Analysis of Lust and Greed in Australia

Name: Dawei Xu

Student ID: 870927

Contents

1. Introduction

1.1 Background

In Christian teachings, there are seven deadly sins which are classified from vices of people. It is also called the capital vices or cardinal sins. They are pride, greed, lust, envy, gluttony, wrath and sloth. People believe that they are behaviors or habits that may lead to some immoralities or crimes.

In this paper, we are interested in what may cause these kinds of deadly sins. The system is based on the NeCTAR Research Cloud. We will first harvest Tweets from Twitter, then identify some of the deadly sins through analyzing of these data and store the analyzed data in CouchDB. Finally, the analysis results are presented on a web page as well as official data from AURIN to find relationships between deadly sins and other factors.

1.2 Scenarios Considered

In this project, two kinds of deadly sins, lust and greed, are considered in different scenarios:

First, there is a saying that material comforts lead to sexual desire. So, we supposed that people live in more developed areas and have a larger amount of income may express more about lust on social media.

Another scenario we considered is about the consequence of greed. We suppose that greedy people who distinctly show their desires in their tweets may work harder than those who do not know what they want, and thus they may earn more than others.

2. UniMelb Research Cloud

2.1 Description

UniMelb Research Cloud, using the services of NeCTAR (National eResearch Collaboration Tools and Resources) which is a powerful OpenStack based cloud, provides researchers in UniMelb with data services. This platform is located in remote locations so it can help researchers to promote remote collaboration and reduce outcomes.

In this project, the Cloud provides four medium-sized virtual machines (VM) with 8 CPUs and 150Gb of volume storage for the team to develop the system.

2.2 Pros and Cons

As a cloud computing platform based on OpenStack, the advantages of UniMelb Research Cloud are listed as follows:

- It provides a platform for researchers across Australia, so developers can connect to the Cloud in remote places.

- Resources sharing among users significantly reduces the cost of the project.
- Users can use the Cloud's computation resources, such as access the same VM or data at the same time, which is beneficial for group cooperation.
- UniMelb Cloud has a large scale of storage and it gives users the freedom to allocate the storage on different instances. This makes it more flexible to develop our system.
- The research cloud allows users to use different security groups to access the instances, such as SSH for system management and HTTP for web services. Besides, it also provides authentication to improve security.
- A large range of APIs and associated services are available in the UniMelb Research Cloud. These APIs can be used to further develop applications easier and better.

However, some challenges may occur when using the UniMelb Research Cloud:

- Internet, as well as UniMelb VPN, is necessary to connect to NeCTAR Cloud. Errors may occur when the network is disconnected. Besides, the latency of the network is also a big problem when real-time interaction is required.
- Nodes can only be accessed using the terminal and command line. GUI of the node is not available, which makes it more troublesome to set up the node.
- Sometimes nodes of the Cloud may go down due to some system issues. It may cause errors to the Twitter harvesting process.

3. System Design

3.1 Description

In this project, we need to complete the whole system of sentiment analysis of seven deadly sins. Firstly, we need to deploy Docker on each node as the containerization tool for the whole project. Secondly, the CouchDB Cluster needs to be built among nodes to achieve data sharing. Thirdly we need to harvest the tweets of Australia from Twitter and determine whether each tweet conforms to relevant sins. Then the result of the analysis will be stored in the CouchDB Cluster. In addition, we also need to harvest the relevant statistics from AURIN. Next, the result of the analysis will be presented and compared with statistics in a front-end web application. Eventually, all these components will be deployed automatically by an Ansible script.

3.2 Resources and Tools

- Ansible - Automation tool
- Docker - The application container
- CouchDB Cluster - shared database between nodes
- MapReduce - For analysis
- UniMelb Research Cloud

- VMs - Four medium-sized VMs with 8 virtual CPUs (36Gb memory total) and 256Gb of volume storage
- Twitter API - Tweets harvester
- AURIN API - Statistic harvester
- Web application and visualization module

3.3 System Architecture

The basic flow graph of this project like this picture:

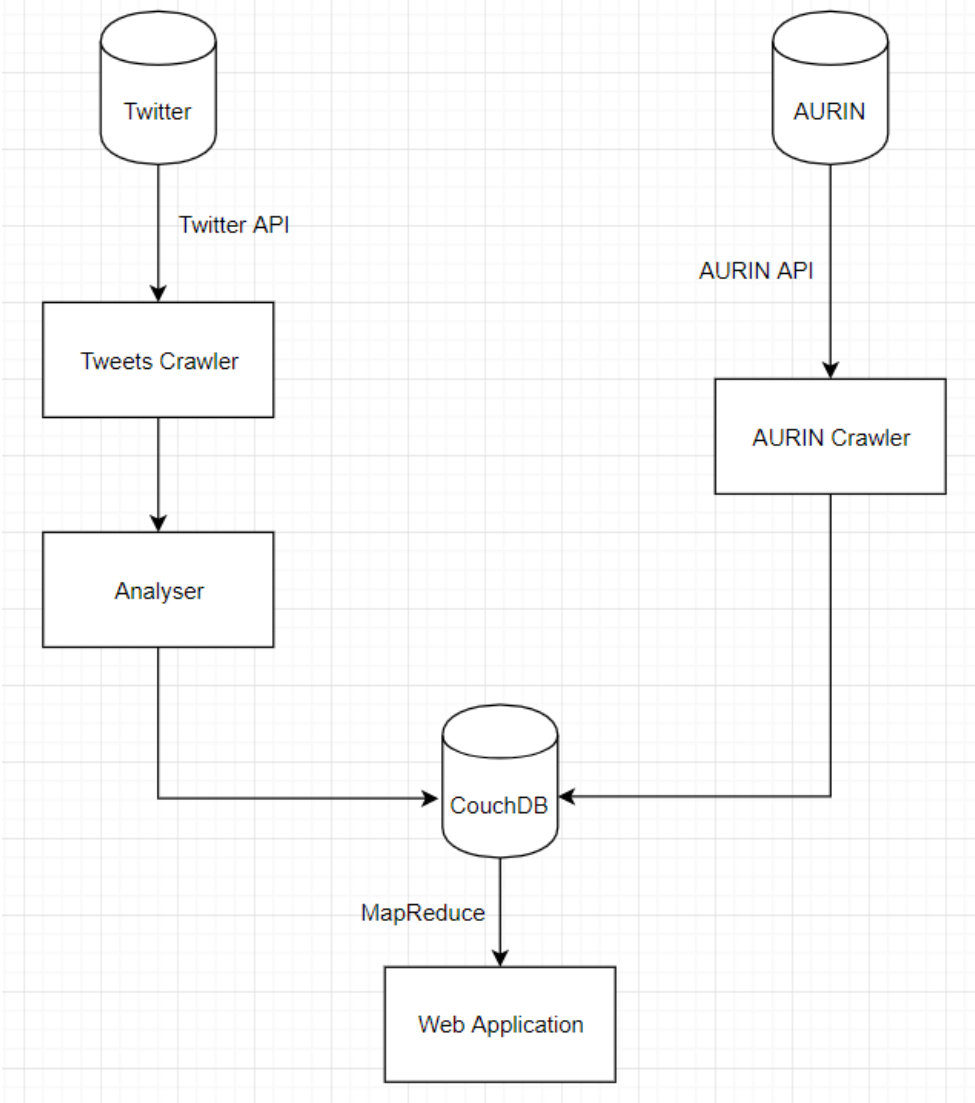

Fig.1 The flow graph of project

It needs to harvest tweets from Twitter and analyze the sin of tweet locally, then the result will be stored in CouchDB to store. In addition, the statistic harvested from AURIN is also uploaded to CouchDB. Finally, all the data in CouchDB will be presented by a Web Application in this project.

The structure of the system in this project and allocation of each node is presented in the picture:

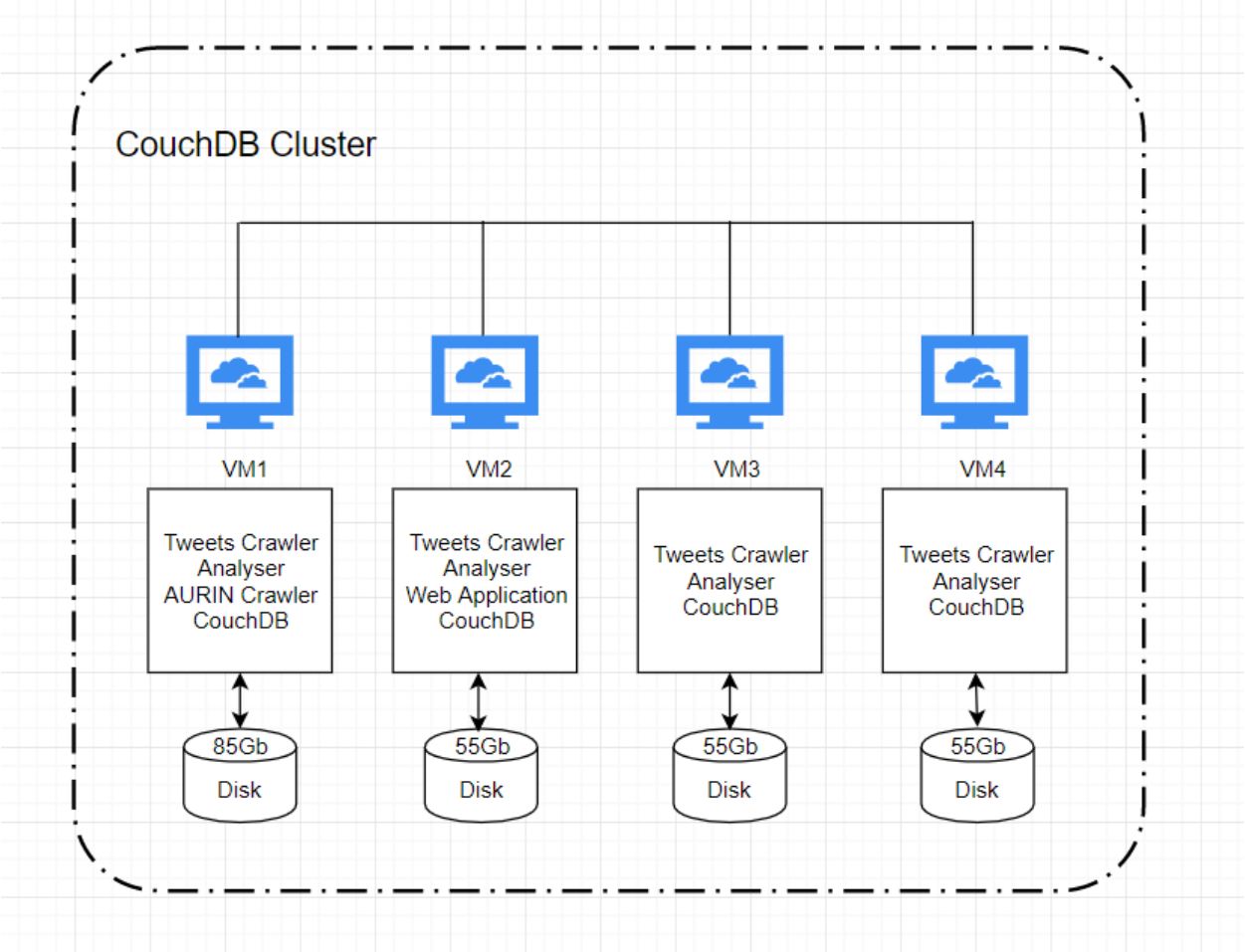

Fig.2 The architecture of system

Like the content of the picture, four crawlers deployed on four nodes will harvest tweets from Twitter concurrently. Then the system will filter irrelevant tweets and duplications and determine the possible sins from the rest of the tweets. The analytical results will be uploaded and aggregated in CouchDB Cluster. In addition, one node of four needs to get the statistic from AURIN and also uploads it to the CouchDB Cluster for future comparison and presentation. Finally, another node will read total analytical results and the statistics from the CouchDB Cluster, and add data to the Web Application to display the result of this project.

In this project, there are four medium-sized VMs with 8 virtual CPUs （36Gb memory total). We allocate 2 virtual CPUs (9Gb memory) to each node.

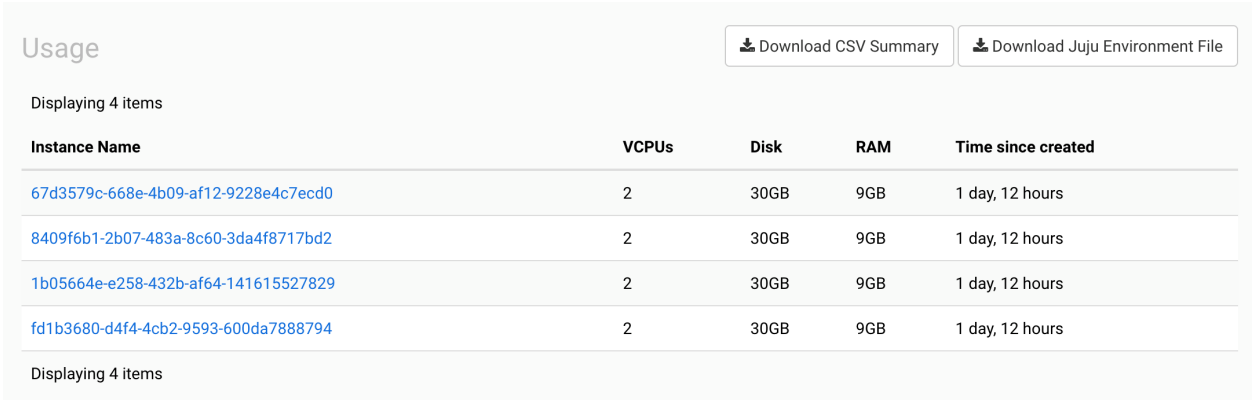

Fig.3 The allocation of CPUs and memory

For total 256Gb volume storage, we allocate 85Gb to one node and the rest of storage is partitioned to other three nodes averagely (each node possesses 55Gb).

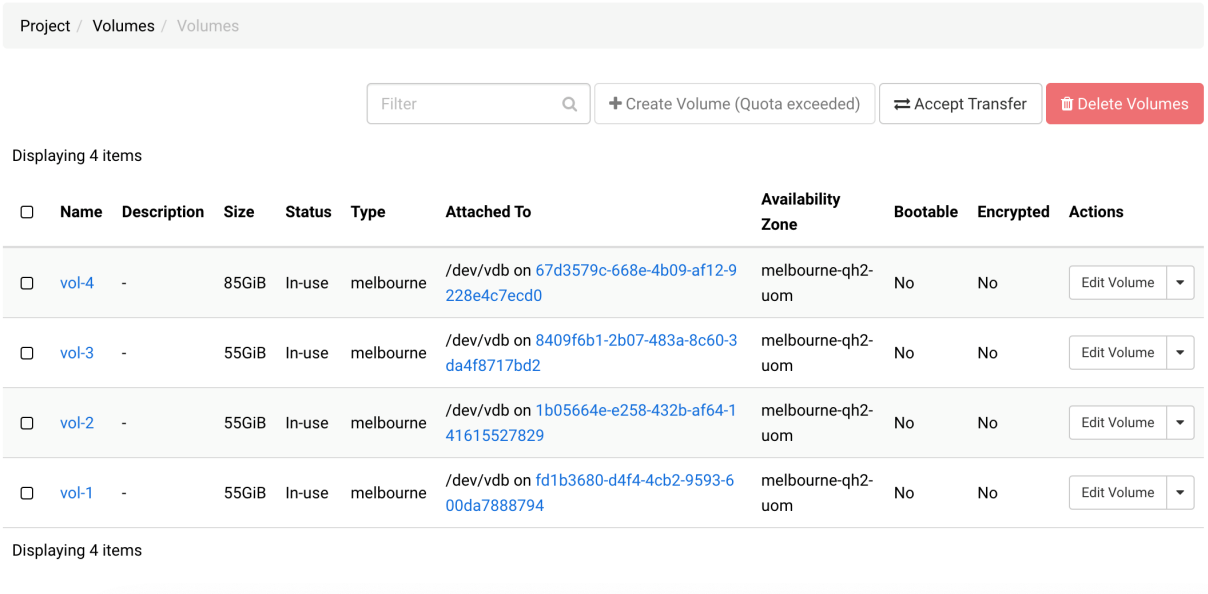

Fig.4 The allocation of storage

4. Data Crawlers

4.1 Overview

The crawlers part of this project consists of two crawlers which are about utilizing APIs and harvesting data from two data sources--Twitter and AURIN. In order to support the effectiveness of data usage and afterwards sins analyzer, the twitter crawler is designed to be able to harvest tweets generated in whole Australia; the AURIN crawler is required to be able to harvest data within a certain range of specified keywords and coordinates. In this section, we will introduce both crawlers and list out the challenges and solutions when implementation.

4.2 Twitter Crawler

Twitter is relatively researcher-friendly in comparison with many other social media apps. It exposed a large number of APIs for researchers to get access to tweets databases, such as Search APIs, Filter APIs, Account Activity APIs, Ads APIs and etc. Furthermore, there are a lot of Twitter APIs' wrappers applying to different programming languages which facilitates researchers in a large degree. In this project, we will take advantage of a wrapper named Tweepy for accessing the Twitter API. Then, we will retrieve tweets posted in Australia and historical tweets of a specific user.

4.2.1 Requirements

Tweets data are the main dataset of this project. The requirements of tweets and tweets crawler could be concluded as follows:

- Tweets must be posted in Australia;
- Tweets must have exact point coordinates for afterwards analysis;
- Tweets must include essential information such as text;
- The crawler is able to harvest tweets as many as possible.
- The crawler is expected to remove duplicates before saving to database.

4.2.2 Design

In order to satisfy these requirements, we designed a Twitter crawler system which has the following steps:

1) Build a coordinate polygon for the whole Australia territory;
2) Use Twitter Streaming API to put a listener on this coordinate polygon to harvest real-time tweets in Australia;
3) For each received tweet, extracting user's screen_name then retrieving this user's historical tweets using Twitter Search API and filtering out tweets that are not posted in Australia;
4) Put the rest tweets through sins analyzer, and get the results;
5) Remove duplicates and save to database.

With this design, we are able to harvest tweets that are definitely posted in Australia, and filtered unwanted tweets. The entire flow of this design is depicted below:

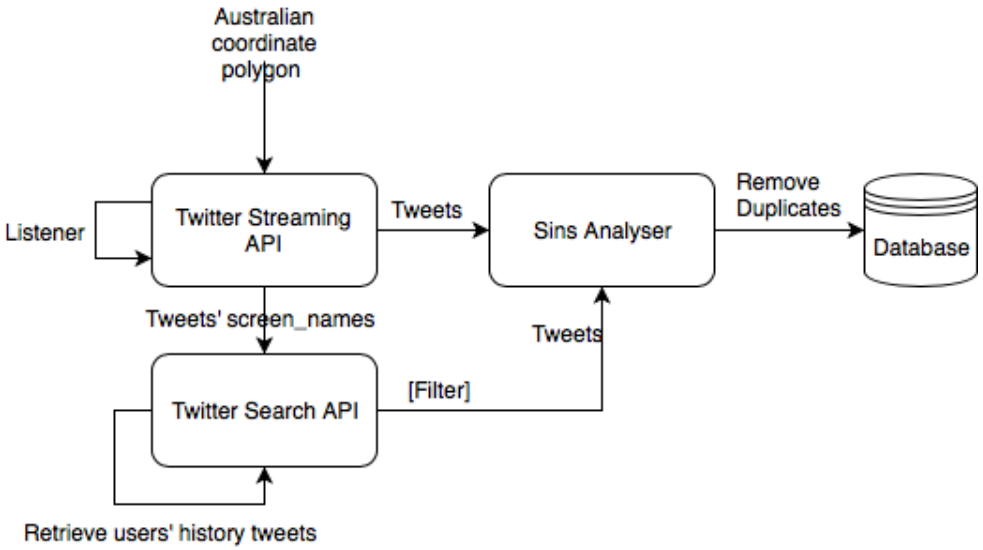

Fig.5 Design flow of twitter crawler

4.2.3 Implementation

We use http://geojson.io to obtain the coordinate polygon of Australian territory and saved it to *config.cfg* file which also contains other configuration information for the crawler such as consumer keys and tokens for accessing to twitter APIs. The value in the location box is latitudes and longitudes of down left and top right vertices.

```
25 [australia]
26 locations = [110.390625, -44.276671273775165,155.56640625, -11.005904459659451]
```

Fig.6 Australia territory coordinates polygon

We select Python as the programming language to realize the crawler and Tweepy as the Twitter APIs wrapper. There are a lot of Twitter APIs wrappers, we chose Tweepy because it is simple and easy to use, it provides complete access to the well documented Twitter API. With Tweepy, we are able to get any object and use any method that the official Twitter API offers.

In order to monitor tweets posted in Australia, we used the *StreamListener* Object. It has two methods *on_data()* and *on_status()* which allow us to do actions when data comes in, or print the status code when some error happened. Part of the code is shown below:

```
144     def on_data(self, raw_data):
145         coordinates = self.has_coordinates(raw_data)
146         if coordinates:
147             print ("This tweet has point coordinates!!!!")
148             db_data = self.generate_db_data(raw_data)
149             db_data["coordinates"] = coordinates
150             analysis_result = analyser.sin_analyse(db_data["text"])
151             if analysis_result:
152                 db_data["sin"] = analysis_result["sin"]
153                 db_data["sentiment"] = analysis_result["sentiment"]
154                 self.save2db(db_data)
155         user_id = json.loads(raw_data)["user"]["id_str"]
156         if self.is_user_tweet_already_returned(user_id):
157             print ("user's tweets have been returned.")
158         else:
159             print ("User history tweets harvesting has started...","="*20)
160             self.return_user_tweet(user_id)
161 
162     def on_error(self, status_code):
163         print(status_code)
```

Fig.7 Code snippet of usage of Twitter API

As we can see from the graph, after received a tweet, the first thing to do is checking whether this tweet contains exact coordinates attribute, if it does, then we will extract the necessary attributes of this tweet and generate a new data which will be in JSON format as well. Next, we will put this data through the analyzer and get the result. Only tweets with the valid result will be saved to the database, therefore tweets without exact coordinates and valid results will be discarded.

After processed each streaming tweet, we will extract out *user_id* of that tweet, then we use user_id to check if this user's history tweets have already been returned and then return the history tweets of this user. To return the history tweets of a user, we utilize *api.user_timeline* which is a method of Twitter Search API. This method is able to return a collection of tweets that up to 3200 of a specific user which will satisfy our data usage requirement. However, we need to deal with "page" manually in our pagination loop, if we choose the original *api.user_timeline* method. In order to simplify this operation, we use the *Cursor* object of tweepy, we pass the method and all the parameters into the *Cursor* constructor method, then *Cursor* will handle all the pagination work for us behind the scenes. Parts of the code is as follows:

```
106     def return_user_tweet(self,user_id):
107         isFailed = True
108         while isFailed:
109             try:
110                 tweets = tweepy.Cursor(StdOutStreamListener.api.user_timeline,user_id = user_id,tweet_mode="extended").items()
111             except Exception:
112                 print ("Cursor Issue.. Reconnecting")
113                 time.sleep(10)
114                 isFailed = True
115             else :
116                 isFailed = False
```

Fig.8 Code snippet of usage of Cursor

For each tweet returned by *Cursor*, we will check if this tweet has exact point coordinate attribute, put through the analyzer and save to database just like what we do to a Streaming tweet, however, we also need to check if this tweet is posted in Australia before put it through Analyzer. The control-flow graph is as follows:

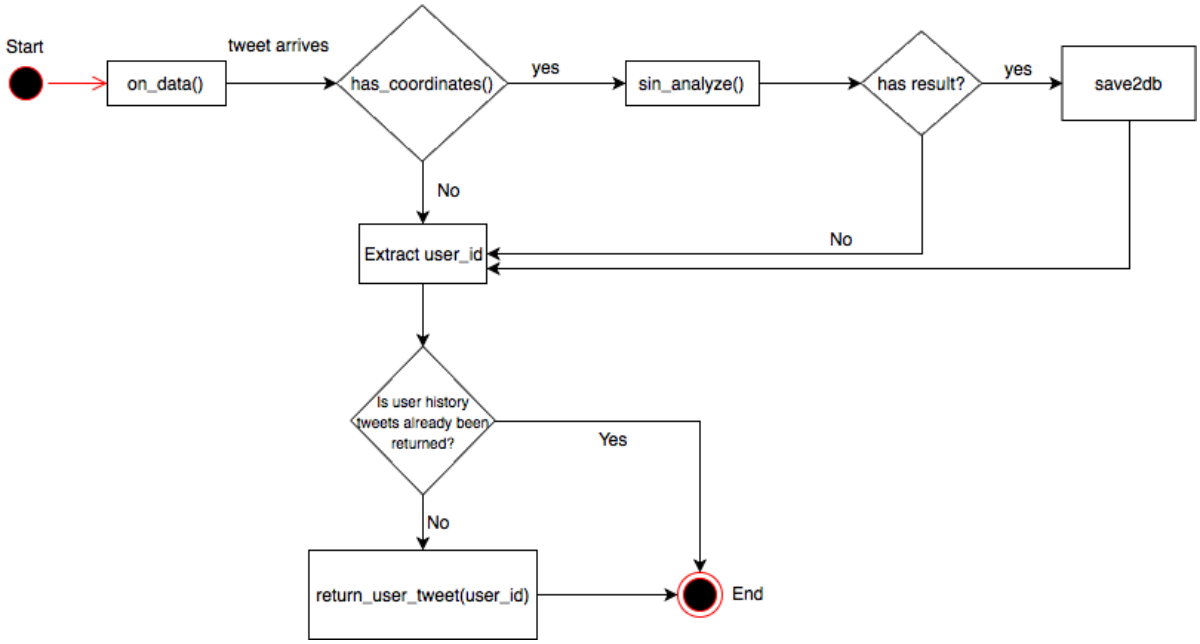

Fig.9 Control-flow of the program

4.2.4 Data Extraction

Raw tweets returned by Twitter APIs have multiple attributes, the attributes we need for this project are in the following table:

No.	Attribute	Type	Description
1	user_id	String	Used to tell if this user's history tweets have been returned.
2	screen_name	String	Used as identifier to return specific user's tweets and for afterwards analysis.
3	created_at	String	Creation time used for afterwards analysis
4	text	String	Used for afterwards analysis

Table 1: Attributes required from a raw tweet

4.2.5 Duplicates and Error Handling

Duplicates would do harm to sins analysis and be a waste of storage space, it is necessary to remove duplicated tweets.

In this project, we consider tweets with the same *id* as duplicates. Furthermore, in our design, each user's history tweets should only be returned once, so avoiding multiple traverses of a user's history tweets is also considered as a duplicate handling. We implement duplicates handling in the program through the interaction with CouchDB. Since documents in CouchDB have a unique identifier *"_id"*, therefore when we save tweets to the database, we set the tweet's id as the *_id* of the document,

next time when a duplicate is saved, there will be an exception given by the CouchDB. Similarly, we use the same strategy to treat the user's id duplicates. In this way, we do not need to set an identifier to remove duplicates locally and improve the utilization of memory. Moreover, crawlers deployed on multiple nodes could have a global database to remove duplicates instead of passing the message to each other for duplicates removal if we do it locally.

Parts of the code are as follows:

```
def save2db(self, db_data):
    try:
        database = self.server.database(self.database_name)
    except:
        print ("no such DB remotely..creating DB:" + self.database_name)
        database = self.server.create(self.database_name)
        database = self.server.database(self.database_name)
    try:
        database.save(db_data)
    except:
        print("Duplicate!!!!","="*50)
    else:
        print ("tweet has been saved to DB!!!","!"*50)
```

Fig.10 code snippet of using Database to remove duplicates

The most frequently triggered error in the system is network connection error, which are urllib2 error, TCP socket error or Twitter request error. These kinds of error happen on server side which is not feasible to be handled locally, so we just catch these errors and restart the streaming method. In this way, we can keep our crawler running forever.

4.2.6 Scalability

The duplicates handling is done through the database, and we use Docker to encapsulate our crawler so that the horizontal scalability is easy to realize. In this project, we deployed crawlers on all the four nodes, if there are new nodes joined in the future, we just need to deploy crawler on the new nodes and they would work independently without popping any collision issues. By the way, if we want to scale up vertically, we can use multiprocessing to run a number of crawlers on each node.

4.2.6 Challenges

4.2.6.1 Streaming API drawbacks

Streaming API provides only a sample of tweets that are occurring, the actual percentage of total tweets users received from Streaming API varies heavily based on the criteria users request and the current traffic. Studies have estimated that using Streaming API, users can expect to receive anywhere from 1% of the tweets to over 40% of tweets in near real-time. Moreover, it only returns tweets that are posted within one week.

To solve this problem, a combination of Streaming API and Search API can guarantee continuous tweets flow, and return tweets that are posted not only in this week.

4.2.6.2 Extended tweets

Since 2018, Twitter has raised the characters limit of each tweet from 140 to 280. However, some APIs still returned trimmed version of the text, and different APIs returns different tweet format. To solve this, we have to keep up with the documentation and have different process strategies for different APIs. In this project, we used two APIs--Streaming API and Search API. For Streaming API, the full text is located in “extended_tweet” attribute, we only need ['extended_tweet']['full_text'] to access it; for Search API, we need to request for “extended” tweet mode when making the request.

4.2.6.3 Limitation of Data Requests:

Although Twitter is quite friendly to developers, it sets a strict limitation for each API. For user_timeline api, it has a limit of 900 times request per 15 minutes; for streaming API, it has a limit of 400 keywords, 5,000 userids and 25 location boxes.

Solving this problem, we should work on not only breaking the limitation but also increasing the number of tweets replied in each request. Fortunately, wrapper Tweepy has a built-in function to wait for the limit automatically, and the combination of Streaming API and Search API could return a large number of tweets. We also deployed crawlers on four nodes to harvest tweets concurrently which improved the efficiency in a large degree. We roughly harvested 30w tweets with results in 48 hours.

4.3 AURIN Crawler

AURIN (Australian Urban Research Infrastructure Network) is a platform that provides various open, distributed datasets across Australia. It establishes an e-infrastructure with data interrogation services and online analysis tools for urban researchers.

AURIN allows data retrieving through its open API.

4.3.1 Requirements

In this project, data from AURIN play a role as proof to back up the conclusion derived from sins analysis, based on harvested tweets. Thus, the overall requirements for AURIN could be concluded as follows:

- Target Searching—searching for relevant databases based on keywords that appeared in the names of datasets or attribute lists.
- Filter the retrieved data that locates within the specified area.
- Pick out the data relevant to the topic of the project and discard the rest of the useless data
- Save data to CouchDB server for further use.

4.3.2 Challenges

When implementation AURIN crawler design, we can obtain only about a quarter of the whole datasets and can hardly find the datasets we need for our research objectives.

To meet our requirements, we have to manually retrieve in AURIN datasets and select a dataset that records Estimates of Personal Incomes which are aggregated in LGA (Local Government Areas) partition. The figure below shows the LGA partition. We can also find the longitude and latitude coordinates of dividing LGA which can be used to identify which area a tweet belongs to. In addition, the population of each area is also recorded in this project to calculate rates of people with some deadly sins.

Fig.11 LGA partition

5. Analyzer

5.1 Overview

The purpose of the analyzer in this project is to identify the tweets which are related to two kinds of daily sins: lust and greed. The analyzer will process the tweets returned by the crawler then filter the non-related data. The rest of the tweets will be labelled as lust or greed based on its analysis result. In order to achieve an accurate identification, Natural Language Processing is used to help us identify the related data.

5.2 Natural Language Processing

Natural Language Processing is a technology for the computer to understand the natural language. It will read, decipher and understand the human language. In this project, we use NLP as a tool to calculate the similarity between the tweets and our target daily sins to get the related data.

5.3 NLTK

In order to implement the NLP in this project, we choose to use a python open source library called NLTK. NLTK is a toolkit for building Python programs to work with

human languages data. It provides the service including classification, tokenization, stemming, tagging, parsing, and semantic reasoning. In this project, we focus on one of the packages of NLTK called wordnet to calculate the Wu-Palmer Similarity. The Wu-Palmer Similarity will return a score between 0 and 1 denoting how similar two-word senses are, based on the depth of the two senses in the taxonomy and that of their Least Common Subsumer.

First Word	Second Work	Wu-Palmer Similarity
Greed	Cupidity	0.94
Greed	Desire	0.58
Greed	Want	0.42
Lust	Carnality	0.83
Lust	Thirst	0.5
Lust	sensuality	0.875

Table 2: Examples of Wu-Palmer Similarity Score

From the table above we can see that the near synonym will always have a high score such as greed and cupidity. Other words that represent some of the emotion of the sin gets a reasonable score as well. Thus, for each tweet we got from the crawler, our analyzer will first tokenize the tweet. Then, we calculated the Wu-Palmer Similarity score between each token and two daily sins. Finally, we added up the score of all the tokens to get the total score of a tweet compared with both lust and greed. As a result, we can choose which daily sin is related to this tweet based on the total score.

5.4 Problems and Improvements

The analyzer above can indeed classify the tweets between lust and greed. However, there is an extreme problem that it cannot filter the non-related tweets. For example, if a tweet contains 5 words and each of them gets a 0.1 score for lust and 0.05 score for greed, it will be labelled as lust. Obviously, this tweet has no relationship with neither two daily sins, but our system will still store them into a database with a problematic label. In order to solve this problem, we set a critical value for the score. Based on many manual tests, we found that two words will have some kind of meaningful relationship (judged by human-beings) if the Wu-Palmer Similarity score is higher than 0.4. As a result, if a tweet cannot have at least one word reaching the critical value, it will be filtered by the analyzer.

6. CouchDB

6.1 Description

CouchDB is the open-source database management system. “Couch” is the abbreviation of “Cluster Of Unreliable Commodity Hardware”. So it emphasizes the usability, reliability and scalability of the database. CouchDB is for distributed environments. The storage system can be distributed in many physical nodes that it can keep the consistency of data when nodes read or write data. Thus, for the web-based applications which use large documents. CouchDB does not need to split documents and store in different places, which can reduce the modification of code in the application layer. In addition, CouchDB is a document-oriented database management system, which stores data as structured documents, usually expressed as XML or JSON. It is more convenient and has better performance than the traditional relational database. What’s more, CouchDB can use REST API that users can use JavaScript to operate the database.

These advantages discussed above just a part of CouchDB and it is very suitable to be used as the database system in this project, especially the CouchDB Cluster, which is designed for the distribution system.

6.2 CouchDB Cluster

Cluster is sets of connected computers and distributed databases run over clusters. Usually, the cluster can distribute the computing load over multiple computers which can improve the availability of the system. In addition, the cluster can achieve redundancy of database by storing multiple copies of data. These two main functions are important and necessary for this project.

6.2.1 Architecture

The architecture of the CouchDB Cluster is like the picture:

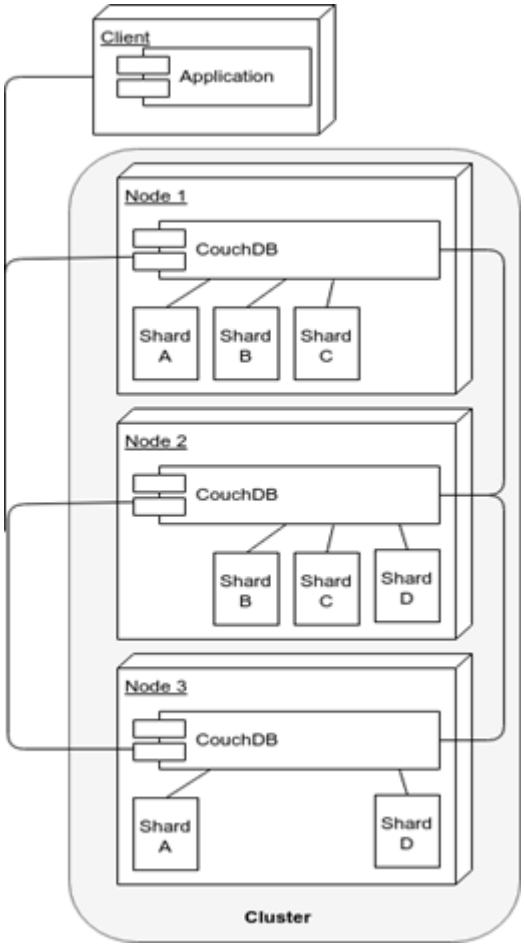

Fig.12 The architecture of CouchDB Cluster

Like the content presented in this picture, the cluster contains many nodes. And they seem to be one node for clients. Clients do not know the inner structure of the cluster.

When clients want to operate the database like read or write data, clients send requests to the cluster. All nodes answer requests at the same time. For the sharding, data will be split on every node. When a node does not have the document that clients request, it will ask this document from other node and return it to clients. In addition, nodes can be added or removed to the cluster easily depending on the requirement of clients. When the number of nodes changes, shards of nodes will be re-balanced automatically upon addition or deletion of nodes.

There are some advantages to the architecture. The first one is consistency, which means if there are many clients who connect with the cluster by different nodes and request the same data or operation, they will receive the same answer from different nodes. The second one is availability. Every node in the cluster sends answers to clients. The third one is partition-tolerance. Partition-tolerance is that the cluster can keep operating when one or more nodes cannot communicate with the cluster.

6.2.2 Pros & Cons of CouchDB

There are many popular document-oriented database management systems nowadays, CouchDB and MongoDB are the two most popular NoSQL databases, however we

choose CouchDB over MongoDB in this project, because CouchDB has following advantages:

- CouchDB Cluster is considerably easier than MongoDB Cluster.
- CouchDB Cluster is more available than MongoDB Cluster. In CouchDB Cluster, every node can send answers to clients, while only primary nodes can talk to clients for reading operations of write operations in MongoDB Cluster.
- CouchDB can connect with any HTTP clients, while MongoDB software routers (MongoS) must be embedded in application servers.
- If CouchDB Cluster loses two nodes out of three, it means losing access to one-quarter of data, while losing two needs in MongoDB Cluster, it implies losing access to half the data, even there are ten nodes in the cluster instead of three.

6.2.3 Build CouchDB Cluster

There are two optional ways to build CouchDB Cluster. One way is directly using Docker to build a Cluster. However, when using this way to deploy Cluster, it cannot be successful. The node cannot connect with other nodes just by using the IP of other nodes. When we want to use the master node to build the Docker containers of other nodes, it always failed that master node can only build the Docker containers of its own and it cannot access to other nodes. In addition, when we want to build the CouchDB Cluster, it only can build the CouchDB Cluster in the master node and other nodes cannot be added in this Cluster. The reason may nodes in this project are virtual machines, not physical nodes, and this way just can be used for physical computers to build Cluster. Another way to build CouchDB Cluster is by using Docker Swarm mode to create Cluster. The specific steps are:

- Create a four-node Docker Swarm cluster
- Deploy CouchDB service to Docker Swarm
- Scale and balance the cluster

Except these steps, it also needs to set up access of relevant ports in the security group of VMs, like 5984 and 2377, to ensure that nodes can connect with each other and anyone can access and manage the database by using valid username and password. After the steps above are completed, the CouchDB Cluster is built successfully.

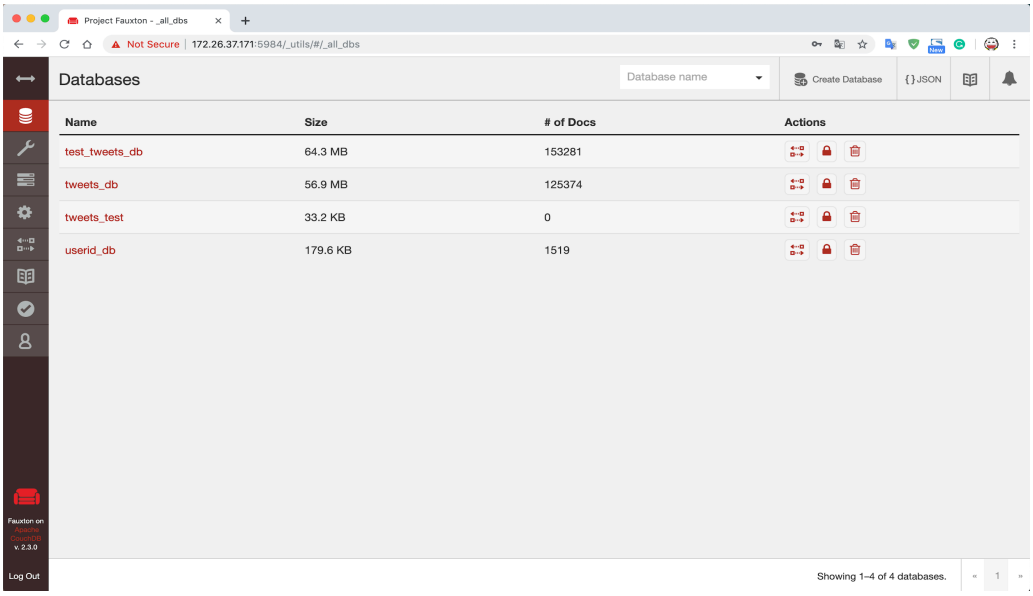

Fig.13 The web page of our CouchDB Cluster

6.3 pycouchdb

We use Python as our programming language to implement the crawlers, and we use *py-couchdb* as our CouchDB client to get access to CouchDB. *py-couchdb* provides faster requests than standard library and is compatible with both python2 and python3. The usage of pycouchdb is also very simple and easy, below is an example to connect to the server:

```
>>> server = pycouchdb.Server("http://username:password@localhost:5984/",
                              authmethod="basic")
```

Fig.14 Example code to connect to server

7. Automatic Deployment

7.1 Overview

It is tedious and time-consuming to create servers one by one and configure them in the same way. Automatic deployment allows applications to be deployed on different nodes without worrying about the environments. Using automatic deployment is more efficient, reliable and predictable.

7.2 Benefits of Automatic Deployment

There are several benefits of automatic deployment:

1. Deployments become less error-prone

   It always makes mistakes during the process of manual deployment. Sometimes, it is easy to forget what software you have installed, what steps you took to configure the systems. Moreover, import steps in a release can be accidentally

missed. Automatic deployment provides the record of what you did and is repeatable which dramatically reduces the burden of developers and testers.

2. Anyone can deploy software

   The system contains knowledge of how to deploy software. It can largely reduce the need for experts to do the work. Manual deployments are often the responsibility of a small group of people.

3. Developers can focus on the more important things.

   The engineers can spend their time on the development of software instead of wasting time on deploying the servers manually and fixing the errors of deployment.

4. Lower costs

   With the automatic deployment, it needs fewer errors and working hours than manual deployment, which means lower costs.

7.3 Introduction of Ansible and Docker

Ansible is a simple, powerful and open-sources IT automation engine. A great number of companies are using it to improve collaboration and increase productivity. Furthermore, its extensibility and modularity are allowed for a simple iterative deployment. With the ability to reproduce the environment from scratch, all environments are virtually identical in the development pipeline, including development, staging and production. A playbook is used by Ansible to define the steps that should be executed synchronously or asynchronously. It is well suited to configure a multi-machine system and deploy complex applications under simple configuration management. The main advantage of the playbook is that there is no need to install any software on the host in advance. It allows that the tasks or files are defined locally.

Docker is a tool designed for easier creating, deploying, and running applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. By doing so, the application will be able to run on any other Linux machine regardless of any customized settings that machine might have that could differ from the machine used for writing and testing the code. In a way, Docker is a bit like a virtual machine. But unlike a virtual machine, rather than creating a whole virtual operating system, Docker allows applications to use the same Linux kernel as the system that they're running on and only requires applications be shipped with things not already running on the host computer. This gives a significant performance boost and reduces the size of the application.

7.4 Design and implementation of deployment

One of the targets of the project is to automatically deploy the entire system, including creating the instances of UniMelb Research Cloud, configuring the environment of the instances, setting up CouchDB, deploying twitter harvesters and other related

applications. In our project, the Ansible playbook is applied to deploy the system for preserving the deployment scripts and reuse. The playbook is extensive that it not only creates and configures the instances in UniMelb Research Cloud but also installs the packages and deploys the application in each instance. The structure of our playbooks files is shown in the following figure.

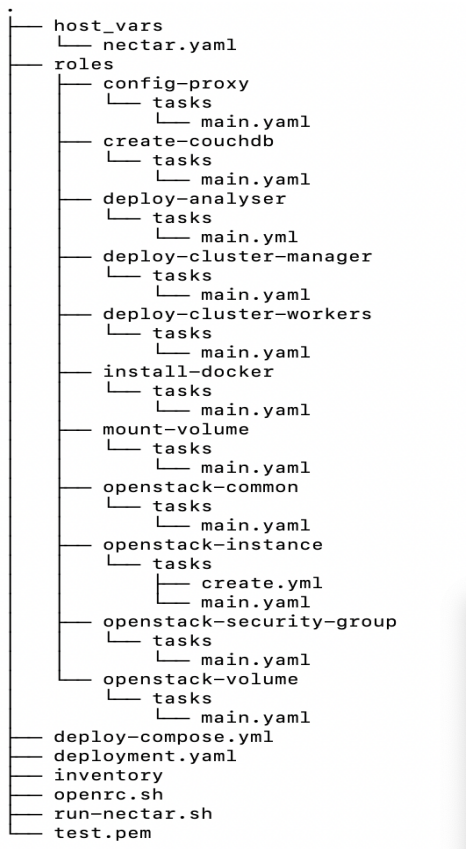

Fig.15 Structure of playbooks

There are several steps to deploy the system, which are discussed below.

1. Preparation of Deployment

   We must prepare prerequisites before connecting UniMelb Research Cloud and configuring instances.

   - The first step is to log into the cloud website, download the OpenStack RC file corresponding to the demanding project and get the API password. The RC file is named openrc.sh in the directory.
   - The instance image id, instance flavour, available zone and instance network are required to search and list out as parameters which are recorded in file nectar.yaml.
   - The allocation of resources plays an important role in making system running in a stable and balanced way. According to the system design mentioned above, set the volume and security group parameters in a file.

- An SSH key is needed to be prepared in advance, which is used to access the instances after creation. The private key we used in our project is test.pem in the main directory.

2. Create instances and relative configuration on UniMelb Research Cloud

   Before creating an instance, the security groups and volumes will be allocated. The playbook will undertake the creating instance part to generate four instances on Nectar based on predefined system parameters. The code of tasks is written in directories openstack-volume, openstack-security-group and openstack-instance respectively. After execution, it will appear four instances with different names on UniMelb Research Cloud website. A part of the code for creating an instance is shown in the figure. The contents in the brace are the variables for the parameters of the instance.

```
#create an instance on nectar
- name: "create an instance"
  os_server:
    name: '{{vol}}'
    image: '{{instance_image}}'
    key_name: '{{instance_key_name}}'
    availability_zone: '{{availability_zone}}'
    timeout: 600
    flavor: '{{instance_flavor}}'
    volumes: '{{vol}}'
    network: '{{instance_network}}'
    auto_floating_ip: yes
    security_groups: '{{sg_names}}'
  become_method: sudo
  register: os_instance
```

Fig.16 Code to create an instance to UniMelb Research Cloud

3. Configure the environment of each instance

   After getting four different virtual machines and IP addresses, mounting the corresponding volume of each VM becomes the first step. Because the VM is under the network of the University of Melbourne, adding HTTP proxy and FTP proxy are necessary for later steps. Directories config-proxy and mount-volume showed in the structure are used to modify the environment of each VM.

4. Install Docker and set up a network

   Docker is the most important platform in our project. The swarm provides great help in setting up a cluster. Installing the docker includes four steps. First of all, appending the GPG key for the official Docker repository in each system. Secondly, add the docker repository with a suitable version in APT sources. After that, install miscellaneous reliable packages including aptitude, apt-transport-https, ca-certificates, curl and software-properties-common after updating the APT package databases. At last, install docker-ce from the repository. The VMs are behind the HTTPS proxy server. In order to make docker connect to the network successfully, modifying the configuration in Docker system service file can figure out this situation. After finishing the tasks that recorded in install-

docker directory, the VMs are available to use Docker to download images and run containers.

5. Build a cluster via Docker Swarm

   This step mainly is divided into two parts containing create a cluster with four instances and deploying a stack to the swarm. As for the first part, the predetermined instance initializes a swarm and acts as a manager of the cluster. The other instances join to the manager's swarm with a token and IP address.

6. Build a load balancer and create CouchDB service on Docker Swarm

   An external load balancer HAproxy is run on the manager node. As for creating CouchDB service, it plans to transfer the compose file named deploy-compose.yml to the manager instance. Before deploying the stack with the compose file to the swarm in manager node, downloading the image of CouchDB in all nodes is a fundamental stage. In this step, the port 5000 will be exposed to visit CouchDB database.

7. Deploy twitter harvester and analyzer

   The twitter harvester and analyzer are packaged together and pushed to Docker hub as an image. Each node just needs to pull the image down from Docker hub and run a container on top of the image.

8. Run the website

   After installing Apache2 through APT, the website files should be sent to the default directory /var/www/html. The web page will be accessible with the IP address and port 80.

7.5 Challenges and solutions

We encountered a number of challenges during the process of deployment. The first issue was recording the IP addresses of virtual machines. Because of the dynamic generation of IP address in UniMelb Search Could, the IP addresses are unknown beforehead. In order to establish 'one command deployment' without checking on website, it is necessary to analyze and record the result from creating an instance. However, the result cannot be simply added to a variable before analyzing when using a loop to generate multiple machines. Otherwise, the variable cannot be parsed as a dictionary. The solution to this issue is that the included module is applied in a loop. The task listed in the include module contains setting up a machine and putting the IP address from the result after analyzing into a list. A further step is needed that the list must be set as a fact of localhost for making it accessible in other playbooks.

Another problem is automatically setting up a CouchDB cluster. We tried several ways to figure out the problem, but it came up with various bad situations. First of all, let each docker of virtual machines run a container of CouchDB and plan to configure a cluster in a manual way. Based on the guide of the official document of CouchDB, it is required to modify configuration files and use Erlang. However, the container does not have an editor to modify the files and can not connect to the network. Using the commands to configure outside the container, it did not seem feasible after trying many

times. The master cannot add other nodes successfully even though the result shows it works. Another way is to adopt docker swarm to set a cluster without using the official provided configuration methodology of CouchDB. Deploying a service to docker swarm before scaling to all nodes can be one of the keys to the headache question. But it is not the final answer to this issue. The database system is not stable with some unknown reason. The databases in the system are 'flashing' that it disappears after refreshing the CouchDB website or becomes inaccessible. Our final approach is to deploy a stack to swarm with a compose file instead of a service. This way is more appropriate and reusable than using extra input in command because all configuration and parameters are written in the compose file.

Besides these two issues, the docker behind HTTP proxy and set up ports for exposing are hidden problems which are demanded to spend time to do research and understand the execution process of the system.

8. Result Analysis

8.1 Statistic

When we started to write this report, there are about 176 thousand tweets from Australia in CouchDB Cluster, which are harvested by Twitter crawler and classified to greed or lust by the analyzer.

	Greed	**Lust**	**Total**
number	146,065	29,862	175,927

Table 3: The number of different sins

In the detailed data, 146,065 tweets are judged to greed, 29,862 tweets are lust. The total number is 175,927.

Then we calculate the number of greed or lust in each state of Australia according to the longitude and latitude tweets and filter some tweets whose locations are not in Australia.

The statistical result is presented in the chart:

State	**Greed**	**Lust**
ACT	4,707	1,680
NSW	55,713	12,035
NT	796	116
QLD	28,040	6,357
SA	2,501	558

TAS	6,333	427
VIC	41,369	7,145
WA	5,155	1,171
total	144,614	29,489

Table 4: The number of different sins in each state

From the AURIN, we harvest the population and mean person income of each state in the below chart:

State	Population	Mean Person Income
ACT	397,397	70,804
NSW	7,955,900	67,806
NT	246,700	59,232
QLD	4,827,000	61,578
SA	1,706,500	57,028
TAS	518,500	54,569
VIC	6,430,000	61,785
WA	2,613,700	71,472

Table 5: The population and mean person income in each state

According to the population of each state, we divide the number of greed or lust by the population to calculate the proportion:

State	Proportion of greed (%)	Proportion of lust (%)
ACT	1.184	0.423
NSW	0.700	0.151
NT	0.323	0.047
QLD	0.581	0.132
SA	0.147	0.033
TAS	1.221	0.082
VIC	0.644	0.111

WA	0.197	0.045

Table 6: The proportion of greed and lust

8.2 Analysis

There are abundant tweets can be regarded as the sin of greet, but the number of lust is small. Because the tendency of greed is easily judged by the content of text, while relevant words about lust are subtle and pictures should be another main basis to determine.

For the relational between greed and person income, our assumption is basically correct. According to the data in ACT, NSW, VIC, QLD, NT and SA, there is a positively related correlation between the proportion of greed and person income, which supports the view that greedy people usually work hard to earn more money. Other two instances are also explainable. In South Australia, it possesses the richest natural resources, especially mineral resources. And miner is generally regarded as a highly-paid job. Life is more comfortable and people are more satisfied with SA. In contrast, Tasmania is remoter and resources and job opportunities are less than other states. Although people live in this place want to earn more money, they do not have enough changes to achieve.

For the scenario between lust and person income, we suppose that people who live in an advanced place and have more income have more tendency of lust. ACT and NSW are the second and third of person income respectively, the proportion of lust in these places also two highest. The economy and person income of Victoria and Queensland are similar. So the percentages of lust in these two states are are equal, which both less than NSW. There is the same relation between South Australia and North Territory. The relevant data is inferior to VIC and QLD. For the most special state, West Australia, the mean person income is highest in Australia, but the proportion of lust is the lowest on the contrary. This situation may be explained by the same reason of greed. So, the assumption about the lust and person income is also true to some extent.

9. User Guide

There are two main steps to test the system including preparation of deployment and executing bash file. GitLab Repository contains all codes: https://gitlab.unimelb.edu.au/zizhen/cccproj2. Before following below steps, downloading the code in nectar from GitLab is the foundation of the entire process.

9.1 Preparation of deployment

As for the deployment of the necessary development environment, the end user has to install pip and ansible before execution. Downloading OpenStack RC file and setting the password of access is important in creating the instances on UniMelb Research Cloud. Moreover, generating the SSH key in advance is to ensure that the end user can

access virtual machines. The RC file openrc.sh and private key test.pem are already prepared for our project which is shown in the nectar directory.

If the end user is using MacOS, the installation command is “sudo easy_install pip”. In an Ubuntu system, you can execute the command “apt-get install pip” after updating the apt repository. After successfully installing pip, it is easy to get ansible with the command “sudo pip install ansible”.

9.2 Executing bash file

Navigating into the location of nectar directory where you download the code of the project. Run the command “sudo bash run-nectar.sh” to deploy the entire system including creating and configuring the virtual machines, deploying database and other applications. After that, you can access the CouchDB database through the IP address and port 5000 of manager and the result of analysis on website e.g. http://172.26.38.66:80/web/index.html.

10. Web and AURIN Data Demonstrations

10.1 Data collection

The main purpose of the website in this project is to create a user interface which can show the summary of the data we collect. In order to show the result to the user more directly, we decided to generate a map which contains all the information. To achieve this, we first build a MapReduce function in CouchDB to extract the coordinates field. Thus, we get a dataset which contains all the coordinates of lust and another dataset that contains all the coordinates of greed. We imported those datasets into the AURIN portal and used “Tools::Spatial Data Manipulation:: Count Points in Polygons” to get the number of such tweets in each state area. After that, we used “Tools::Spatial Data Manipulation::Spatialise Aggregated Dataset” to generate the shapefile we can use to show the result on a map. For the AURIN data, we imported them into AURIN portal from AURIN dataset directly, them run the same tools on them to generate the shapefile.

10.2 Google Map API

The Google Maps JavaScript API can help us customize maps with our own content and imagery for display on web pages. Unfortunately, google maps api doesn’t support shapefile as resource file. In order to post our map on the website, we use a geometric toolkit called QGIS to transfer the shapefile into kml which google maps api supports. As a result, we got an embedded map inside our website like this:

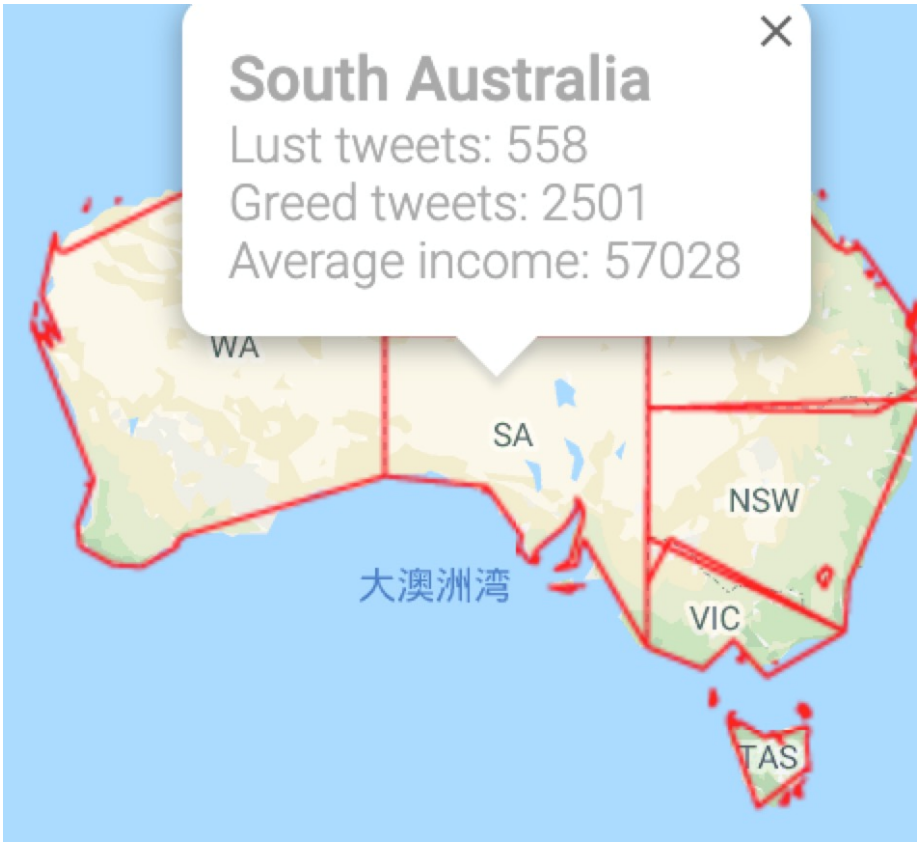

Fig.17 Embedded map in the website

11. Conclusions

In this paper, a system based on UniMelb Research Cloud has been designed to harvest data from Twitter, store and analyze them. First, we used the Twitter API to crawl data and remove duplicate data. Then, using NLP methods, we classified crawled tweets as two deadly sins, lust and greed, and stored them in a CouchDB cluster. Millions of tweets across Australia were processed and about 176 thousands tweets were collected to compare with statistics of income provided by AURIN to prove our supposed scenarios that lust and greed are both positively correlated with income. Finally, we demonstrated our results on a web page.

References

[1] Apache Software Foundation. "*Apache CouchDB*". Retrieved 15 April 2012 from http://couchdb.apache.org/

[2] Anderson, J. C., Lehnardt, J., & Slater, N. (2010). *CouchDB: The Definitive Guide: Time to Relax.* " O'Reilly Media, Inc.".

[3] AURIN.org.au, (2018). *GCCSA Estimates of Personal Income - Total Income 2010-2015*. [Online] Available at: https://data.aurin.org.au/dataset/au-govt-abs-abs-epi-total-income-gccsa-2010-15-gccsa-2016

[4] Hochstein, L. (2015). *Ansible: Up and Running*. Sebastopol: O'Reilly.

[5] Google Developers. (2019). *Overview | Maps JavaScript API | Google Developers.* [online] Available at: https://developers.google.com/maps/documentation/javascript/tutorial [Accessed 14 May 2019].

[6] Go, Alec, Richa Bhayani, and Lei Huang. "*Twitter sentiment classification using distant supervision.*" CS224N Project Report, Stanford 1.12 (2009).

[7] py-couchdb 1.14. Last released:6 Nov 2015. https://pypi.org/project/pycouchdb/

[8] Nltk.org. (2019). *Natural Language Toolkit — NLTK 3.4.1 documentation*. [online] Available at: https://www.nltk.org/ [Accessed 14 May 2019].

[9] W. Li, J. Liang, X. Ma, B. Qin, and B. Liu, "*A Dynamic Load Balancing Strategy Based on HAProxy and TCP Long Connection Multiplexing Technology*," Heart Failure Cardiovascular Medicine, pp. 36–43, 2018.