COMP90042 Project 2 – Sins on Twitter

Team: Group 68

City: Melbourne, Australia

Members (name, id):

Darren Pinto, 1033936

Matthew Yong, 765353

Mengzhu Long, 943089

Wenhao Zhang, 970012

Yang Liu, 837689

Table of Contents

Introduction

The seven deadly sins, pride, greed, lust, envy, gluttony, wrath and sloth are deemed as the bad behaviors or thoughts of human. This project is to explore the seven sins based on a huge number of tweets collected. Through twitter's streaming API, we collected around 60,000 pieces of tweets. Each twitter is tokenized, stored in CouchDB, and can be indexed by searching a keyword fast. A web application that is implemented using Flask, which has functions of visualizing the result of tweet's data analysis and comparing it with official AURIN data in different ways by interacting with the user. The whole system is implemented and automatically deployed on the Nectar cloud system based on the computing resources allocated.

Scenarios and Application Functions

The sins we choose to explore in this project are wrath, sloth, greed and lust. We define four scenarios under which should a tweet commit one of the sin.

- Wrath: Tweets indicating anger and rage towards someone or some incident in positive emotion. Cursing and swearing also fall under the sin of wrath. A list of swearing words are used to help identify a sin tweet of wrath.
- Sloth: Tweets includes words related to exercise, study or work negatively are identified as sin tweets of Sloth.
- Greed: Tweets includes words related to wealth and money in positive emotion are identified as sin tweet of greed.
- Lust: Tweets includes obscene words which are banned by Google are identified as sin tweets of lust.

Hypotheses

In this project which is based on tweets analysis, we define a sin degree in a state of Australia as the percentage of sin tweets within that state. A tweet can be classified as either non-sin tweet or sin tweet based on the wordlist which are highly related to a specific sin, i.e., wrath, sloth, greed, or lust. Based on the AURIN data available and relatively with the sins, we make hypotheses below.

- The percentage of sin tweets of sloth as defined above might be higher in a state which has higher percentage of population who did low or no exercise.
- Within a state, the rate of types of criminal offences should have the similar spread shape with that of the percentage of corresponding type of sin tweets. For example, robbery and extortion is related to the sin of greed because greed will lead people to commit crime to extort money from others. Sexual Offences are related to sin of lust. Offences against person are the sin of wrath.

Application Functions

The presentation layer of our web application provides the user with three functions.
To explore and classify if a tweet has a given sin, a wordlist will be needed, either pre-defined one or a user-defined one to be utilised. As shown in Figure 1, when the user clicks a search button with a blank Keyword Search Window, a list of sin-related words will be utilised as the key wordlist to help classify a sin tweet. In another situation, if the user input a list words into

the search window, then these words will be utilised as a user-defined wordlist to help classify a sin tweet.

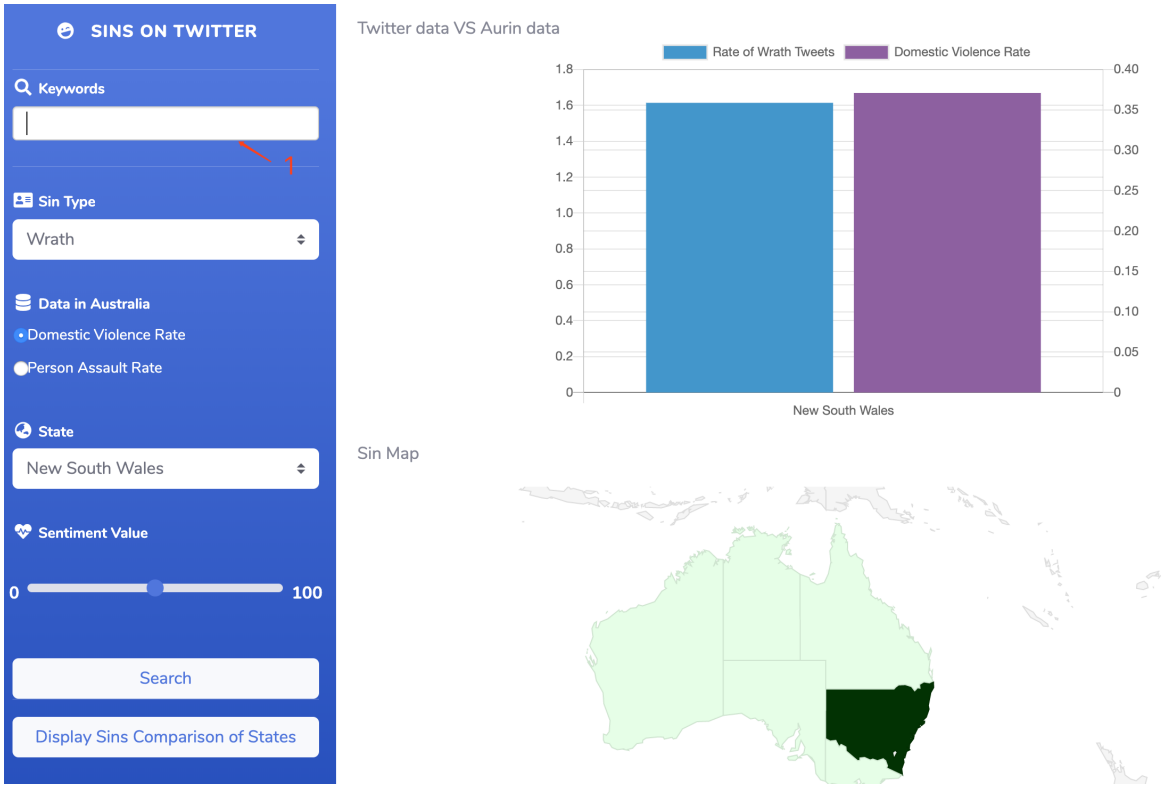

Figure 1. Search Function of the Web Application

As shown in Figure 2, the user is allowed to look at the result and compare it with AURIN data by selecting which sin, which state as well as which AURIN dataset. The bar chart and radio chart are two types of charts for different objectives. The bar chart is to compare result of tweets data analysis and AURIN data. The radar chart is to show the spread of sins in terms of each state in Australia. The map is to show the result of tweets analysis in each state.

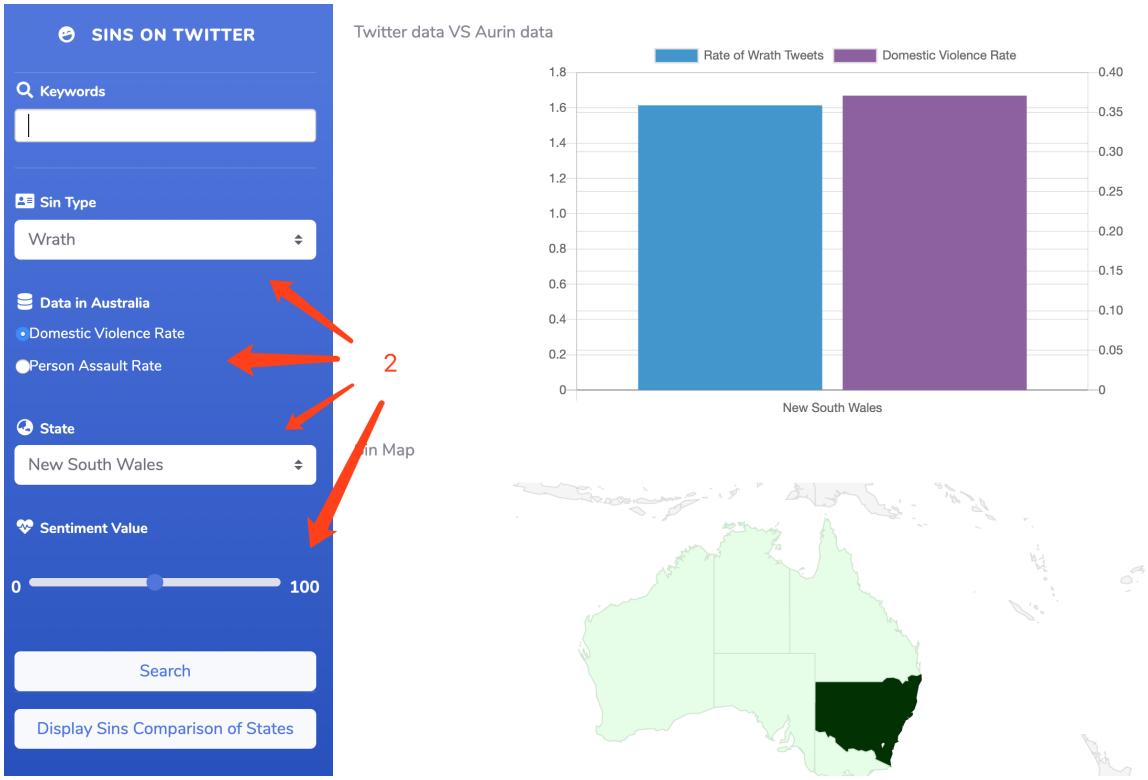

Figure 2. Selector Function of the Web Application

Visualizing

There are three visualization components in front end. Based on the state location of the tweet, we calculate the percentage of sin tweet which shows a degree of sin in that state.

Bar Chart

The bar chart provides comparison between twitter data and AURIN data. For example, in the Figure 3, the bar in purple means the AURIN data of the criminal rate of offences against person in NSW, which is shown in ascending.

The bar chart in blue shows the percentage of Tweets that “commit” the sin of wrath. As mentioned above. Tweets of a sin are classified out of the tweets collected. In this way, we can easily tell if the trend of sin degree of states aligns with that of the AURIN data.

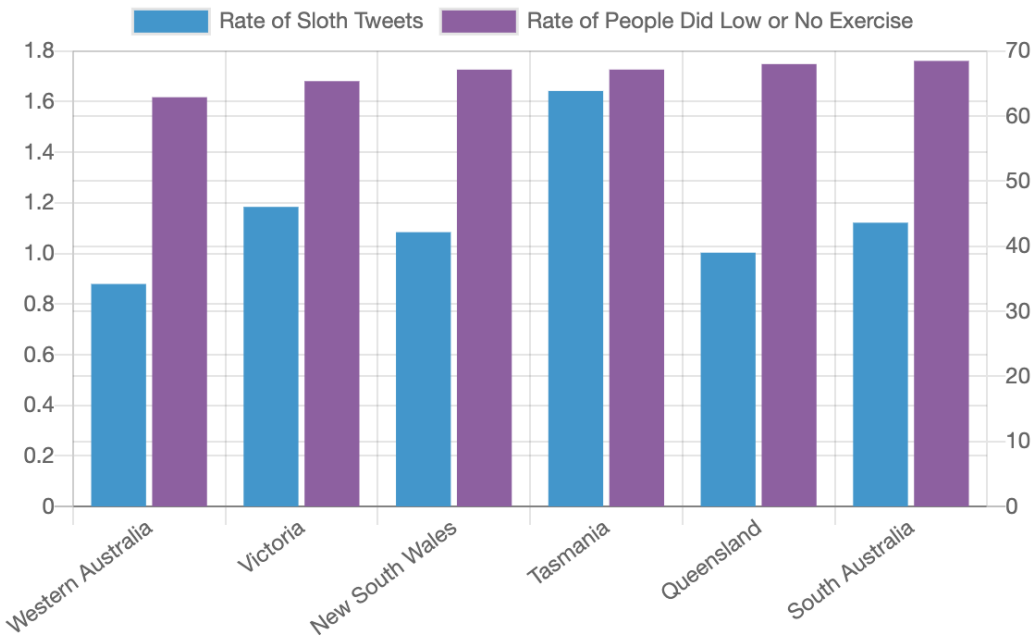

Figure 3. Bar Chart of the Web Application

Map

Based on the sin selected by the user, the map shows degree of the sin in each state. The degree of a sin in a state is defined as the percentage of sin tweets over the number of total tweets in the state. The user can find out easily which state has more degree of sin by checking which state is in darker color than others.

Figure 4. Map of the Web Application

Radar Chart

The radar chart enables the user to explore the degrees of sins and compare them between states. In this way, the user can easily find out the spread of sins in a state. The user can make a state invisible on the chart by clicking the corresponding label, as shown in figure 5.

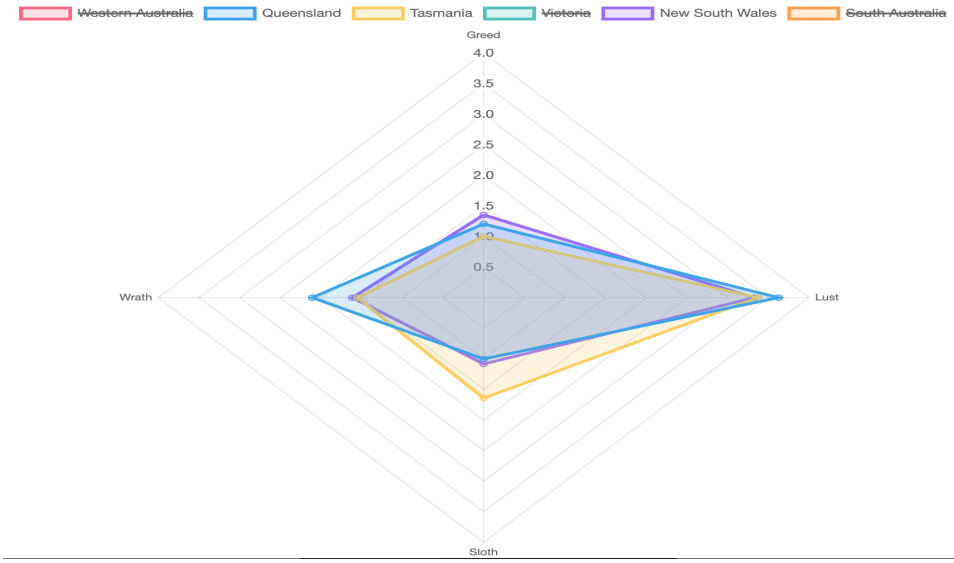

Figure 5. Radar Chart of the Web Application

System Architecture

Overview

To implement our cloud-based solution, we would first require major computing resources especially in computing power and data storage capacity. The teaching team has allocated us four medium-sized virtual machines, eight virtual CPUs, 36 gigabytes worth of memory and a total of 250 gigabytes of volume storage to be used in our project on the Nectar Research Cloud.

System Design

The general top-level diagram of our system design can be seen in Figure 6. The components of our system can be categorised into different groups which are:

- The Virtual Machines (also known as nodes or instances)
- The Database System
- The Twitter Harvester
- The Web Application

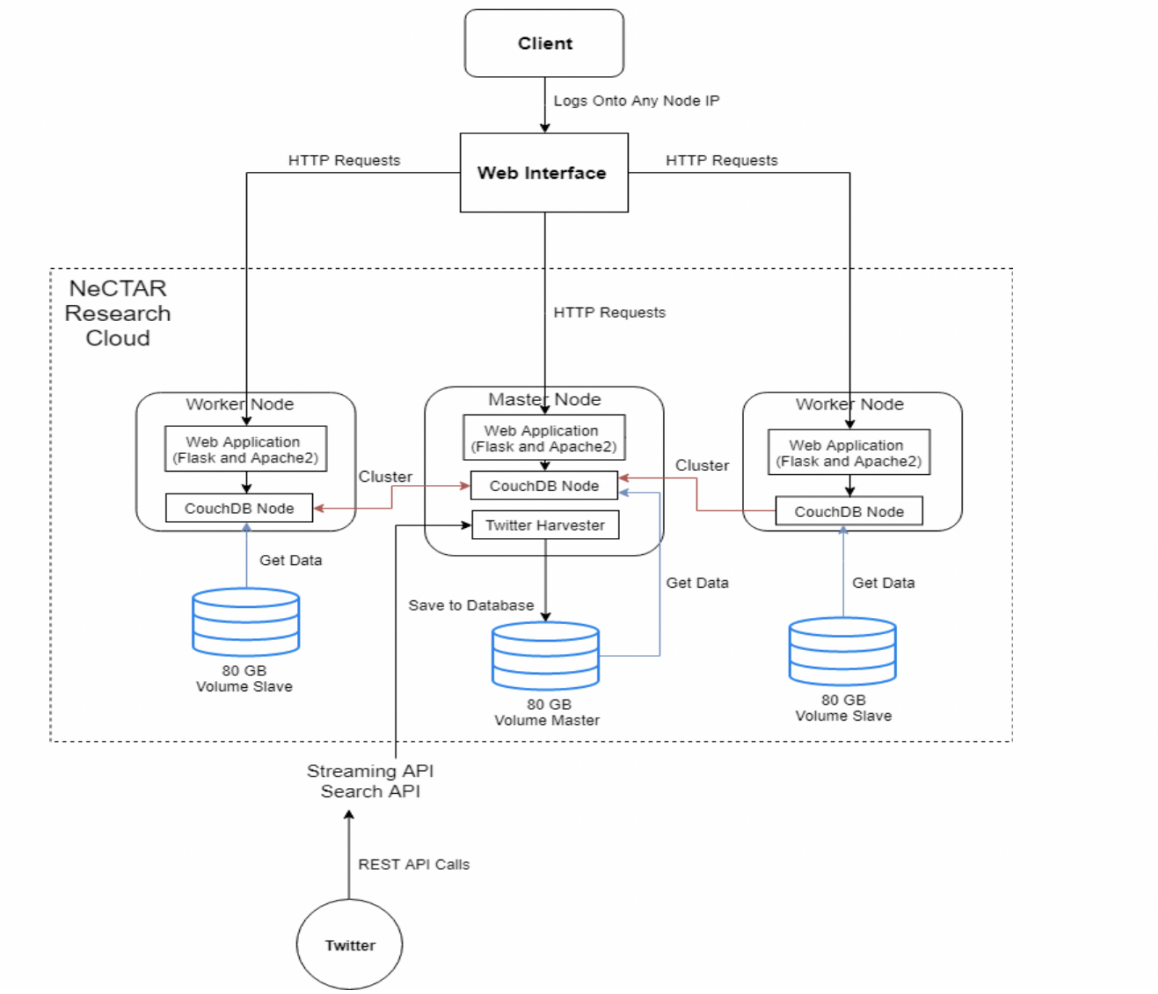

Figure 6. System Architecture of the Web Application

Virtual Machines

Using the available resources given to us, we decided on having a system setup comprising of three virtual machines. The virtual machines are individually installed with an Ubuntu 18.04 LTS (Bionic) amd64 Linux operating system which was obtained through Nectar. Each of those machines run on 2 virtual CPU cores and 9 gigabytes worth of memory as we wanted to have an even distribution in computing power across each node. Machines were also attached a volume storage worth 80 gigabytes to ensure that we have sufficient buffer space for any sized data that we may use in our project. With the remaining resources, we decided to create a test instance to perform any software or scripting tests before migrating or applying the new changes to our actual system.

All instances host two common components, a single instance of our web application and a single CouchDB node instance. All single CouchDB node instances are then joined together to form a collaborative three-node cluster which can be seen as a “single” database for our system. In having multiple nodes serving the web application and hosting a database instance, our system would be fault-tolerant to any unexpected outages that may occur on any node. With this, we were also able to balance the web serving load across each node by equally

distributing our IP addresses to each client in order to access our web application thus avoiding overloading on a single machine.

Slave Node

In our three-node cluster setup, we first have our two slave nodes. These nodes are configured identically and are replicas of each other. The reason for this was because we wanted our system to have an equal amount of work distributed among slave nodes and thus configuring them similarly made it easy for us to distribute the work load. From Figure 7, we can see the general arrangement of a single slave node in our system. As mentioned previously, a single slave node hosts a single instance of our web application and has a single CouchDB node connected in a cluster between the other slave nodes and master node CouchDB instances.

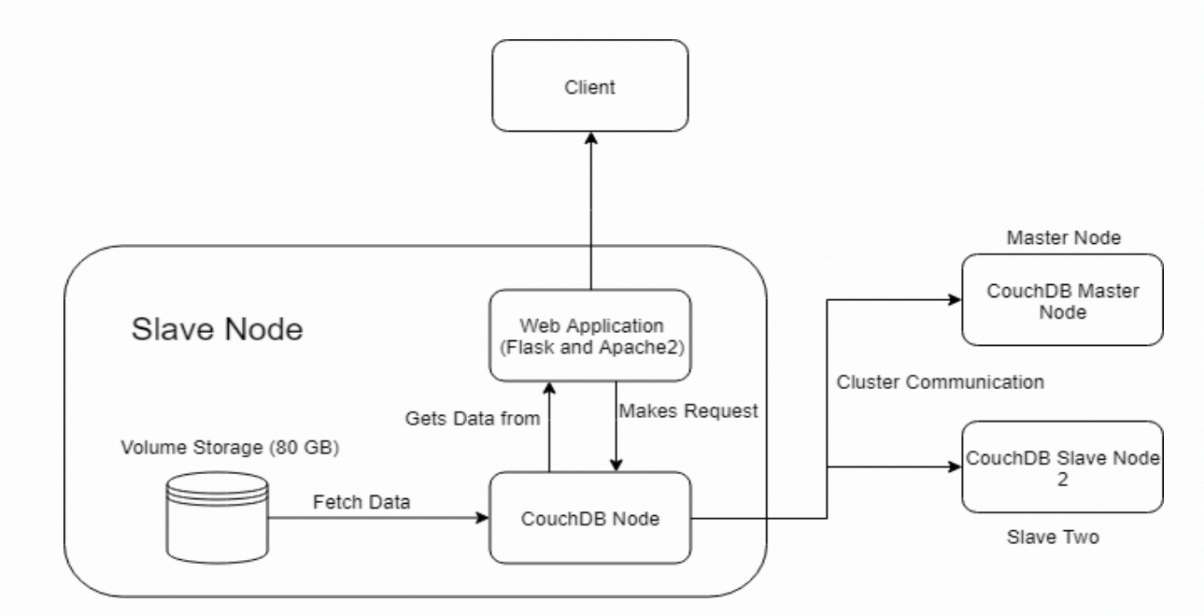

Figure 7. Overview of the configuration of a single slave node

Any client can interact with a slave node's web application by accessing its IP address on a browser. Once the client makes a query on the web interface, the web application then sends a HTTP request to the local CouchDB node hosted on this slave. As CouchDB nodes are configured in a cluster, the data needed by the web application may either be fetched from the slave's local volume storage or in any other CouchDB instance's volume storage living on other master or slave nodes (depending on where the specified data resides in the cluster). The data requested by the client is then sent to the local CouchDB instance in which is finally passed on to the web interface to display the results. Theoretically, our system can be scaled to any size we want by having as many slave nodes hosting our web application and CouchDB instance as they do not have distinguished roles that require them to be uniquely present in our system.

Master Node

The master node in our system encompasses all of the functions provided by a single slave node but with an exception of having the role of being the node that configures other slave nodes. As our database system revolved around a cluster setup, the master node was allocated to configure other slave nodes to be gathered together in the cluster. This meant that our system cluster was set up by running CouchDB cluster commands on the master in order to attach all remote slaves to be in the cluster.

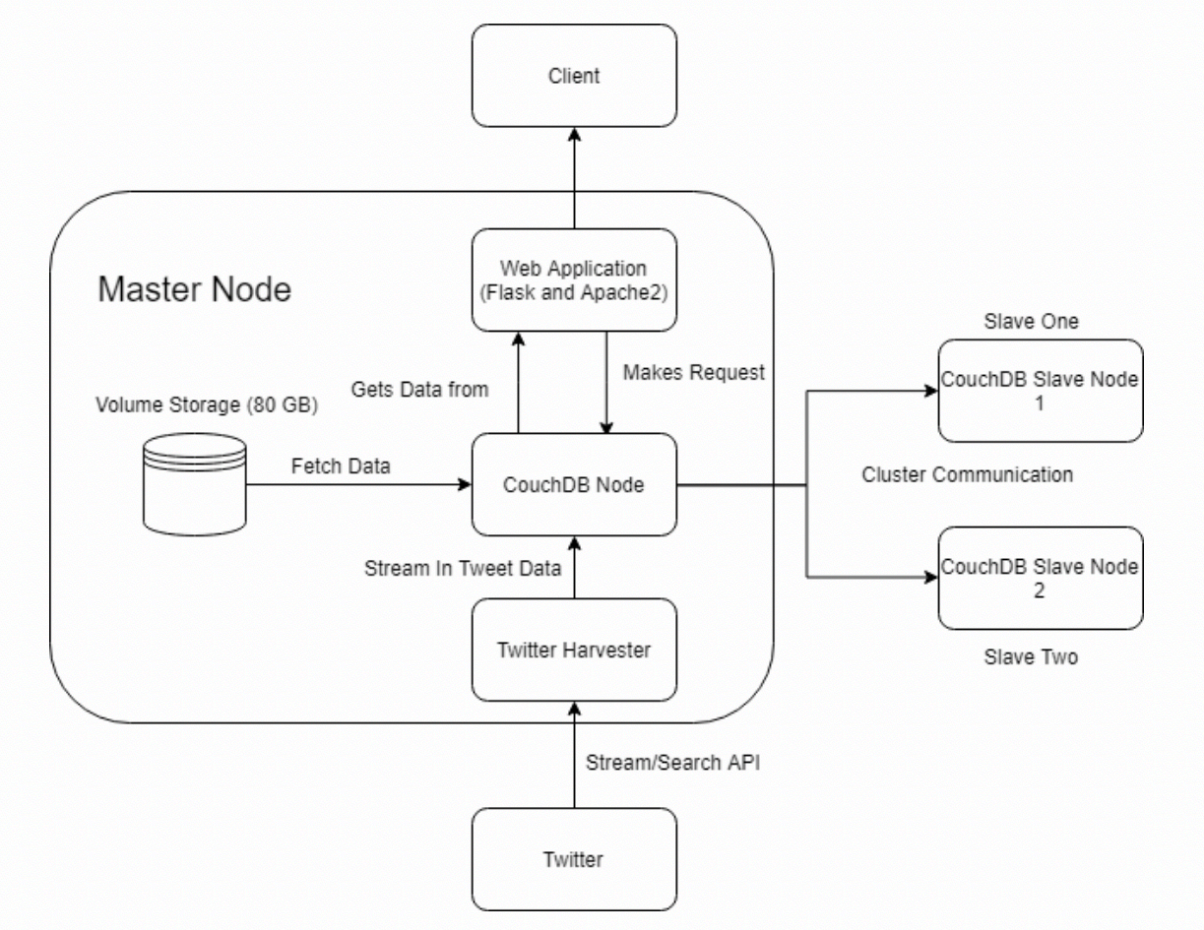

Figure 8. Overview of our master node setup

As seen in Figure 8, the maser node performs and fulfils the client's request on its web application instance just like any other slave node. Additionally, in our system, the twitter harvester resides on the master node and streams processed twitter data into the CouchDB node which is ultimately distributed in our CouchDB database cluster.

Database System

In deciding on our database design, we decided to use CouchDB, a document-oriented NoSQL database architecture. This NoSQL database structure revolves around storing data in documents, a key-store. As documents are not relational, we had the ability to append additional or remove unwanted fields in a document providing us with flexibility in altering our data to suit our scenarios queries. In deciding on CouchDB, we then had the choice of either having a single main CouchDB database that would then be continuously replicated

across other CouchDB nodes or having a CouchDB cluster setup. We decided on the CouchDB cluster setup as it provided us with features that a single database system lacked which was the ability to scale our system. In a CouchDB cluster, a single dataset is split up into 'q' number of horizontal sections called shards and each document is replicated n times across each node (called replicas). With our 'q' set to eight and 'n' set to 3, we have eight shards from our entire dataset with each shard copied three times and stored uniquely on each node. This provided us with a database system that is robust to data losses through node outages as any disrupted node can be joined back to the cluster and could also replicate the existing shards from the other nodes. Moreover, using CouchDB provided a web-friendly interaction between our database and our web application as CouchDB stores its documents as human readable JSON format and interacts via web friendly REST API calls. Scalability can be done as well in our system by just adding slave nodes configured with CouchDB into our current database cluster. In setting up our CouchDB instances on each node, we pulled a pre-existing CouchDB Docker image into each virtual machine instance (we used the CouchDB version 2.3.0 image). Before running the container, we first configured the following security rules on each of our instances on Nectar:

1. CouchDB Ports
    - Allow all to Port 5984
    - Allow all to Port 5986
    - Allow all to Port 4369
    - Allow all to Ports 9100 to 9200
2. Internal Group
    - All TCP Ports within this security group
    - All UDP Ports within this security group
    - All ICMP Communication within this security group

These rules were configured to allow for the respective ports on each node to perform their necessary communication in a CouchDB cluster. Port 5984 was opened to allow for general HTTP database related requests, port 5986 for administrative and node shard management requests, port 4369 for the Erlang Port Mapper Daemon and finally ports 9100 to 9200 for inter-node interactions between CouchDB cluster nodes.

In running our CouchDB image, we mapped the node's external mounted volume path to the internal CouchDB container data path, passed in our desired CouchDB administrator username and password details and mapped the container's ports to the correspond host ports. After running the container, cluster configurations were performed on the CouchDB master node add slave nodes into our CouchDB cluster. (Details in Ansible Playbook)

Twitter Harvester

In our system, our twitter harvester runs on the master node using Python and Tweepy, a third party twitter API wrapper for Python. Due to the limit imposed by Twitter to only allow for tweet collection from the last seven days, it was redundant to have multiple streamers as our pool of available tweets are limited.

Web Application

Our web server was developed using Flask, an easy to use micro-web framework for Python based web servers. Flask too offered ease in writing up a REST endpoint which could easily pull data from CouchDB and send data to the front-end interface via REST API calls.

Apache2 was used to deploy our existing Flask application to the web. This was done using the Apache mod_wsgi server module which enables Apache2 to host Python based web applications. After pulling the web files from our git repository, we then transfer the web files to Apache's /var/www directory, created our .wsgi file for WSGI configuration and created a FlaskApp.conf file to provide configuration details for Apache to run our Flask web application.

Front-End Visualisation
Chart.js was the external script used to visualize data and to enable user interaction. We used it to draw bar charts and a radar chart on our front end web interface. Geochart was another visualisation tool by Google Chart library that we used to draw a map to visualise data based on Australian state regions.

In the following sections, details of each component within this system will be explained.

Tweet Harvester

Technology used

For the project to function, first we need to gather data. There are collected tweet data on the web, but in order to gain more exposure to data collecting and processing technology, we decided to do the collection by ourselves.
After some discussion during our first meeting, it was decided that we will be using python to write the collection application. There are three main reasons for this choice:

1. Most of the team members are familiar with python, so it will be easier for us to implement and debug the collector.
2. Python has a free and open source library, Tweepy, for communicating with the Twitter APIs. There are also many tutorials online for Tweepy with good examples.
3. The couchdb database interaction code will be done with python as well, utilizing the couchdb-python library. It might be easier for us to use a single language for these two components of the project, as they are heavily intertwined.

Collection Methods

At the beginning of the project, we spent several days learning about Twitter APIs and Tweepy, before deciding we will make a collector utilizing both the (free version) streaming API and search API to gather tweets.

The Streaming API

The Twitter streaming API uses a listener system. The caller can make a request to the API, to be registered as a receiver of the stream. The free version of the API sends a small portion (about 1%) of real time tweets to all the receivers, which need to implement their own listeners to process the incoming tweets. There is a limit of number of connection attempts

from the receiving application to the API for each 15-minute time window. Once such limit is reached, new requests will return a 420 error. If too many attempts are made after reaching the limit, the API key will be banned. Since the limit is not by the number of tweets collected, if the receiver functions without error, the limit should never be reached.

The Search API

The Twitter search API provided the functionality to gather tweets from the past. The free standard version allows the calling application to gather data from a pool of tweets for the past 7 days. It also accepts arguments, which are used to specify a geographical filter or a keyword filter. There is a limit for each 15-minute time window, allowing the caller to gather at most 20k to 45k tweets for a single run. There is a risk to get the API key banned if the caller keeps making requests after reaching the limit.

Implementation

We made a single class, SinCollector, to wrap the code of both search and streaming APIs. The two functionalities will be referred as the searcher and the streamer.

After the first meeting, we decided to start the tweet collection as soon as possible, since there are limitations in the free version of APIs. If we want to collect a meaningful number of tweets (e.g. 500k) before the due date, we need to start early. Because at that early stage we still haven't decided on the topic of the project (which sin), we agreed that all tweets from Australia should be collected, rather than focusing on a smaller area like Victory, or limiting to a group of specific keywords.

Since the cloud-instance-based couchdb system has not been built yet by our Ansible specialist at that time, a temporary database using the couchdb docker image has been setup locally to store the tweets.

Streamer

The streamer was built first, since it seems to be easier after our initial research.

As the Tweepy library really wrapped this part well with its built-in StreamListener class, we quickly built our streamer (without filter) and started to test things out.

The first difficulty we faced about the streamer is that for geo-filter arguments, the streaming API only accepts a bounding box (in geojson format) for an area, rather than a string-based keyword like "AU". After some research, we made a bounding box for Australia with the help of a webtool (http://boundingbox.klokantech.com/). The bounding box covered all territory of Australia, and proved to be appropriate after some test runs of the streamer.

We connected the streamer to our local couchdb database by parsing the incoming tweets in the on_data method of our listener. The streamer is proved to be writing into the database without error, however we encountered another problem soon. Though the streamer worked well during the daytime, it crashed at about 7 PM. After some investigation, we found out that Twitter will close connections if the streamer cannot keep up with the velocity of the stream. We believed the problem was caused by doing processing and file writing within the on_data method, which caused the listener to be unable to keep up with the stream during Australian Twitter peak time. To fix this problem, we made the processing multi-threaded, creating a new thread to parse and writing each incoming tweet while allowing the listener to focus on the stream itself.

The streamer with the new fix worked perfectly, until it crashed during an overnight test run. The problem is found out to be a bug with httplib caused by an existing issue within Tweepy's streaming module. To counter it (and other unexpected crashes in the future), we made the streamer to be self-healing by wrapping it with code which will restart the streaming if it stops unexpectedly. To make sure our application won't send too many requests to the Twitter streaming API (which will then get our keys banned), we also added a sleeping mechanism to the code, making it to sleep for increasingly longer time after each failed attempt to restart the stream, and reset the sleep time after the streamer worked successfully for some set period.
The final streamer we built is quite robust. We had it running in the background on the cloud for several weeks (up to now) without any problem.

Searcher

The searcher was built immediately after the first version of streamer was done. We utilized the Cursor module provided by Tweepy to construct the search functionality. We built a subclass of Cursor to handle RateLimitError, which will be thrown when the search limit for a time window is reached. The searcher will be put into sleep for 15 minutes and 5 sec each time the limit is reached, waiting for the time window to be reset.
We didn't encounter many problems with the searcher, as the searching API is easier to use, e.g. it accepts multiple arguments, and the geo argument can be given by a string keyword like "Australia". The only thing worth mentioning is the query size. The Twitter search API allows the caller to specify how many tweets to return for each query, with a maximum of 100. Before we found out this information, we were unaware that we used a query size of 10 which is the default value provided by Tweepy. Since the rate limit is actually by number of queries (not number of tweets in total), we have been running the searcher at 1/10 of its potential for the initial couple of days.

Duplicated Tweets

We built our tweet database in a way that the ids used for docs (tweets) are their tweet message ids (which are unique and sorted according to the time they were created), thus there will be no duplication of tweets in the database. Nevertheless, to increase efficiency (mostly for searcher) we made our tweet collector to check the id when a new tweet is encountered, and discard it if the same id already exists in the database.

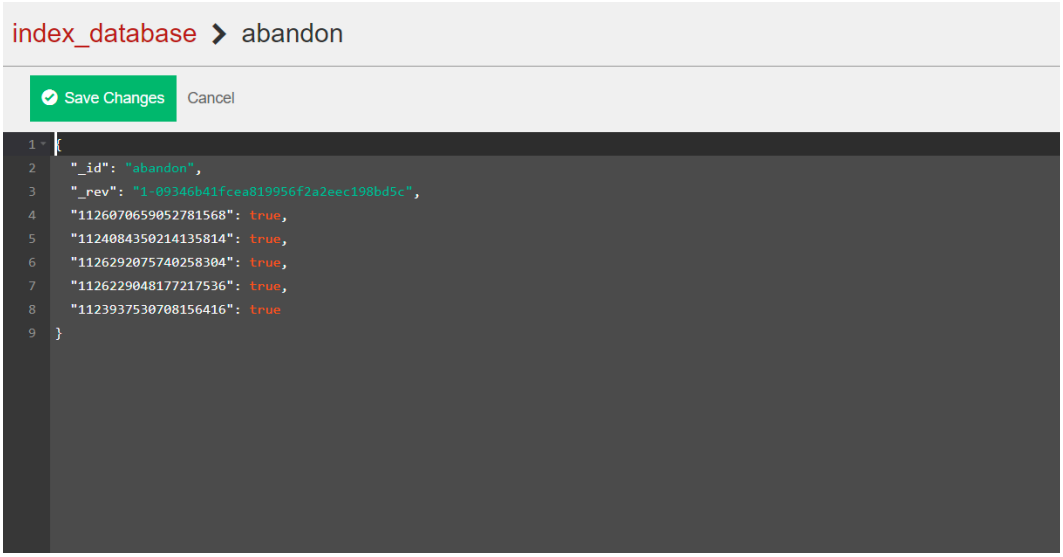

Figure 9. Screenshot of index database (each doc is an index for a word)

If we were to import tweets downloaded from the web (we didn't do that though), we plan to use the parser (which will handle duplicates) within our collector to parse them before writing them into the database.
A note about retweets: our collector was designed to discard retweets by checking the retweet field. It was made this way to suit our project topic.

Containerization of the Collectors

We made a docker image for the collector and pushed it to the dockerhub (our repo does not have the image in it, but it does have the Dockerfile used). However later it was found out that multiple streamer or searcher will probably not increase the tweet harvesting speed: they get tweets from the same pool. Therefore, we gave up on running multiple instances of the collector in a swarm, and simply started the python application on the master node.

Migration of database

When the database system on the cloud was setup and running, we have already collected 200k tweets and stored them locally. As we could not afford to lose these data, the needs to migrate database to the cloud arose.
Initially, we used an approach found online to simple copy the database folder, which is in the mounted volume for the couchdb docker image, to the cloud. When the uploading finally finished, we mounted it to the couchdb on the cloud instances, but it didn't work: the cloud database could not recognize the existing database files. The reason for this remains unclear, but it may be caused by the updates of couchdb (the copy and paste method was introduced in a rather dated stackoverflow entry) which introduced some metadata to recognize the databases.
Then we tried the replication way after doing some research on the couchdb documentation. We created a database with the same name on the cloud, then used the curl -X POST command and the couchdb _replicator to replicate the local database to the cloud, and it worked well. We now have a 3-nodes couchdb running on the cloud to store the tweets.

Data Indexing and Pre-processing

Index database

After the project topic and presentation method were chosen, we agreed to support real time text search over the whole tweet database (only the text field of tweets). The search key will be a list of words provided by the website user.
To make the text search feasible (not taking a long time), we decided to build a word index database along with the main tweet database. The doc structure in the index database is:

- id: a single English word, e.g. “food”
- entries: each entry has the format of: {id of a tweet that contains the word: true}. There will be multiple entries for each word, including ids of all the tweets that contain this word.

The doc is structured like this to make sure a single tweet will not be recorded twice for the same word (as the tweet id is used as the key for a field). Since we have other means to prevent duplicate tweet ids (done during the indexing), this only acts as a last defence (similar with how we handle duplicates in the tweet database).
With such an index database, when we try to search for tweets which contain a specific word, for example “food”, we can simply get all the keys (except “_id” and “_rev”) inside the “food” doc.

Pre-processing

Another feature of our project requires the sentiment analysis results and LGA (local government area, used as a geo code in most AURIN government databases) information for tweets. In order to save time when running website queries, we decided to pre-process the tweets and store the information. Rather than building new databases, we saved them as extra fields directly into the tweet database. This could be achieved using views, but considering we are doing the indexing which already involves going through the whole tweet database, we implemented them together in a single tweet processor to improve efficiency.

Tweet Processor

We built the TweetProcessor to handle all the indexing and pre-processing work.
For indexing, the TweetProcessor will use TextBlob, a text processing library, to get the words from tweet text. Next it will use Enchant, a library for spell checking, to discard all words that are not English. Finally, the tweet id is put into the docs in the index database for all words the tweet contains.
For sentiment analysis, the sentiment module of TextBlob is used. It will generate 2 scores for each piece of text, a polarity and a subjectivity. Polarity is from -1: very negative, to 1: very positive. Subjectivity is similar, ranging from objective to subjective. Due to time limitation, the sentiment analysis method we used is quite naïve, but it can be improved easily in the future by using a machine learning approach or a third-party API if needed, since the code is made to be modular.
We wrote our own code to generate LGA information. We obtained geojson bounding box coordinates for each LGA area from Australian Bureau of Statistics (ABS), then used the ray-casting algorithm found online to check which LGA area a single tweet belongs to. We also used another document from ABS to find out the state (e.g. VIC) information.

The TweetProcessor can work in two modes. Mode 1 is to batch process existing tweet database. In this mode it uses more memory and multi-threading to improve performance. It is also idempotent and supports suspend and continue (e.g. can break the whole processing in two time period, or rerun the code as many times as needed), since it recognizes and skips the tweets it has already processed, and does not do file writing for docs which are not changed from last run. Mode 2 is integrated into a new version of streamer of the SinCollector (it has a TweetProcessor as an instance variable). It will automatically modify index database and do pre-processing work when new tweets are collected from streaming.
We indexed our tweet database (has 530k tweet at that time) with the Tweet Processor in several hours, and started using the newly integrated streamer to collect tweets, so no batch processing will be needed anymore if we don't download tweets from other sources.

Data Analysis

AURIN Data

The dataset of AURIN is based on Local Government Area (LGA). For this project, we consider the data set of income to find the wealth gap, dataset of crime which consists of sexual offences, person offences, robbery and extortion for finding data on Lust, Wrath and Greed respectively. We also take adult health dataset to get the rate of sloth.
Since is data is primarily in LGA and SA4(Statistical Area) we normalize it to state level via grouping.
It is also noted that AURIN dataset is limited to some states in some cases such as crime rate is only available for South Australia.

Tweets

The tweets under analysis are real-time tweets with a max age of 20 days since the day was tweeted on Twitter. Each tweet consists of numerous fields such as text, location, user-id, date and time, and similar fields. We are interested in the location and the content of each tweet the user has tweeted.

Location Information

For the location, it was noted that most of the tweets didn't have precise coordinates or location information. Thus, we calculate the rough location based on the bounding coordinates of the tweet. The bounding box coordinates are the coordinates of an area where the user may be located. This method of finding the location is quite accurate when we need data divided into state-level. We also observed that almost 90% of the tweet have bounding box is defined thus making it possible to use the location of the tweet. The information is collected at the state level since its AURIN database provides data on Local Government Area (LGA) level, but there are not enough tweets in each LGA for a clear contrast. Thus, the data is mapped to the states of Australia.
It is also found that Northern Territory has the least number of tweets to be used for leveraging thus we avoid that state in this project.

Sentiment Analyze

The content(text) of the tweet is the next aspect which we will be analyzing in this project. The tweet text tells us about the user's opinion, his/her thoughts, behaviour and much more.

Therefore, for analyzing the tweet in depth we carry out the sentiment analysis of the tweet using Text-Blob. Subjectivity and polarity of the text are calculated based on the sentiment of the user. We can further explain this by, when a user posts a text like “I hate Monday”, polarity: (-0.8) and subjectivity: (0.9) is derived. Here polarity defines sentiment the user is feeling which can be in the range of between [-1 to 1], where -1 showcase very negative sentiment, 1 showcasing very positive sentiment and 0 being the neutral sentiment. Value of subjectivity shows the users personal feelings, views and alike where the text describes a fact. It ranges from [0 to 1].

Processing

We store the LGA code, LGA name, state name, state code as well as polarity and subjectivity for further segregation and finding the sins within the tweets. For accelerating the processing of tweets and easy lookup we implement an index DB which will hold CouchDB id for every tweet. The Index DB is built based on every word which appears in the tweet. The use of this implementation will be further discussed in the scenarios for this project.

Scenarios:

1) Find the percentage of four sins committed in each state(sentiment Analysis).

The sins under consideration are Sloth, Greed, Wrath and Lust. To find these sins in tweets primarily we collect a group of words (see appendix - citation) which directly signify the sin based on sentiment in the tweet. This word is searched within a tweet and the sentiment is also taken into consideration for terming the where the tweet is sinful or not. For example, when a tweet has swearing words and has negative sentiment, we classify it has Wrath sin tweet. Therefore for Sin of Wrath, the wordlist consists of swear words and we consider the negative sentiment. In the case of Greed, the wordlist consists of wealth describing words thus positive sentiment is taken. For Sloth, the wordlists talk about words pertaining to being ‘active’, thus tweet consisting of an ‘active’ with negative sentiment can be classified as a Sloth sin Tweet. In case Sin of Lust, the word list words are taken directly taken into consideration without sentiment.

The ‘sinful’ tweets are divided by state based on their location. Percentage of sinful tweets to the total number of tweets in that state is calculated.

This percentage is then visualized with the normalized (state wise) AURIN data for comparing.

2) Searching the keyword:

In this method, the user registers a keyword (a word signifying a sin such as ‘hate’) and he/she can enter the sentiment to be taken into consideration. The sentiment bar supports (0-100) which is then normalized to [-1,1] so that we can collect tweets having this keyword and sentiment. The tweet database is searched for this keyword and the resulting data is returned. Again, this data is divided among the states and we calculate the ratio of sinful tweets based on the keyword to the total tweets in that state. Since AURIN data is limited to some states we limit the user from selecting other states which are not supported for data comparison.

For speeding up the above processes we use index DB lookups to improve the execution speed dramatically.

Deployment

One major objective that we would want to achieve in our project was to reduce the inefficiencies in system setup through automation or dynamic deployment. Manual system setup proves to be time consuming and inefficient as human actions are prone to mistakes/typo errors and more. As more features are added into our system, more time is required for a user to type in the required configurations and eventually causes more complications. With an automation scripting tool (such as Ansible), we can now have a consistent set of instructions carried out to deploy and set the system up. This would ensure that any machine or system would be in the same consistent state as long as the same setup script is used making the process easily repeatable. Automation scripting tools also allow us to easily add new functionalities or features to the system as we do not have to go through the whole set up procedure from scratch manually.

Ansible was chosen as our automated scripting tool to dynamically deploy our system. Using Ansible was extremely easy as it uses YAML, a human readable mark-up language and allowed for easy feature additions as well. We were able to re-create our whole system using Ansible from nothing and the newly created system would have the exact configurations that we wanted or specified.

An Ansible folder structure would have these major components:

Playbooks have:

- Roles
- Variables
- Tasks
- Templates
- Inventory

Generally, Ansible would first look into a playbook, which contains a series of plays. A single play is a set of instructions defined to run on a particular target host machine. Roles are a set of tasks that the play would run on the defined host. Variables are just like any other programming language where it holds a certain value to be used throughout tasks. A task is a single instruction. Templates are files that contain configuration parameters that you would want to copy into a target host. Finally, an inventory file is a file that keeps play book items (such as keys or IP addresses) to be used throughout tasks.

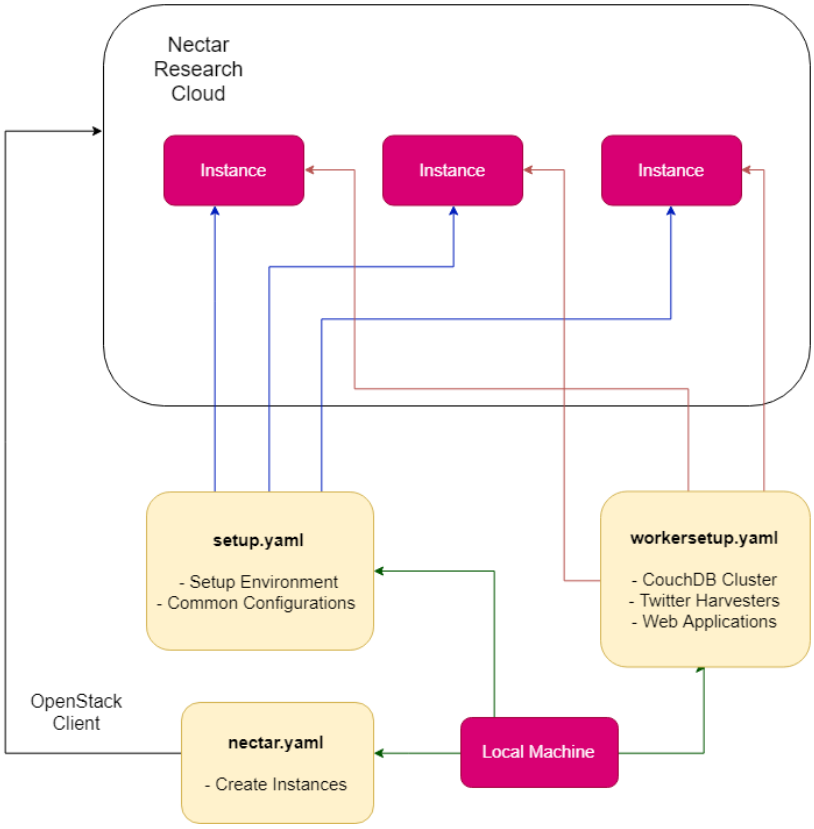

Figure 10. Ansible Workflow

In our Ansible workflow, we first had three main playbooks. The playbooks in chronological order are:

- Nectar.yaml
- Setup.yaml
- Workersetup.yaml

Nectar.yaml sets up and deploys instances on the Nectar Research Cloud using the OpenStack Client. It runs tasks such as creating a key pair, creating security groups, volumes and instances. Next, Setup.yaml takes over and performs common configuration setup steps on each machine such as proxy configurations, volume management, installing required dependencies and Docker. Finally, Workersetup.yaml finishes the setup procedure by setting up the CouchDB cluster system, running our web-servers on each node and running the twitter harvester on the main master node.

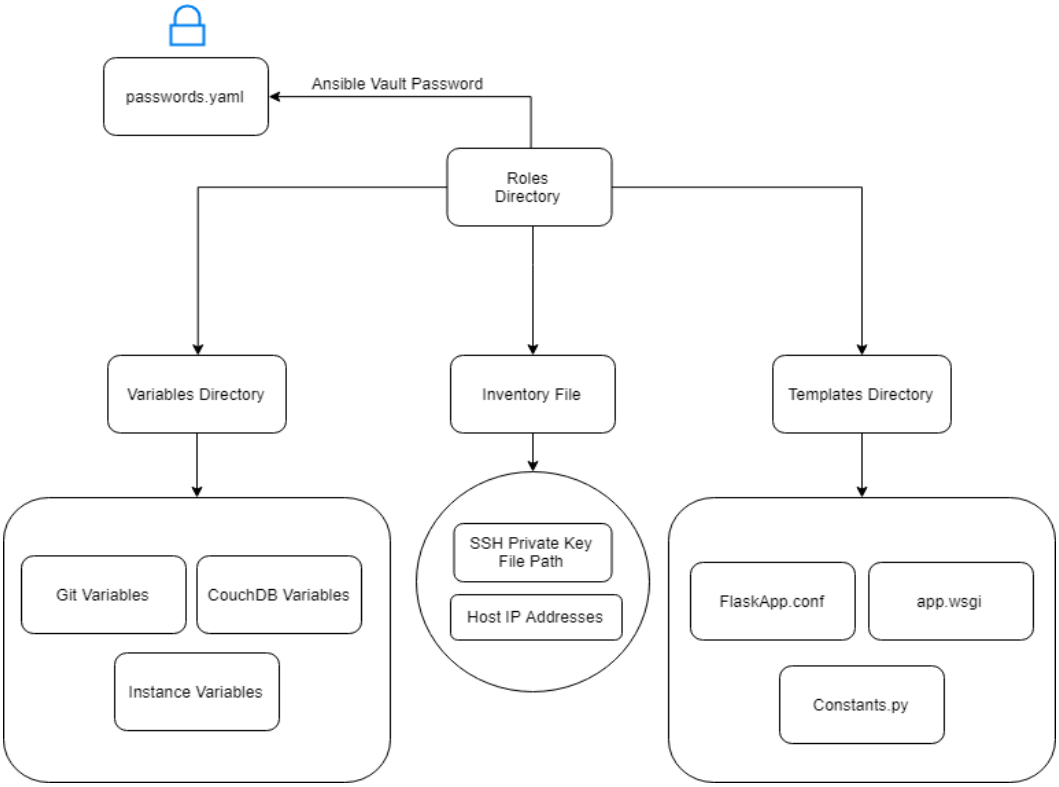

Figure 11. Ansible Playbook File Structure for our system

Our overall Ansible playbook is structured like in Figure 11. We have three main directories, roles, which possess the roles we use in our playbooks. These roles group together tasks that contain the necessary steps to configure our environment.

Our variable files (in YAML form) are located under our variables directory. Git Variables are used to set out git parameters (such as git repository link) to successfully pull our project repository into each machine while CouchDB variables are used to store our CouchDB variables (such as username) to successfully run the CouchDB setup. Instance variables are where all instance variables live (such as volume names and sizes, instance configurations and security groups) which would be used primarily by Nectar.yaml to create the instances on Nectar.

The inventory file consists of the target host IPv4 addresses that the playbooks would use to target and run plays. It also contains the path to the private key on our local machine.

The template directory consists of the required files that we would be copying over to each remote machine to perform configurations. In this case, we had FlaskApp.conf and app.wsgi configuring our Apache2 configurations and Constants.py containing twitter API keys from our passwords file that we need to run the twitter harvester.

Last but not least, all passwords and sensitive credentials are kept in our passwords.yaml file. This password file is encrypted by Ansible-Vault to allow for security in storing sensitive information in our playbooks. Currently it stores the CouchDB password, Git Password and Twitter API credentials. Our key to decrypt the file is Group 68.

NeCTAR Research Cloud

NeCTAR, which stands for National eResearch Collaboration Tools and Resources, is an online platform that provides technological services and resources catered towards the field of

research primarily in Australia. As research today involves handling large amounts of datasets, researchers and academics require powerful computation tools to generate the required results from huge amounts of data. NecTAR removes the cost for researchers need to have large computational power as it provides this targeted group the needed infrastructure and computational resources online to produce the needed results for analysis and research. An example of the many resources provided would be Spartan, a high performance computing system which can provide the necessary computational power to run heavy tasks. NeCTAR cloud is a cloud service platform that provides cloud services to academics and researchers that may need technology or infrastructure to store large amounts of data while also having the benefit of remotely accessing them and processing them. In this project, we were given the opportunity to use NeCTAR cloud as a platform to host our application and store our data remotely. While using NeCTAR throughout our project, we had experienced different advantages and disadvantages. Under advantages, we observed the following:

1.Ease of setting up wanted environment
2.An easy to use and user friendly online web interface
3.Well documented tutorials and documentation on NeCTAR's website
4.Ease of remote access on resources
5.Many tools provided such as operating system images, load balancers, databases and more
6.Easy to interact with resources using REST API Calls through OpenStack
7.Backup tools such as volume snapshots and backups

One of the most impactful advantages that we clearly observed was the ease of setting up our desired working system's environment. As users, all we did was to click on buttons and filled up forms to then let NecTAR create the desired environment that we wanted through their online web interface. Examples include setting up keys, security groups and choosing operating system images through buttons in which NecTAR does the rest of the heavy lifting in configuring it for users. The online interface on Nectar was also very intuitive and easy to use. Dangerous requests such as deleting instances, keys and volumes are also warned by NeCTAR to the user if a user accidentally clicks on them to protect theuser from accidental misconfigurations in their system.Online tutorials were also helpful and well detailed on the NeCTAR website. These documents enabled us to easily know the required steps to perform a certain task such as creating volumes, creating instances and more. Pictures of sample configurations and Linux based commands too helped us in perfectly setting up our environment.
Furthermore, NeCTAR is available across Australia which implied that our resources could be accessed by any client based on our chosen availability location. As our location was set to Melbourne, we had the benefit of providing our solution to the clients that reside in our availability location. NeCTAR too provides users with a huge list of tools that you may require for your system. Items such as load balancers, operating system images, databases and more are all available for use in setting up your wanted system. Other than their interactive web interface, NeCTAR allows users to also manage their resources through API calls instead. This is beneficial in terms of auto deployment as we can configure the necessary commands and endpoints to make requests and generate our environment through automation tools such as Ansible. These API management endpoints are managed and configured by OpenStack, a cloud operating system.Last but not least, NeCTAR provides users with backup storage and volumes snapshots that can be used by users to keep safe copies of their data. These backups

and volume snapshots can also be retrieved any time and users can re-deploy their systems using these backups in case of any failure. However, there a few disadvantages that we also observed which were:

1.Data Security issue
2.Requires advanced knowledge in networking (such as opening ports)
3.Issues when transporting data around
4.No physical control, handled by NeCTAR team

A major issue was the lack of physical hardware control on our resources. If in any case sudden outages occur, we as end users are not able to fix any of the issues as we do not have the permission to physical access them. These outages may also result in our system data files being corrupted while running a certain process or task.In setting up our system, end users are required to know advanced networking setups in order to have a well-protected working system. If a user exposes the wrong ports, their system would then be vulnerable to unauthorized access and may have possible corruption in their data and system. Transporting data was also an issue as bandwidth is limited when sending data to the cloud. This meant that large datasets would require a huge amount of time tofully transport successfully. Any outages that occur during the transmission would then result in possible data loss or corruption. Finally, data security is also a major issue. As these resources are handled and managed by an external team, users would not know how safe their data would be on their applications. Any leak that occurs due to a fault on the team's side would be out of the user's control. Thus, users should be wary of the risks in data protection that they may face in storing data on the NeCTAR cloud.

Appendix A - User Deployment Guide

CouchDB and Web Servers Deployment

Firstly, user must first ensure that Ansible is installed on the local machine. Once Ansible is required, go into "*deploy/Ansible/variables*" to adjust and set your parameters.

Under the "variables/couchdbDetails.yaml" variable file, set the username and password for an admin account on the CouchDB cluster as well as the number of nodes you intend to have in the CouchDB cluster.

Next, once move to "variables/gitDetails.yaml" to adjust the GitHub username that you would want to use to pull our project repository.

Once that is done, adjust instances parameters under "variables/instancedetails" yaml file. Here you must set the following parameters:

1. Set your current Ubuntu username under the local_user field
2. Set the name of the SSH key you would want to create under Ansible_key_name

NOTE: ensure that this key does not exist on Nectar or your current playbook directory as it would return an empty private key

3. Set the volumes you would want to create, (append to list with each having a name and size)
4. Set the instance parameters for each instance to be created under instance_list

These would be the names of the instances hosting CouchDB and the web application.

The last one on the list would be set as the "master" node.

Set the instance name, image ID, flavour and volume to be attached.

Finally, decrypt the current Ansible-vault protected "variables/passwords.yaml" file by running the command:

Then, type in the password "group68" to decrypt the password file and set your CouchDB password, GitHub account password and the Twitter API credentials.

Once all variables are set up, run the command:

And type in your sudo password and openstack password to launch the playbooks to set our system up. This command would create the instances on Nectar, configure each virtual machine, setup the cluster and setup the web-server on each node. The twitter harvester is also set up on the user defined master node. The total process time will take approximately 10 to 15 minutes.

After running the playbooks, user can now check if our CouchDB instances are set up by entering the following URL into the user's browser and setting the <Node_IP_Address> field to the any of the IP addresses in the inventory.ini file.

Log into the Fauxton interface and click on the databases tab. Refresh the page to see the tweet_database document count rise as it streams in tweets.

To check for cluster setup, run the following curl command:

If all node IPs addresses in the all_nodes field matches the node IPs addresses in the cluster_nodes field, then we can confirm that we have a successful cluster setup.

Finally, to check for our website setup, browse on to any of the node's IP addresses and append "/app" to see our front end web application.

User can now interact and use our front-end application.

Appendix B - Team contributions

Darren Pinto: Tweet Analysis, Views of CouchDB
Matthew Yong: Automatic Deployment, Ansible Script
Mengzhu Long: Flask, Front End, Visualization
Wenhao Zhang: Flask, Web, AURIN Data Processing
Yang Liu: Tweet Harvester, Indexing, CouchDB

Appendix C - Links

Videos on Youtube link:
https://www.youtube.com/channel/UC5s46Qw660NS0FOShCxEniw

Project Codes on Github:
https://github.com/mchozhang/SinsOnTwitter

Appendix - Citation

Details on Sentiment Analysis. Retrieved from: https://medium.com/@rahulvaish/textblob-and-sentiment-analysis-python-a687e9fabe96

TextBlob Library For Sentiment Analysis
https://textblob.readthedocs.io/en/dev/

Wordlist Related to Lust:
https://github.com/mchozhang/SinsOnTwitter/blob/master/src/resources/sin_wordlists/SexualDirty_Lust_Wordlist. Retrieved from: https://github.com/LDNOOBW/List-of-Dirty-Naughty-Obscene-and-Otherwise-Bad-Words/blob/master/en

Wordlist Related to Wrath:
https://github.com/mchozhang/SinsOnTwitter/blob/master/src/resources/sin_wordlists/curse_word_list_Wrath. Retrieved from https://github.com/RobertJGabriel/Google-profanity-words

Wordlist Related to Exercise, Work or Study:
https://github.com/mchozhang/SinsOnTwitter/blob/master/src/resources/sin_wordlists/exercise%26work_Sloth_wordlist. Retrieved from: https://www.merriam-webster.com/thesaurus

Wordlist Related to Money:
https://github.com/mchozhang/SinsOnTwitter/blob/master/src/resources/sin_wordlists/money_greedy_wordlist. Retrieved from: https://www.merriam-webster.com/thesaurus