covid Tweets Bar Plot of Each Pilitician From Two Parties

5. Discussion

5.1. UniMelb Research Cloud

5.1.1. Pros

- UniMelb Research Cloud is easy to navigate.

- It is easy to create volumes, security groups, and instances using the web console.
- The web console can easily allow anyone to create a basic instance setup with security groups and volumes attached.
- The Cloud also provides several different VM images to work with and different flavors for each instance.

5.1.2. Cons

- It is very hard to navigate and look for image flavors in the cloud.
- Also, when creating an instance using ansible, the OpenStack tries to recreate an instance if there is a slight change in the instance parameters. It assumes that the VM does not exist leading to failure of the playbook.
- It is also hard to navigate to the group projects section.
- The system is not designed for a non-coder to launch instances using ansible easily.
- Some basic understanding of the OpenStack framework is required.

5.2. Error Handling

5.2.1. Harvester

5.2.1.1 Duplicated Tweet Removal

The harvest component uses tweet id as document id when inserting collected data into the Apache CouchDB database. It means that the database only allows inserting a new document whose id does not exist in the working database. As a result, the duplicated tweet issue is resolved.

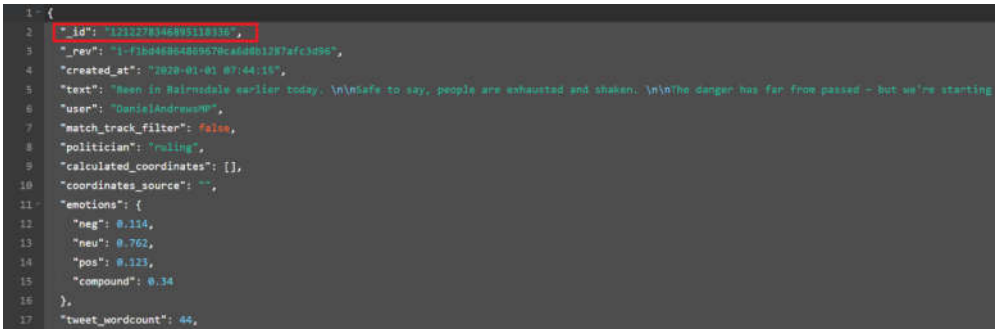

Figure 26 - The duplicated tweets error handling

5.2.1.2. Stream Listener Error Handling

The harvest component catches MemoryError when listening to real-time Twitter messages. If this exception is encountered, the harvest process will be stopped. The warning message will be displayed on the terminal, and the user must manually restart the harvest process.

For other exceptions, the harvest component will automatically re-connect to Twitter to harvest the twitter messages. The exponential backoff is implemented to ensure that our Virtual Machine does not generate excessive load.

```
while(True):
    try:
        # Instantiate the stream listener
        listener = StreamListener(self.tweepy_api, self.config_loader, self.writer)

        # Streaming and filtering tweet data
        stream = tweepy.Stream(auth = self.tweepy_api.auth,
            listener = listener, tweet_mode = constants.TWEET_MODE)

        # Filer tweets based on configured locations
        locations = self.config_loader.get_streaming_locations()
        if (locations is not None):
            stream.filter(locations = locations)
    except MemoryError as error:
        print("Encountered the memory exeption. Please restart harvester process.")
        break
    except Exception as ex:
        print(f"Exception occurred during tweet streaming. {ex}")

        if self.minimum_backoff_time > self.maximum_backoff_time:
            self.minimum_backoff_time = self.default_backoff_time

        delay = self.minimum_backoff_time + random.randint(0, 1000) / 1000.0

        print(f"Trying to reconect after {delay} seconds")
        time.sleep(delay)
        self.minimum_backoff_time *= 2
```

Figure 27 - Streaming error handling

5.2.1.3. Rate Limit

We are using a total of 9 separate twitter authentication keys for harvesting. One for streaming mode. Two for searching mode (one process in Virtual Machine takes one configured keys). Six for tweetID mode (there are 2 parallel processes: 3 threads for each process. Therefore, there are a total of 6 threads, each thread uses one separated authentication key).

When querying historic tweets for a specified user, we try to get all maximum of 3200 tweets allowed by Twitter. We send separated requests, each request only get 200 tweets, the harvest component will sleep 2 seconds before sending new requests. These continuous requests will be stopped when we reach the maximum of 3200 tweets.

```
try:        You, 22 days ago • - Support to query 3200 tweet from any users.
    # Make initial request for most recent tweets
    new_tweets = tweepy_api.user_timeline(screen_name = screen_name,
        count = constants.LIMIT_COUNT_PER_REQ, tweet_mode = constants.TWEET_MODE)

    # Put all new tweets into final tweets list
    alltweets.extend(new_tweets)

    if (len(new_tweets) > 0):
        # Save the id of the oldest tweet less one
        oldest = alltweets[-1].id - 1

        # Keep grabbing tweets until there are no tweets left to grab
        while len(new_tweets) > 0:
            time.sleep(constants.TWO_SECONDS)

            # All subsiquent requests use the max_id param to prevent duplicates
            new_tweets = tweepy_api.user_timeline(screen_name = screen_name,
                count = constants.LIMIT_COUNT_PER_REQ, max_id = oldest, tweet_mode = const

            # Put all new tweets into final tweets list
            alltweets.extend(new_tweets)

            # Update the id of the oldest tweet less one
            oldest = alltweets[-1].id - 1
except tweepy.TweepError as ex:
    print(ex)
```

Due to a huge number of followers of a specific user, we continuously query 200 follower users in each request, then, sleep one minute to avoid the rate limit. In searching mode, after query historic tweets for a configured user in filterConfig.json, we get all follower users of this configured user, then querying the maximum of 3200 historic tweets for each follower user.

```python
def get_followers(self, tweepy_api, user_name, max_count):
    # Initialize a list to hold all followers
    followers = []

    for page in tweepy.Cursor(tweepy_api.followers,
                              screen_name = user_name,
                              wait_on_rate_limit = True,
                              count = constants.LIMIT_COUNT_PER_REQ).pages():

        try:
            # Put all new follower into final follower list
            followers.extend(page)

            # Check if total followers reached max count or not
            if ((max_count != -1) and (len(followers) >= max_count)):
                break
        except tweepy.TweepError as ex:
            print("Going to sleep:", ex)
            # Sleep 60 seconds to avoid rate limit issue
            time.sleep(constants.ONE_MINUTE)

        # Sleep 60 seconds to avoid rate limit issue
        time.sleep(constants.ONE_MINUTE)

    return followers
```

There is 1 million tweet id in each Covid19-related data set in TweetID harvest mode. Therefore, we cannot look up all twitter messages at the same time. We divide 1 million tweet ids in each 100-tweet id package (the maximum tweet id number can look up at the same time). Then, continuously sending requests to lookup all tweets, the harvest component will sleep 2 seconds before sending a new request to Twitter API.

```python
def lookup_tweets(self, tweet_ids):
    tweet_count = len(tweet_ids)

    try:
        for i in range((tweet_count // 100) + 1):
            all_tweets = []
            # Catch the last group if it is less than 100 tweets
            last_index = min((i + 1) * 100, tweet_count)
            # Sleep 2 seconds to avoid rate limit issue
            time.sleep(constants.TWO_SECONDS)
            full_tweets = self.own_tweepy_api.statuses_lookup(tweet_ids[i * 100 : last_index])

            # Check if tweet is in configured user filter locations
            processed_tweets = self.helper.filer_tweets_with_coordinates(full_tweets, True)

            # Write all tweets to counchdb
            if (processed_tweets):
                self.writer.write_to_counchdb(processed_tweets)

        return full_tweets
    except:
        print("Failed to loopkup statuses.")
        traceback.print_exc(file = sys.stdout)
```

5.2.2. Backend & Frontend:

For RESTful APIs in the system, they will always return a 400 response in case there are exceptions during the process of handling.

```python
class GetUserInfoView(views.APIView):
    user_dao = UserDAO()

    def get(self, request, pk):
        try:
            result = self.user_dao.get_user_info(int(pk))
            return Response(result, status=status.HTTP_200_OK)
        except Exception as e:
            return Response(str(e), status=status.HTTP_400_BAD_REQUEST)
```

For each API call on the frontend, the response will always be checked if its status is 200 (a success response). The frontend will always catch exceptions in case the response is not a success response. This guarantees that the availability of the frontend always remains stable even when several APIs on the backend side is down.

```javascript
axios.get(backendUrl.get_most_active_users).then(response => {
    if (response.status === 200) {
        let treeMenuData = [];
        let users = response.data;
        users.forEach(user => {
            let dataItem = {
                key: user.key[0],
                label: user.key[1],
                nodes: [],
                user_key: user.key,
                number_of_tweets: user.value
            };
            treeMenuData.push(dataItem);
        });
        this.setState( state: {
            treeMenuData: treeMenuData
        });
    }
}).then(() => {
    this.setState( state: {
        loading: false
    });
});
```

6. Team Roles

Name	Responsibilities
Hong Ngoc Nguyen	Designing and developing the application (both frontend and backend). Reviewing the process of harvesting.
Nhan Kiet To	Supporting the development on front end.
Ting-Yu Lin	Supporting some functions in harvesters. Data preprocessing and data analysis.
Mayank Sharma	Deployment of the entire system
Duc Trung Nguyen	Developing twitter data harvester with 5 different modes: All, streaming, searching, migration, and TweetID.

Appendix

Github link: https://github.com/trundn/COMP90024_Assignment2

Youtube link:
Part1: https://youtu.be/NlOQqtNZgKg
Part2: https://youtu.be/qYTEiEaWyhc

Live Application link: http://172.26.132.32:3000/

References

Asynchronous Actions and Polling. (2020). Retrieved 27 May 2020, from https://docs.ansible.com/ansible/latest/user_guide/playbooks_async.html

"Ansible Documentation", 2020. [Online]. Available: https://docs.ansible.com/

AU_Govt_ABS_Census, "SA3-P13 Language Spoken at Home by Sex-Census 2016", 2017. Accessed from AURIN on May 26, 2020.

Banda, Juan M., Tekumalla, Ramya, Wang, Guanyu, Yu, Jingyuan, Liu, Tuo, Ding, Yuning, … Chowell, Gerardo. (2020). A large-scale COVID-19 Twitter chatter dataset for open scientific research - an international collaboration (Version 11.0) [Data set]. Zenodo. http://doi.org/10.5281/zenodo.3842180

Bird, Steven, Edward Loper and Ewan Klein (2009), Natural Language Processing with Python. O'Reilly Media Inc. https://www.nltk.org/

Cove Schneider, Joshua Conner, Pavel Antonov, Thomas Steinbach, Philippe Jandot, Daan Oosterveld, Chris Houseknecht, Kassian Sun, and Felix Fontein (2020) Docker_container – manage docker containers. [Online]. Available: https://docs.ansible.com/ansible/latest/modules/docker_container_module.html

"Docker Documentation", 2020. [Online]. Available: https://www.docker.com/

Felix Richter, "Only 34% of All Tweets Are in English," Dec. 16, 2013. [Online]. Available: https://www.statista.com/chart/1726/languages-used-on-twitter/

Fong, S., Wong, R., & Vasilakos, A. (2015). Accelerated PSO Swarm Search Feature Selection for Data Stream Mining Big Data. IEEE Transactions On Services Computing, 1-1. https://doi.org/10.1109/tsc.2015.2439695

Hutto, C., Klein, E., Pantone, P., Berry, G., & Suresh, M. (2020). nltk.sentiment.vader — NLTK 3.5 documentation. Nltk.org. Retrieved 25 May 2020, from https://www.nltk.org/_modules/nltk/sentiment/vader.html.

Lev Lafayette, Greg Sauter, Linh Vu, Bernard Meade, "Spartan Performance and Flexibility: An HPC-Cloud Chimera", OpenStack Summit, Barcelona, October 27, 2016. Doi.org/10.4225/49/58ead90dceaaa

"Loop", 2020. [Online]. Available: https://docs.ansible.com/ansible/latest/user_guide/playbooks_loops.html

"Reactjs Documentation", 2020. [Online]. Available: https://reactjs.org/

Richard O. Sinnott, Christopher Bayliss, Andrew Bromage, Gerson Galang, Guido Grazioli Philip Greenwood, Angus Macaulay, Luca Morandini, Ghazal Nogoorani, Marcos Nino-Ruiz, Martin Tomko, Christopher Pettit, Muhammad Sarwar, Robert Stimson, William Voorsluys and Ivo Widjaja. 2014. The Australian Urban Research Gateway. Concurrency and Computation: Practice and Experience, 27(2), 358-375. DOI: https://doi.org/10.1002/cpe.3282.

Quickstart. (2020). Retrieved 26 May 2020, from https://www.django-rest-framework.org/tutorial/quickstart/