Cluster and Cloud Computing Assignment 2
City Analytic on the Cloud

Zhaofeng Qiu 1101584
Aoqi Zuo 1028089
Rongbing Shan 945388
Tingli Qiu 990497
Yawei Sun 1050317

May 26, 2020

Abstract

This paper is a summary on the group project of COMP90024 Cluster and Cloud Computing at the University of Melbourne 2020 Semester 1, conducted by group #17. The use of social media has been increasing rapidly in past years, as a result, many useful information can now be provided by such social media, like Twitter. A cloud-based solution is introduced in the project to collect and analyze tweets from Australia based on six different scenarios, with parallel comparison with other Australian data gathered from *AURIN*. The results are visualized on a web portal for better illustration. The cloud-based system is deployed to Melbourne Research Cloud using software configuration management tool *ANSIBLE* and docker technologies.

1 Introduction

Social media has become an essential part of peoples daily life. The usage of Facebook, Twitter, Instagram and other social networks have been growing tremendously during the past decade. Taking Twitter for an example, based on statistics *from Internet Live Stats*[1], twitter now has over 330 million monthly users and there are over 500 million tweets sent each day. People nowadays share their activities, opinions, feelings and knowledge through social media, thus making the data valuable and informative. Despite the rich information given by the tremendous number of data, such large data volume also causes challenges for the analyzing system, since systems performance may drop heavily and failure rate arises with traditional methods. Therefore, high performance computing and cloud-based solutions come into our sight to harvest the informative big data from twitter, with good performance as well as improved disaster recovery and scalability.

In this project, we will develop an integrated cloud based system with the functionality of collecting tweets from twitter, managing the large-volume data in NOSQL database, analyzing the tweets data with comparison to AURIN statistics and visualizing the result on a web page. We have divided the system into several major function parts: Tweet Harvesters, Database, Tweet Analyzer, Backend and Frontend, and they will be discussed in detail in coming sections. Multiple technologies are used in this project. The major functioning body of the system was developed with Python and Java and would be deployed to Melbourne Research Cloud (NeCTAR Research Cloud) with Ansible. CouchDB has been used as a centralized database for data storage in a cluster mode. Also, docker technologies are taken for simple deployment and better program management.

In this report, we will introduce the design and architecture of our system, and go through each major functioning part. Besides, the five analytics scenarios we have developed from tweets data will be explained in detail. Also, we will address the challenges met during both the development and deployment phase of the project and discuss the error handling capability of the system.

[1] https://www.internetlivestats.com

- The scripts of our project can be accessed on github repository:
  - https://github.com/CCC-2020-G17/city-analytics-on-the-cloud
- And there is a video demonstration on the whole system through:
  - https://www.youtube.com/watch?v=LYTV-1LKQMo&feature=youtu.be
- The webpage can be accessed (with Unimelb VPN) through:
  - http://172.26.132.125

2 System Design and Architecture

The architecture of our system is well designed. It is aimed to provide both a beautiful front-end web application and a RESTFul API server with high availability, high scalability, and fault tolerance. The overall architecture is shown in Figure 1.

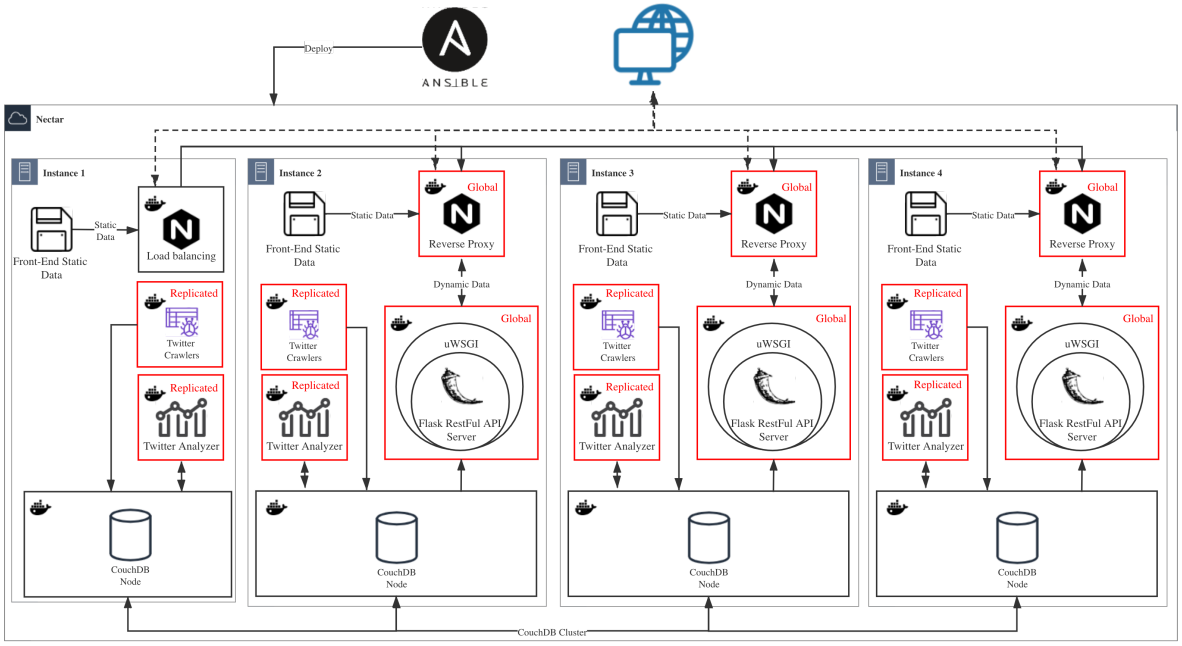

Figure 1: System Architecture

2.1 Availability

We continue to optimize our architecture through in-depth thinking and repeated practice to improve the availability of our system with limited resources.

Firstly, we run an *Nginx* server in our first instance to provide Load balancing. For requests that need to access the dynamic data in our database, it would pass the request to the other three servers averagely. In this way, we can efficiently distribute the incoming network traffic to our backend servers in the cluster. When it comes to the situation where the first instance may crash, users can still access our application through the IPs of the other three nodes. However, this solution is not good enough and is contradict to high availability. We had tried to use two *Nginx* server with a software called Keepalived to provide a more fault-tolerant system. However, we found out that we are not allowed to use floating IP in *Nectar*. Since we cannot set a virtual IP for our *Nginx* servers, the only way to handle this issue is to let the other three nodes to keep the full functionality of our application. Users can access our application through any IP of our instances, but only the IP of the first instance would provide load balancing control.

Secondly, we use *Docker Swarm* to manage part of our services. The containers which are in red boxes in Figure 1 are managed by *Docker Swarm*. The containers that with a "global" deploy mode setting would run in each of our machines except Instance 1. The containers with a "replicated" deploy mode setting would run in one of the four nodes. Actually, we have twelve *Twitter* Crawlers in our cluster, each for different purposes. They would be distributed to different nodes in our cluster by *Docker Swarm*. The *Docker Swarm* does not manage the *CouchDB* container and the Load Balancing *Nginx* server. But they can restart themselves if they crush since we set the "restart" configuration in docker-compose to "always". We didn't include the *CouchDB* into the *Swarm* cluster because using *Swarm* to manage the *CouchDB* database could be counter-productive if the *Swarm* manager deploys two *CouchDB* together to the same node. The Load balancing *Nginx* server is also not included because its port number is in conflict with the reverse proxy *Nginx* server in the *Docker Swarm* network. We set three *Swarm* manager nodes in our cluster. Any instance with an ID bigger than three would be set to be a worker node. A single leader is elected from the three *Swarm* manager nodes to conduct our orchestration tasks. Our manager would also run the services as worker nodes. According to [1] and *Swarm* Admin Guide[2], the number of our manager nodes is set based on the following facts:

- More managers would result in a longer election for a new leader when one goes down. The *Swarm* is in a read-only state which means that no new replica tasks can be launched and no service can be updates. No auto-recover can happen during the election.
- More managers would require tighter management of resources to prevent manager starvation.
- Fewer managers would increase the probability of all managers going down.
- To support manager node failures, the number of managers should be an odd number and should bigger than one. In our cluster, we can only provide four nodes. So, setting the number of managers to be three in our system is our best choice. In this way, our system can provide fault tolerance that two of our manager nodes can crash.

Thirdly, we use a *CouchDB* cluster with four nodes to improve our system's availability. We keep three replicas in our cluster, which means that any two nodes can be down in our system without crashing down our application. The Flask RESTFul API server, the *Twitter* Crawler, and the *Twitter* Analyzer of our system would only access the *CouchDB* database in their localhost. In this way, errors on a single node will not affect other nodes.

In conclusion, our system is a fault-tolerant system. Our application could run well even if two nodes in our cluster crashed. Except for the first node, any node in our system can provide all the services we need to run our front-end web application and RESTFul API server themselves. The system can be easily improved to a real high availability system by adding one more Load balancing *Nginx* server and using Keepalived to manage the floating IP system.

2.2 Dynamic and Static Separation

We use *Nginx* to separate static from dynamic traffic. For accessing the static data on our websites like HTML and image files, the *Nginx* server would handle them itself. For accessing the dynamic data from the database, the reverse proxy *Nginx* server passes the request to the uWSGI-Flask server through socket protocol. When using a socket, external browsers will not be able to directly access our uWSGI-Flask service, which can provide additional safety. It is faster to transfer through the socket. There are many advantages to separate static traffic. For one thing, it can help reduce the file I/O and storage consumption of our uWSGI-Flask server. *Nginx* is good at handling static traffic, which can provide a quick response to our front-end application. Further more, we can reuse the RESTFul API in our front-end application to reduce workload.

2.3 Scalability

We use Ansible to deploy our system. To add more nodes into the cluster or change some settings of the instance, we can change a few variables in our Ansible playbook and deploy them by a few commands. Instead of running all the playbook, we can run some specific tasks to save the deployment time since

[2]https://docs.docker.com/engine/swarm/admin_guide/

we also set the tags of our tasks on the Ansible playbook properly. Docker Swarm is used to providing self-healing and Rolling updates. The core docker images of our system have been published to Docker Hub. We can update our system by asking the Docker Swarm to pull the new images and use a rolling update to deploy a new version system. The Ansible playbook is well designed, using many new features of Ansible. We use many variables in our Ansible playbook to provide a better scale-up experience instead of hard coding.

2.4 Dynamic and Static Separation

We use *Nginx* to separate static from dynamic traffic. For accessing the static data on our websites like HTML and image files, the *Nginx* server would handle them itself. For accessing the dynamic data from the database, the reverse proxy *Nginx* server passes the request to the uWSGI-Flask server through socket protocol. When using a socket, external browsers will not be able to directly access our uWSGI-Flask service, which can provide additional safety. It is faster to transfer through the socket. There are many advantages to separate static traffic. For one thing, it can help reduce the file I/O and storage consumption of our uWSGI-Flask server. *Nginx* is good at handling static traffic, which can provide a quick response to our front-end application. Further more, we can reuse the RESTFul API in our front-end application to reduce workload.

3 System Functionalities

In this section, we will focus on explaining the tools that we use to implement each system functionality and the methodology of each implementation together with the reason we choose to use it. There will be some figures to support the explanation of the functionality.

3.1 Twitter Harvest

Twitter mainly provides two types of APIs for gathering tweets, Search and Streaming APIs. The search API supports developers to gather tweets posted in the past. The streaming API captures the real-time tweets. To access the APIs, we create python programs to harvest the tweets by using the *tweepy* library. Our harvester makes use of both APIs to gather as many tweets as possible.

Meanwhile, we applied harvesters and *CouchDB* on the same instance to ensure that tweets would be delivered and stored into *CouchDB* directly. Our system allows multiple harvesters and *CouchDB* to run concurrently. As our analysis focuses on five cities in Australia, Melbourne, Sydney, Adelaide, Brisbane and Perth, we deploy twelve harvesters. Each of the five cities has two harvesters to gather past and real time data respectively. Besides, two more harvesters gather data in the whole Australia as complementary, which makes our harvesters more robust. These harvesters are randomly assigned to four instances. The architecture for twitter harvester in the cloud is deployed as Figure 2. And in the development phrase, we gathered nearly 8 million tweets for analysis, with ten percent contains specific coordinates.

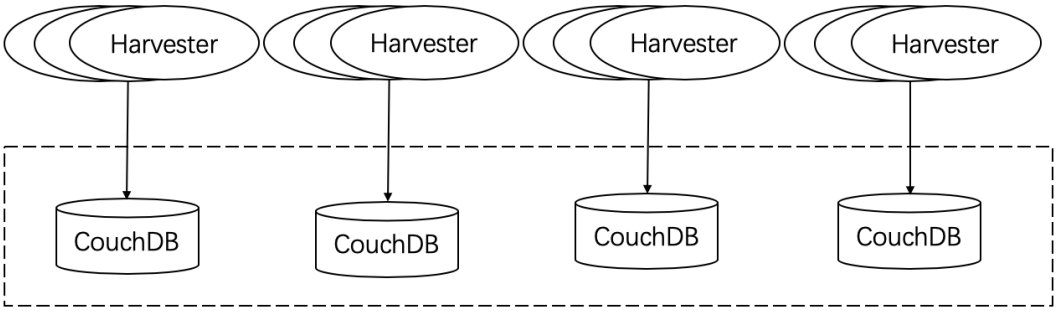

Figure 2: the Architecture for Twitter Harvester

3.1.1 Search API

Twitter Search API is a part of REST APIs. It is retrospective and will capture tweets in the last 7 days. We can query and gather tweets for a specific location, a specific time period or with certain interesting keywords through Search API. In this part, we specify location by the city bounding box, which is represented by the range of longitude and latitude. As each tweet has the unique tweet ID, the duplicate tweets are easy to recognize and cannot be stored in the *CouchDB*, of which the details will be explained in the data processing section.

There are some limitations of mining twitter content when using the Search APIs:

- The first limitation is that for per-user authentication, one can only send 180 requests per 15 minutes with the maximum of 100 tweets for each request, resulting in a total limit of 18,000 tweets per 15 minutes. In order to improve the efficiency of harvesters, they need to maintain the upper limit allowed by the APIs.
- The second limitation is that it will not be able to get all the tweets that match search criteria, even if they are present in the desired time slot because not all of the tweets will be indexable or made available.
- The third limitation is that the Search API focuses on matching the search condition (relevance) rather than all the eligible tweets (completeness) as some users and tweets will be missing during the usage of Search API [3]. But Streaming API is match for completeness, which will be introduced in the following subsection.

To overcome these limitations and improve the efficiency of harvesters, we come up with several solutions:

- In the development phase, we get *friends_id* of users and gather tweets for each user and their friends. This method helps us capture most tweets as the request limitation per 15-min window for per user authentication is 900 and request limitation per 24-hour window is 100000 with up to a maximum of 200 tweets per distinct request. Therefore, we can obtain up to 180,000 tweets per 15 minutes, which is ten times of the common Search API. In the implementation, we gather tweets from users oldest tweet ID available by the method *user_timeline()* and record the earliest crawled tweet id, which can be the value of the parameter *since_id* for next harvest iteration. In this way, we can gather substantial different tweets.
- Besides, in order to crawl as many tweets as possible, we make full use of all consumer and token key pairs of six accounts. Instead of harvesting tweets in the whole Australia by one account and having to sleep when they hit the limit very soon, we harvest tweets in each city by one authentication account. Therefore, it takes a long time to hit the limit and more data can be gathered. This method makes the system easier to deploy and manage as well.
- Moreover, another strategy we tried is to replace the authentication account with another whenever it hits the limit. However, this makes the deployment very complex, confusing and did not make good use of our parallel systems. Therefore, we discard this strategy when deployment.

3.1.2 Streaming API

Compared to the search API, Streaming API can give developers a lower-latency access to twitters global stream of tweet data [4]. Also, Streaming API has a persistent HTTP connection from client side to server side so that its a real-time delivery and can ensure the completeness of tweets as it can gain more tweets than the search API. For example, the Search API works in a request-reply (RR) way. However, the streaming API is different in the way the server responds. The server would send a response to the client whenever a tweet is available, therefore, a continuous stream of responses will come to the client from the server and the message will appear into the stream immediately with minimal overhead. And in this part, we specify place by the city *place_id* in tweets.

Unlike Search API, Streaming API does not limit the amount of tweets harvested per time slot and it

[3] https://developer.twitter.com/en/docs/tweets/search/overview/standard
[4] https://developer.twitter.com/en/docs/tutorials/consuming-streaming-data

supports collecting the complete intraday tweets [5]. Therefore, for intraday tweet data, the Streaming API will theoretically capture more valid tweets than the Search API. Nonetheless, these tweet data are only the fraction of numerous eligible tweets.

3.2 Data Processing

In this section, the data processing steps to support the analysis scenarios will be discussed.

3.2.1 Architecture

Our analysis system generally follows the common pattern of data analysis: gather data -> save data -> analyze data -> present data. The architecture shows in Figure 3. The data processing part supported by *CouchDB* then connects the Tweet harvesters, the analyzer module, and the backend together as an integrated system. Harvested tweets will be cleaned up, transformed, and saved to the database. Obtaining the data from the database by using views filtered through *MapReduce*, the analyzer is able to further generate an analysis result for the given data, and save the result or update the existing result to the analysis result database. The backend fetches the result to the frontend for presenting.

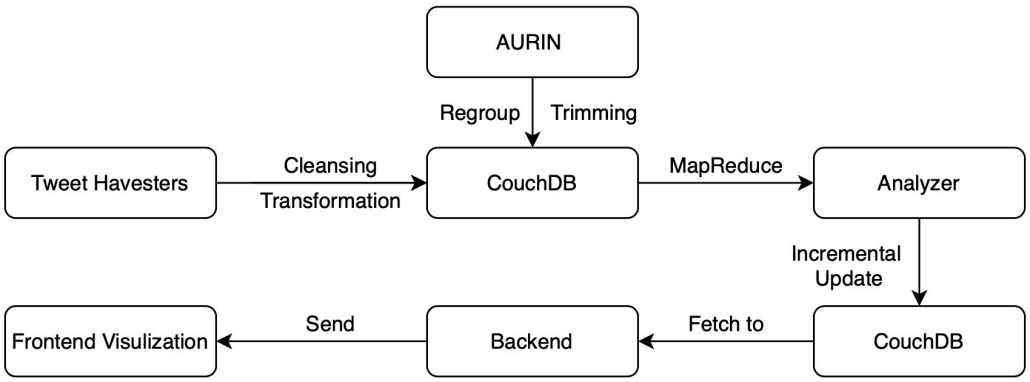

Figure 3: the Flow Chart for Data Processing

Whats more, to have a stable and reliable system, we have implemented an Incremental Update algorithm. During the development phase, as the data volume grows to 10 million levels, the performance and stability of the system starts dropping if the data is analyzed as a whole. But with an incremental update system, each time we will only extract the latest, not-yet-analyzed data from *couchdb* based on timestamps, to the analyzer, and update the previous analyse results. In such a way, there will be no need to analyze billions of tweets every time, hence improving the performance and stability of the system. Whats more, with such architecture, more flexibility is brought up to the deployment phase, as we will only need to migrate the analysis results from the development environment but not all the tweets stored. With the help of limitations on twitter API, which allows collection of only the latest data (no more than 7 days), our analysis system will be able to avoid duplicated results and work properly.

3.2.2 Data Cleansing and Transform

Before saving raw data from twitter and *aurin* into *couchDB*, we will go through a data cleansing process to filter out unnecessary data and transform the data into aligned format, which will be easier for further processes.

Twitter Data:

In the architecture, we will collect tweets continuously from twitter API, however, the transformation of twitter data was relatively simple, as twitter has already processed its data and all data collected from

[5] https://developer.twitter.com/en/docs/tutorials/consuming-streaming-data

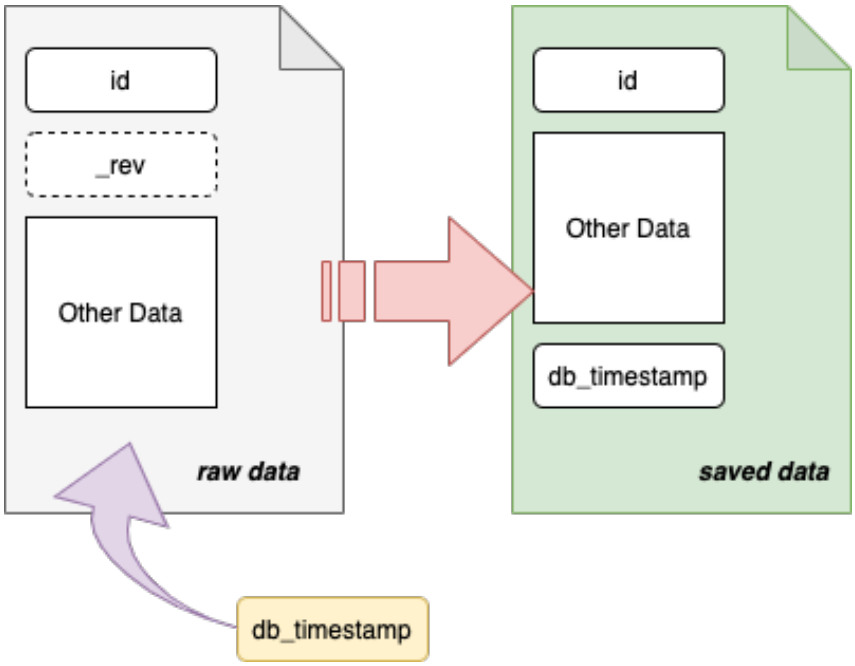

Figure 4: The Transformation of Twitter data

twitter are ready in a synchronized format. We will extract the twitter ID from the data as the unique key for storage in *couchDB*. And since there is a tweet corpus provided to the project, which means that the data are not crawled directly from twitter, we will also check if the data has _rev, the revision reference in *couchdb*. If _rev exists in the data, we will simply remove it before saving to avoid update conflicts. A code demonstration is shown in the figure 4. Also, with consideration on the incremental update algorithm with our design, we will need a timestamp before inserting the twitter data into our database. Hence a field named db_timestamp will be inserted together with each tweet stored, as illustrated in figure 4. A database named tweets_mixed is created to hold all twitter data crawled and extracted from tweet corpus. Then the data can go through MapReduce and be analyzed in further steps.

AURIN Data:

AURIN data is quite different from twitter data since it is not updated very often and in our project, we will not update the data from AURIN regularly. In order to make analysis based on different suburbs, we will collect multiple data sets with suburb information from AURIN. The data formats are not quite desired as they are hard for further process, hence we will re-group the information based on suburbs and extract the suburb names as keys for other information under this suburb, as shown in figure 5 Also, some unnecessary information in the AURIN data will be removed. The data from AURIN will be saved to database aurin, and like other data, a timestamp will be inserted. But unlike tweets, we can always update a same document for AURIN, the data wont be discarded like with tweets, but instead, the data will be updated and a latest timestamp will be inserted.

Error Handling:

There are several problems met during the ETL process in both development and deployment phase and they will be addressed here.

- **Duplicate tweets**: The major problem we would need to address will be the duplication of data from twitter harvesters. Both in development and deployment, there are multiple duplicate tweets collected. To address the problem, Tweet ID is used as an unique identifier (like surrogate key in relational database) to distinguish among these data. If the crawled tweet ID is already in our database, we will get an update **Resource Confict** from *couchDB*, which means the data is duplicated and the data processor will ignore this tweet. Also, such an algorithm also guarantees that the incremental update of our architecture works correctly: all processed tweet IDs will not be handled twice as their timestamp wont be changed if they have been saved to our database. We implement

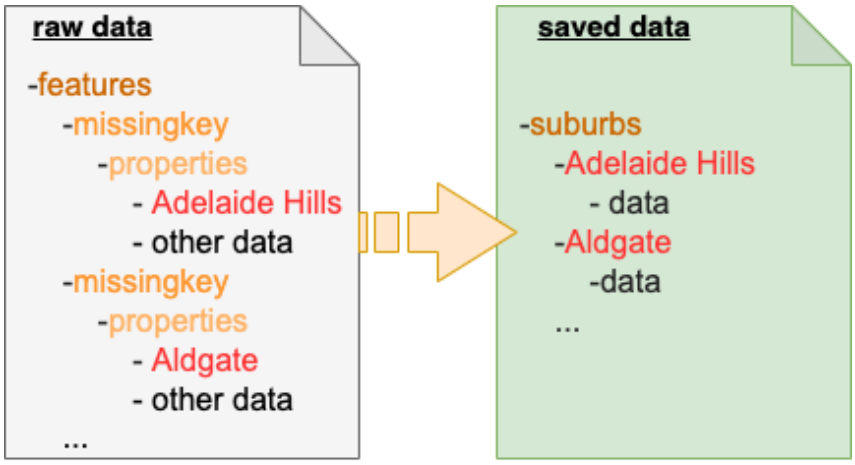

Figure 5: The Transformation of AURIN data

UPDATE FUNCTION

INSERT_TS

```
function(doc, req){
doc["db_timestamp"] =
(Date.now().toString()).substr
ing(0,8)+'00';
return [doc, "OK,timestamp inserted!"]
}
```

Figure 6: Update function for inserting timestamp

this based on the assumption that twitter wont update its data if the tweet is already saved into their system, so we wont have any data loss.

- **Timestamps update**: During the development phase, we firstly do not have a timestamp of our database system in the beginning. All the tweets collected will be processed in one time with Map Reduce. However, with the increment of data volume, the performance starts dropping quickly and we came up the idea of incremental analysis, which requires a timestamp of our own system. *CouchDB* update functions in figure 6 are utilized in order to update tweets that are already saved in the database without a timestamp. With execution of the update function insert_ts, all existing tweets in our database will be inserted with a timestamp.

3.2.3 Data Manipulation using *CouchDB* functions

After the twitter data is saved into the database, we will need to do some research on the data and also extract the data. Here we will utilize different *Couchdb* built-in functions to study the data and extract the data.

To study the data at an early stage, we mainly relied on the text search function [6]. For example, at the very beginning, we were trying to make an analysis on most popular car brands with their most mentioned factors mentioned in tweets, and their relations to the poor and wealthy suburbs in Australia. We have used text search functions in *Couchdb* to search for different brands. However, as twitter limited us to collect only data for the past 7 days, it turns out there are very few tweets mentioning automobiles. So we have abandoned the scenario, but *CouchDB* built-in functions have indeed made the study of data easier.

[6]https://docs.couchdb.org/en/master/ddocs/search.html

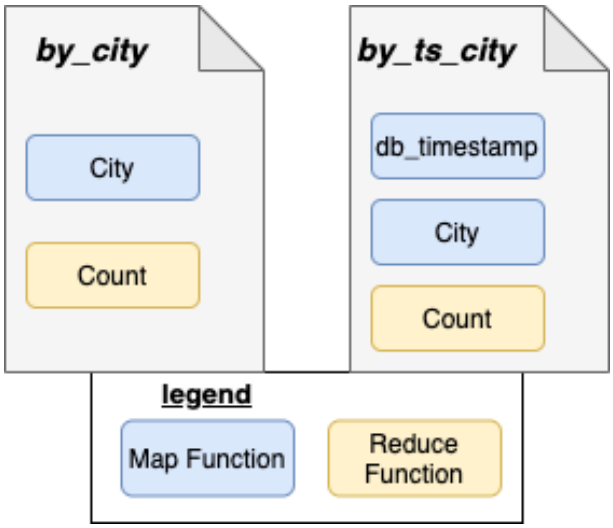

Figure 7: Function structure of MapReduce views

To extract the data in later stages, we have turned into MapReduce views [7] in *CouchDB*. Although with the previous data processing steps, we have limited the tweet data we collected in some way, however, there are still many redundant data that is not fit for our scenarios. We kept these data in the database with consideration on further usage, but in the current stage, we would not feed such data into the analysis phase. But it will be very much complicated and resources consuming if we are going to filter out such data one by one. Also, as introduced in section 3.2.1, our analyzer system uses an incremental update method. So for each epoch, only the latest collected tweets that have not been analyzed will be extracted and passed to the analysis module. MapReduce function in *couchDB* will be used to make life easier. Besides, the views created with MapReduce will also assist in the analysis process.

There will be two views created in the system to assist in the analysis process, kes of whome demonstrated in figure 7. The fist one is named *by_city*, which is used to extract data by city name and count the number of tweets within the specified city. This view will be used mostly to inspect data and in calculating ratios with the total number of tweets in a city.

The other one is named *by_city_ts*, which can extract tweets of a certain city by the timestamp that it was inserted into our system. This is the main view we will use in our incremental update analyzer module, where we will get only the latest data that is not analyzed before, determined by the timestamp. Also a count reduce function is implemented with this view to get statistics on the data processed in each increment.

3.3 Analyzer Module

The analyzer module supports six scenarios which include three city-level scenarios and three suburb-level scenarios, and the details of each scenario will be introduced in the Analytic Scenarios part of this report.

3.3.1 Analysis Result Structure

The analysis results for the scenarios are saved in the database for the frontend further presenting. The structure of the result contains three layers and can be load from result.structure.cfg configuration file. The first layer contains the city name, city tweet counts, and keys for the second layer values. The second layer contains the city-level analysis and the suburb keys. The third layer contains suburb-level analysis.

[7] https://docs.couchdb.org/en/stable/ddocs/views/intro.html

3.3.2 Static Result Manager

The job of the static result manager is handling the data obtained from AURIN. Theoretically, this part of the analysis result will not be changed after the first time it is loaded to the database. The AURIN data is stored in the database according to different scenarios and different cities. The manager fetches data from the database, picks the valuable data chucks and fills them into the result structure. The manager is also able to decide which parts of data should be load.

3.3.3 Dynamic Result Manager (Tweet Analyzer)

The tweet analyzer has two modes, one is generating analysis results from the given historical tweets, and one is non-stop analyzing newcome tweets and updating the existing result. While the whole system is running on the cloud, the mode used is the second one.

As more tweet data coming into the database, it is necessary to make sure that the analysis result is up-to-date. Therefore, the analysis result in the database should be dynamic. The way to achieve that is running the tweet analyzer every period t (which now is set to be 1000 seconds), and each time the analyzer will get the newcome tweets from the past period t and analyze them. After analyzing, the tweet analyzer will update the existing analysis result in the database, and sleep until the next wake up. The tweet analyzer also has a logger that logs whether the analyzer is running or sleeping, the timestamps of the tweets have been analyzed and any potential errors. The group member in charge of the server will, therefore, be easily monitoring the status of the analyzer.

There are three main functionalities designed for city-level scenarios. The first one is identifying whether the tweet mentions COVID19, and if it does, then get the number of followers that the tweet poster has. The analyzer will check if the keywords like COVID or coronavirus appear in the tweet text, hashtags, or links. Follower counting is done by using Twitter API *get_user().followers_count*. The second one is identifying whether the tweeting time is at night and whether the location label is enabled. Transforming UTC time in tweet JSON to AEST and finding the coordinates value of key geo are the main steps. Lastly, counting the number of unique users and therefore calculating the tweeting density, which is the number of tweets per user.

In terms of suburb-level scenarios, the analyzer will first try to match the tweet to a specific suburb based on the coordinates of the tweet and the boundaries of the city suburbs. It is done by using Polygon and Point classes in package shapely.geometry. The analyzer uses package vadarSentiment to identify the sentiment of a tweet, and the sentiment can be positive, negative, or neutral. The analyzer uses package profanity for searching profanity in the tweets text. The tweeting language is simply identified from the value of key lang in tweet JSON.

3.3.4 Database Connector

Database Connector contains two main parts, data loader and analysis result saver. The data loader helps the other parts of the analyzer module load different data according to different needs. Loading Twitter data by timestamps or by cities, loading the city map coordinates, and loading AURIN data for certain scenarios on certain suburbs are examples of what the data loader is able to do.

Analysis result saver is able to simply replace or incremental update the existing analysis result with the new result. The reason why the result saver has this design is the thinking of incremental computing. It is necessary to consider the efficiency and performance of the system since we are doing a cloud computing project like this. The analysis is on big data, although our analysis is only based on millions of tweets, it would be unrealistic to analyze all the data at the same time when the data amount grows to a certain level.

Dynamic Updating works on the basis of MapReduce algorithm and recursive functions. After generating the analysis result for the data filtered through MapReduce, the result saver will also fetch the existing result, then check each key in each layer of the result structure, to see whether it needs to be updated or should keep it as it is. Like for numbers related to tweets, most of the time it needs to add the renewal to the existing ones. As for strings or static results, however, we may want them to remain the same.

Analysis result saver is also to reset any part of the analysis result, in case theres a data error or something. It is also in line with the thinking of incremental computing because we dont want to analyze all the data

gain if only some parts of the result are affected by dirty data. Just reset those parts of the analysis result to let people know that they are affected and not available, and then wait for the correct version to come.

3.4 RESTFul API server

We provide RESTFul APIs for accessing our analysis. We use an Nginx-uWSGI-Flask framework to build our RESTFul API server. They are all running in Docker Containers and are managed by Docker Swarm. The uWSGI-Flask service is listening on port 3000, and the Reverse Proxy Nginx server would pass the request to the uWSGI-Flask service through socket protocol. The uWSGI-Flask service would only access the CouchDB on their localhost. In this way, the crash of one CouchDB would not affect other nodes. Moreover, a load-balancing Nginx server is running in Instance 1 to improve our system's availability. Details about the architecture are shown in Section 2.

Our RESTFul APIs provide six types of data accessing for five cities, including both city-level and suburb-level analysis of Melbourne, Sydney, Brisbane, Adelaide and Perth.

Our Web Application also uses these RESTFul API to access the dynamic data. By separating static from dynamic traffic, the file I/O and storage consumption of our uWSGI-Flask server reduces.

3.5 Web Application

The web application is composed of front-end and back-end. We use VueJs framework to implement data visualisation and Flask framework to provide APIs. The server will receive the requests from users browser and then return corresponding JSON data fetched from CouchDB. With unimelb VPN, you can access our web application with http://172.26.132.125/.

Our front-end web application is composed of HTML, CSS and Javascript.Many tools and plug-ins are used to make our presentation clear and easy to understand. Google Map provided by google API is selected as our base map. We also fetch the GeoJson file of city and suburb from Aurin APIs. And the socio-economic features we got are added into those GeoJson files. As a result, it is easy and fast to present the layer of city and suburb map combined with the corresponding analysis result at the top of the base map.

Numbers of coordinates form the boundary of the city and its suburbs. And the shades of the color of those polygons present the distribution of varieties of socio-economic attributes. Moreover, we use pie charts and bar charts to represent and compare data by ChartJs.

4 Data Visualization and Analytic Scenarios

The Scenarios are across five main cities in Australia, Melbourne, Sydney, Brisbane, Adelaide, and Perth. There are three city-level analysis and three suburb-level analysis.

4.1 City-Level Scenarios

4.1.1 The First City-Level Scenario

This scenario is the relationship between the number of tweets that mentioned coronavirus and the number of followers those tweets posters have. That is, are the users with more followers more likely to mention coronavirus in their tweets? There are reasons for us to come up with this scenario. Users with more followers usually have a greater sense of social responsibility than normal people, so they may care more about the virus, which is affecting everyone. Also, users would draw more attention from the public if they talk more about hot topics. Well, theres always a reason that someone is the one with more followers. As a result, we can see that our assumption is correct. A large proportion of the tweets mentioned coronavirus is from twitter users who have more than 500 followers, but only a few of them were tweeted by users who have less than 100 followers. The situation in each city is basically the same. The analysis bar chart for this scenario is as Figure 8.

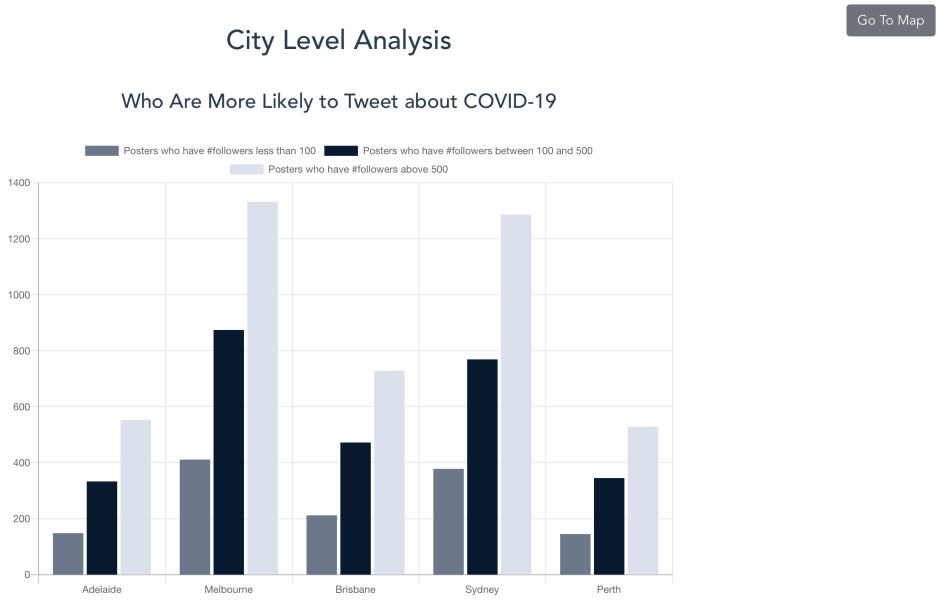

Figure 8: Scenario 1 in city level analysis

4.1.2 The Second City-Level Scenario

This scenario is the young people's preferences when tweeting. We would like to see whether a city has more night tweets and located tweets if the city has a larger proportion of young people. By young people, I mean people aged between 15 to 35. And night tweets are the ones that were tweeted between 11 pm to 4 am. We made a guess that young people are more likely to stay up late, and young people would love to share where they are when they tweet, to show some nice food or some fancy activities, or stuff like that, so they have their location label turned on. But the result shows that there no such relationships. It might be explained by the fact that we are in a very special period, so people dont want to share their locations because they have nowhere to go. And Twitter might not be the night social media choice for young people. Because we also have Instagram and Facebook etc. From the plot 9, we can see that Adelaide is the city with the biggest proportions of located tweets and Perth is the city has larger night tweets contribution.

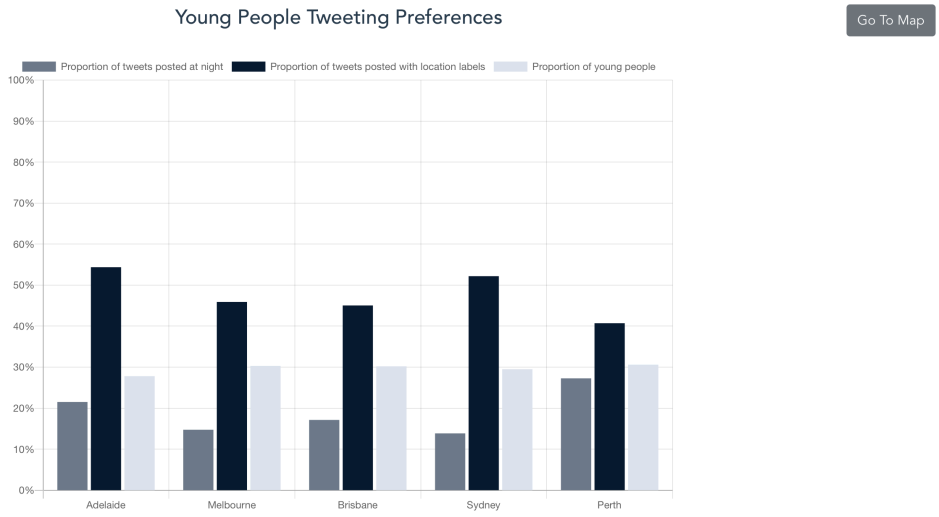

Figure 9: Scenario 2 in city level analysis

4.1.3 The Third City-Level Scenario

This scenario is tweeting frequency versus the proportion of people who speak English as their mother tongue. I believe that most people in Australian cities do speak English at some level, but it looks like there are still many people who dont use Twitter. I know many people whose English is good enough to tweet frequently, and do have the Twitter app on their phones, but they are not used to tweet. Therefore, we made a guess that theres a preferred social media for each culture, and Twitter may not always be the one for cultures other than English culture. However, the result shows that cities with a lower proportion of native speakers have a relatively high tweeting density instead. For example, Melbourne has only 62% of people whose mother tongues are English, but there are about 28 tweets per user in Melbourne. The reason might be that the more multicultural the city is, the more interesting things to tweet. Collisions between cultures always bring interesting stories. The plot 10 demonstrates this scenario.

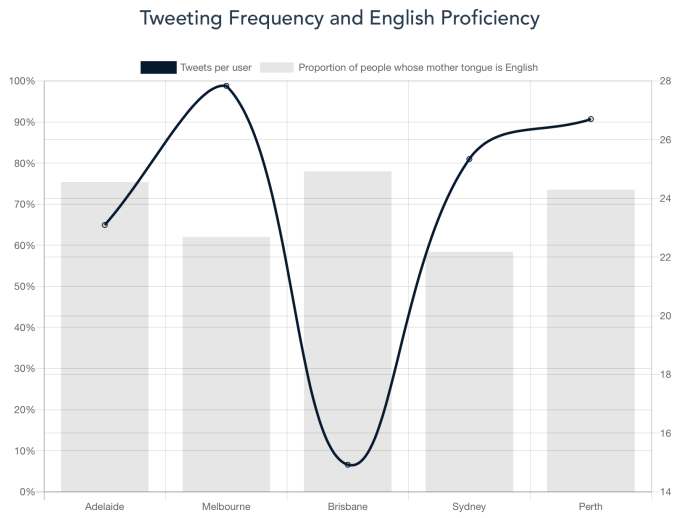

Figure 10: scenario 3 in city level analysis

4.2 Suburb-level Scenarios

Within each city, we have gone into details and made some analysis for the suburbs in that city. For this part, we add some pie charts to the map. Whenever a suburb is clicked on, some pie charts will pop up. Each pie chart shows the distribution of each level for the corresponding attributes.

4.2.1 The First Suburb-Level Scenario

This scenario attempts to find the relationship between the proportion of income level and the tweets sentiment polarity in some city. The reason why we have this scenario is that we want to see whether people have a greater income will post more positive tweets. The color on the map in Figure 11 represents the income level for the suburbs. The darker the color is, the higher the average income the suburb has. Here, the tweets sentiments polarity is represented as the ratio of the number of positive tweets to the number of the negative tweets. Therefore, the greater the ratio is, the more positive the emotion is.

Lets especially mention Melbourne city. In the map, clicking on the suburb Parkville, we can see two pie charts showing the tweets sentiment distribution and the income level distribution in Parkville respectively. Moreover, the bar chart in Figure 12 shows the horizontal comparison across suburbs in Melbourne. Each suburb has two bars, one is the proportion of high-income people and the other one is the ratio of number of positive tweets to number of negative tweets. We can see that on the whole, the suburb with a higher proportion of high income tends to be more positive, except for some small fluctuations, like the suburbs Melbourne and Carlton. It's interesting to see that the average income is not high for these two suburbs, but the proportion of positive tweets is relatively higher than other suburbs largely. The reason we think

may be there are many students in Melbourne and students are generally more positive and optimistic. This relationship is also similar for other four cities.

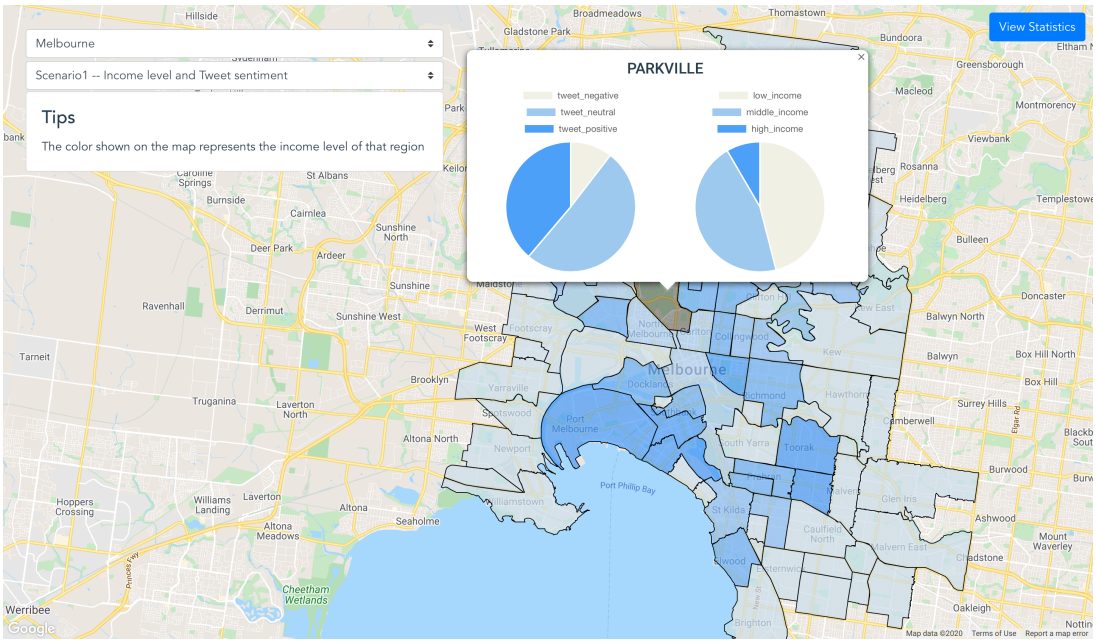

Figure 11: the Pie charts in Scenario 1 for the suburb Parkille

4.2.2 The Second Suburb-Level Scenario

This scenario is to find if there are some relationships between the education level and the proportion of tweets containing vulgar words for each suburb in some city. Lets take the city Sydney as an example. In the map in Figure 13, the darker the color is, the higher vulgar tweets proportion is. Clicking the suburb Sydney, we can see there is a pie chart showing the proportion of people who have completed 12 years of education and not completed. Another pie chart shows the proportion of youth who are in study or work and who are not. These pie charts give us more details about the suburb. Its obvious from the bar chart in Figure 14, that on the whole, the suburb with a higher proportion of people with low education achievements tends to have more tweets with vulgar tweets. Similar for other cities, such as Adelaide and Perth.

4.2.3 The Third Suburb-Level Scenario

This scenario is to detect if there is some relationship between the proportion of non-English tweets and proportion of people who don't speak English at home in a city. For instance, in the city Perth, in the map in Figure 15, the darker the color is, the higher the non-English tweet proportion is. Clicking the suburb Subiaco, we can see there is a pie char showing the proportion of people who speak English at home and people who do not. Another pie chart shows the proportion of people born overseas and born in Australia. From the bar chart 16, we can see that in general, the higher the proportion of people who dont speak English at home, the higher the proportion of non-English tweets, except some suburbs, like Mount Claremont and Dalkeith. In Dalkeith, there is a higher proportion of people who dont speak English at home, but a relatively lower proportion of non-English tweets, which is really weird and there are may be some reason for that. Similar for other cities, such as Sydney and Brisbane.

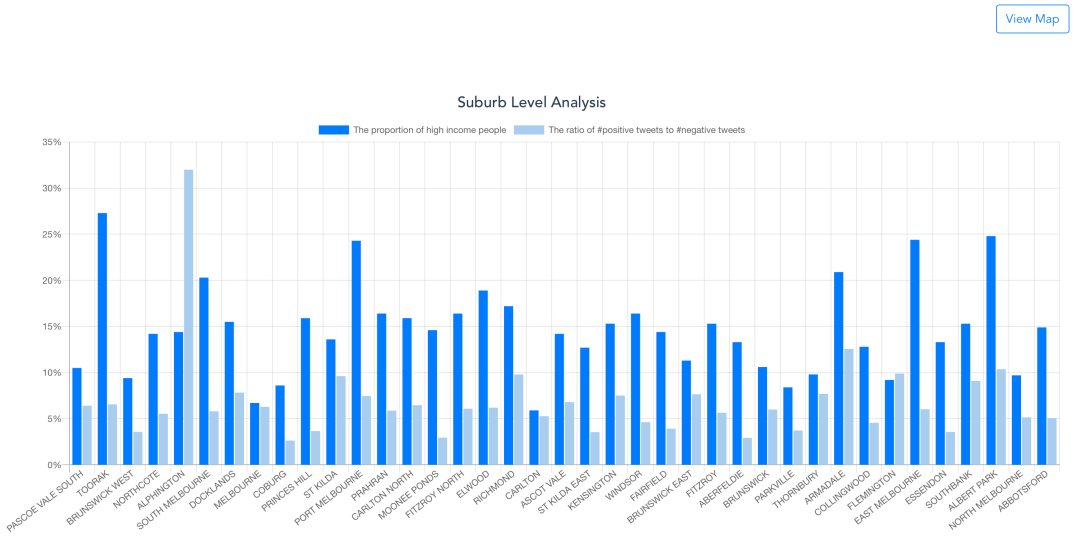

Figure 12: the Relation between Income Level and Tweet Sentiment in Melbourne city

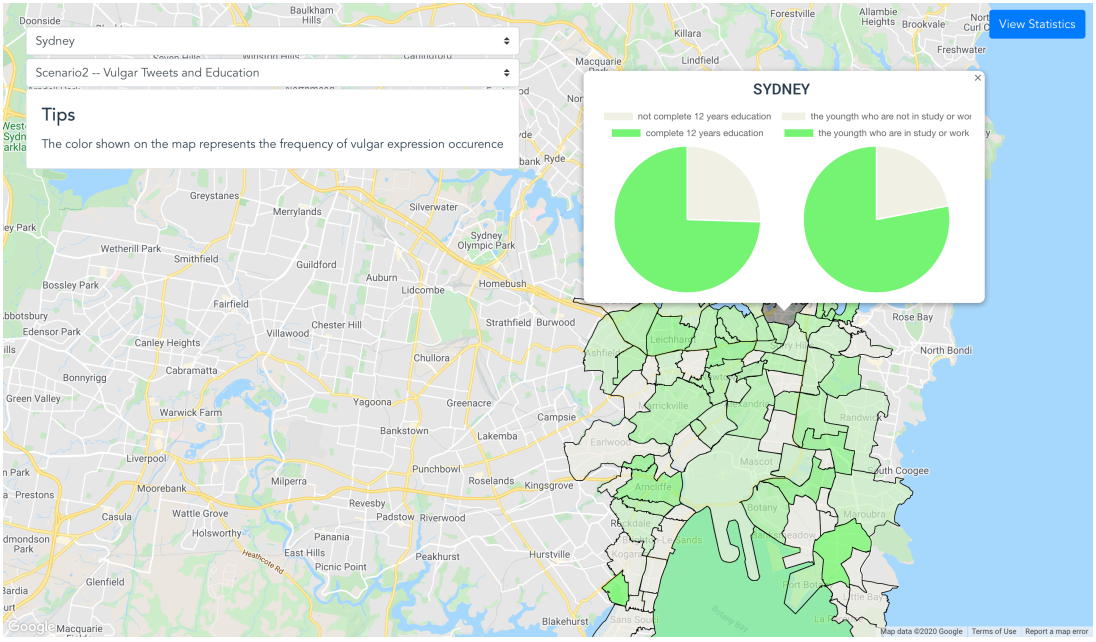

Figure 13: the Pie charts in Scenario 2 for the suburb Sydney

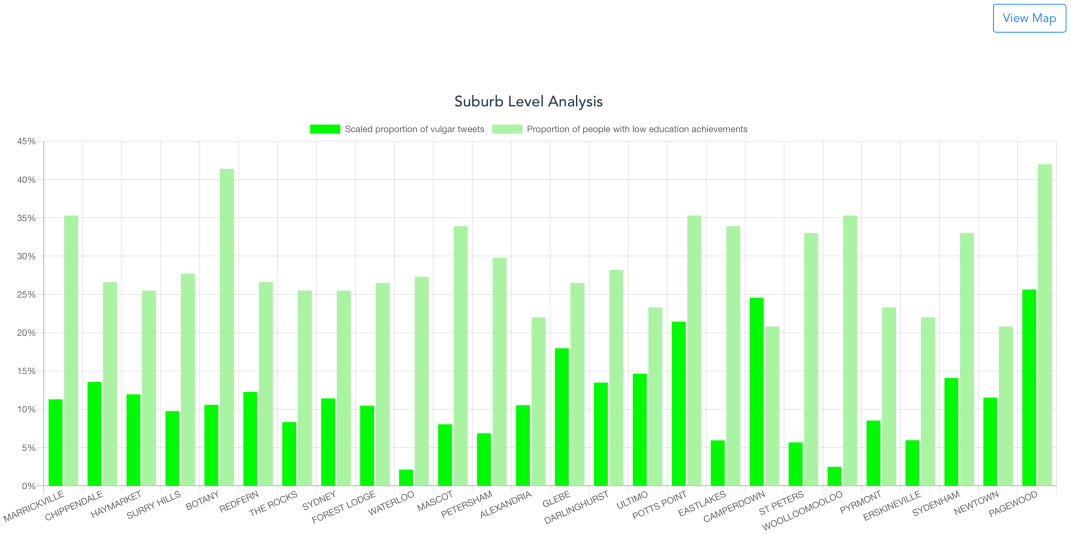

Figure 14: the Relation between Vulgar Tweets and Education Achievements in Sydney city

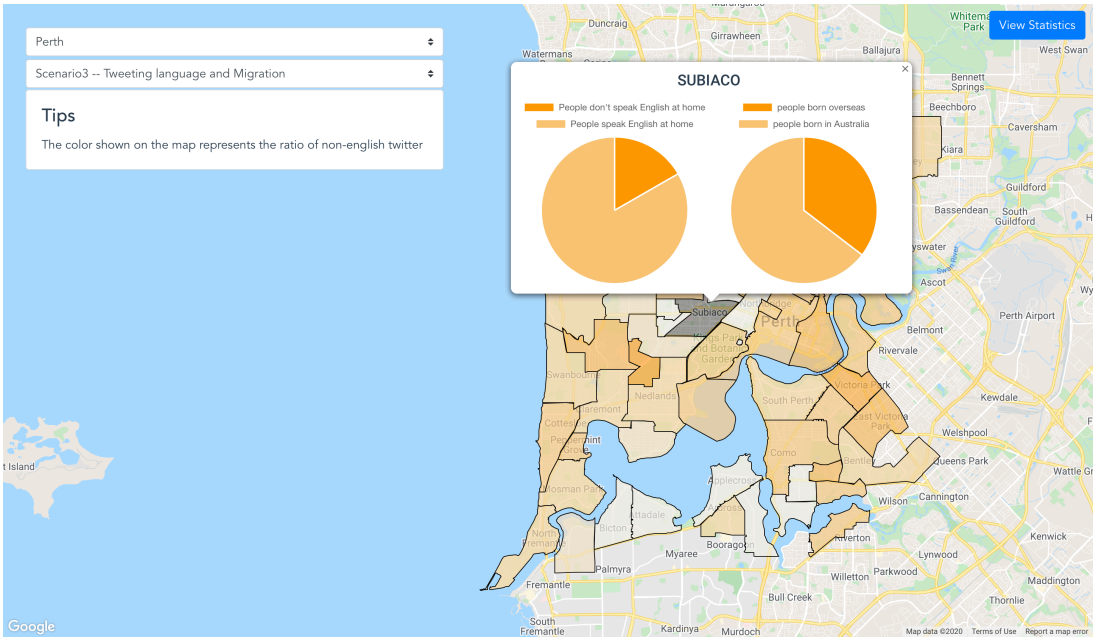

Figure 15: the Pie charts in Scenario 3 for the suburb Subiaco

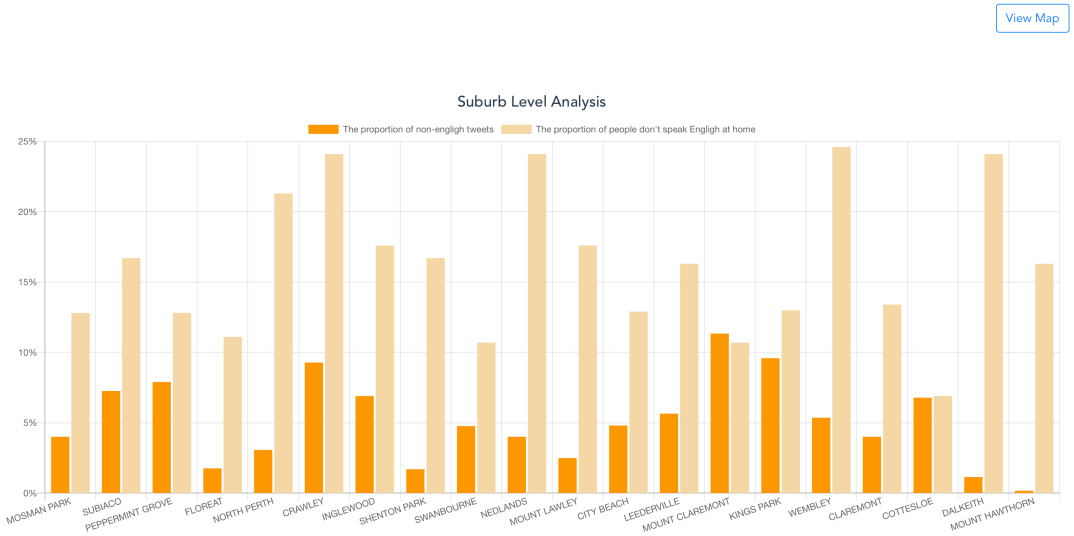

Figure 16: the Relation between Non-English Tweets and the Proportion of people who don't speak English in Perth city

5 UniMelb Research Cloud

5.1 Pros

- Using the IaaS provided by the UniMelb Research Cloud, we don't have to buy our own machines, which can reduce infrastructure costs. The cloud can provide virtually limitless scalability, which can help us scale our services based on the market. Using a cloud to deploy our system can not only save money but also save time. The consolidated disaster recovery infrastructure provided by IaaS is also essential.
- The UniMelb Research Cloud is based on OpenStack, which is an open-source cloud technology. Many tools provide support for OpenStack based cloud. For example, Ansible provides modules for controlling OpenStack based cloud. It is straightforward to deploy systems through those Ansible modules.
- The User Interface of the UniMelb Research Cloud is user-friendly. The system is easy to use. All the information we need to know about the instance is listed clearly on the website.
- The network of UniMelb Research Cloud is fast. It cost a little time for the virtual machines to download dependency from the Internet, which can help both development and deployment.

5.2 Cons

- It is hard to find the document of the UniMelb Research Cloud. The document cannot be easily opened on the web page.
- The front-end interface is missing correct error feedback. Sometimes it is tough for us to understand the error. For example, if you accidentally create a snapshot for one of your volumes, you would not be able to delete the volume. But the system would only tell you the delete is not allowed without any reasons.
- No forum is provided. The only way to find a solution is to ask questions on the discussion board. If there is a forum where history Q&A can be accessed, it will help developers a lot.
- Web page loads slowly. It takes time to check the condition of the machine.
- Not easy to learn. There are no official beginner tutorials for the cloud. Without watching the subject's workshop, it is tough to develop on it. An understandable document for quickstart is needed for reducing learning costs.
- The proxy setting is very annoying. Not like running in a local machine, we have to configure the proxy for all the software we need. And sometimes you have to set it twice for some software like the Docker Swarm. For the container running in the docker swarm, not like running directly through docker, you need to set up the proxy one more time to help services connect to the network.
- No floating IP is provided to students. Without a floating IP, we cannot develop a High Availability server. We tried to use Keepalived + Nginx to improve the availability of our system but finally found out that no floating IP is provided in our project.

6 User Guide

Our project is available on *https://github.com/CCC-2020-G17/city-analytics-on-the-cloud*. For more details about the usage, please access *README.md* in the GitHub repo.

6.1 Prerequisites

The VMs we use are provided by UniMelb Research Cloud (Nectar), which is based on OpenStack. And the Ansible playbook in ansible/playbooks mainly use OpenStack modules to create machines on the Instance Initialize Process. The following setting is based on UniMelb Research Cloud. You can change the code inside based on your cloud suppliers.

- Gain access from cloud providers:
  1. Login to *https://dashboard.cloud.unimelb.edu.au*.
  2. Download openrc.sh from Dashboard.
     - Make sure the correct project is selected
     - Download the OpenStack RC File
  3. Reset API password
     - Dashboard -> User -> Settings -> Reset Password
  4. Replace the *ansible/openrc.sh* with your own one. We suggest changing the *OS_PASSWORD* to your API password so that you don't have to input it every time.
  5. Generate a ssh key pair in the cloud and keep the private one on your own computer.
- Install Ansible.
- Set the Ansible variables to your own.
  - Open *ansible/playbooks/variables/nectar.yaml*
  - Change the *instance_key_name* in *line 30* to the name of your ssh key.
  - Change any other variable based on your personal need.

6.2 Deployment

If you set up the Prerequisites properly. The deployment would be very easy. We use Ansible to deploy our system. We also provide some script to simplify the command. But before running any command, if you are using the UniMelb Research Cloud, you need to connect to the Unimelb VPN.

You need to first enter the *./ansible* dir and give permission to our script. Run the deployment with command: *./runAll.sh*.

6.3 RESTFul API Usage

We provide RESTFul APIs for accessing our analysis data. Since our server's IP can only be accessed within the Unimelb network, you need to connect to the Unimelb VPN before accessing our server. And they would not be accessible after the end of the 2020 semester one because the cloud resources are only available when we were taking this subject. We only provide analysis for five cities in Australia, which is Melbourne, Sydney, Brisbane, Adelaide and Perth.

Replace the <city-name> below with any one of "melbourne", "sydney", "brisbane", "adelaide", "perth" to access our RESTFul API.

- To gain map info of a specific city:
  - http://172.26.132.125/api/v2.0/map/city/<city-name>
- To gain city-level analysis of a specific city:

  - http://172.26.132.125/api/v2.0/analysis/city/<city-name>
- To gain suburbs-level analysis of a specific city:
  - http://172.26.132.125/api/v2.0/analysis/suburbs-of-city/<city-name>
- To gain all analysis include both city-level and suburbs-level of a specific city:
  - http://172.26.132.125/api/v2.0/data/<city-name>
- To gain all city-level analysis:
  - http://172.26.132.125/api/v2.0/analysis/city-level/all
- To gain all suburbs-level analysis:
  - http://172.26.132.125/api/v2.0/analysis/suburb-level/all

7 Conclusion

In the project, an integrated cloud-based HPC system is developed on NeCTAR Research Cloud to gather and analysis tweets in Australia with six interesting scenarios. The system has scripted deployment capabilities using Ansible. Also, it is designed with high availability and fault tolerance capability. The analysis covered five major cities in Australia and can be drilled down to suburb levels. A web portal is provided to end users for accessing the data through ReSTful design.

8 Team Members and Roles

Many of the techs used in the project are quite new to our team, and thus we have been facing a lot of challenges and problems. But our teammates have done great work to study and utilize these cutting edge technologies into our system. We had good communication and proper project management to finish the development followed by a success deployment right on time. Each member has been taken a role as listed below:

Name	Contributions
Aoqi Zuo	Twitter harvester implementation, scenario designing
Rongbing Shan	Database administration, data processing, Docker SWARM.
Tingli Qiu	Analyzer module implementation, scenario designing, and analysis result presentation exemplar designing
Yawei Sun	Web Application Implementation
Zhaofeng Qiu	Ansible Deployment Implementation, System Architecture Design and Implementation, RESTFul API server Implementation, Docker images, Docker SWARM and CouchDB cluster setup.

Table 1: Team Members and Roles

References

[1] B. Fisher, "Pros and cons of all managers as workers in swarm," https://stackoverflow.com/questions/48853473/pros-and-cons-of-running-all-docker-swarm-nodes-as-managers, 2018.