COMP90024 Cluster and Cloud Computing

Project 2 - Team 52 Report

Xiaotian Li
1141181
Xiaotian.li5@student.unimelb.edu.au

Rui Zhang
1294310
zhang.r9@student.unimelb.edu.au

Huahu Wen
1215209
huahu.wen@student.unimelb.edu.au

WeiFeng Wu
1232129
weifeng.wu@student.unimelb.edu.au

Bocan Yang
1152078
bocan.yang@student.unimelb.edu.au

The University of Melbourne

April 30, 2022

Abstract

This is the report of the COMP90024 Group 52 project. In this project, we have developed a system for exploring liveability in Melbourne, which is a cloud-based solution to harvest tweets and analyze Twitter data with official statistics data from the AURIN system in the Melbourne Research Cloud (MRC). The system consists of comprehensive technologies including Ansible auto-deployment, distributed CouchDB cluster with built-in Map-Reduce functionality, RESTful API designs, Docker, Nginx load balancing, Flask backend server application and React.js frontend.

In Section 1, we will introduce the structure and design of the whole system, in each small chapter the detailed program module and techs will be described. In Section 2, we give the user guide on how to use and how to deploy our system on the MRC by different modules (e.g., how to deploy the instance of a Cloud server or the frontend application). In Section 3, we will provide the illustration of the functionality of our system, in other words, our thoughts on analyzing different scenarios and why we choose them. Moreover, some real charts in our system will be displayed. In Section 4, we are going to discuss some difficulties and errors occurring during the development and how we addressed them, also, the properties of our project like scalability will be discussed. Finally, in Sections 5 and 6, the contribution of each member and video links will be displayed.

Keywords: Melbourne Research Cloud; Twitter; CouchDB; Flask; Docker; Ansible

Contents

1 System Design and Architecture

1.1 System Architecture

The architecture diagram for the system deployment is shown in figure 1.1.1

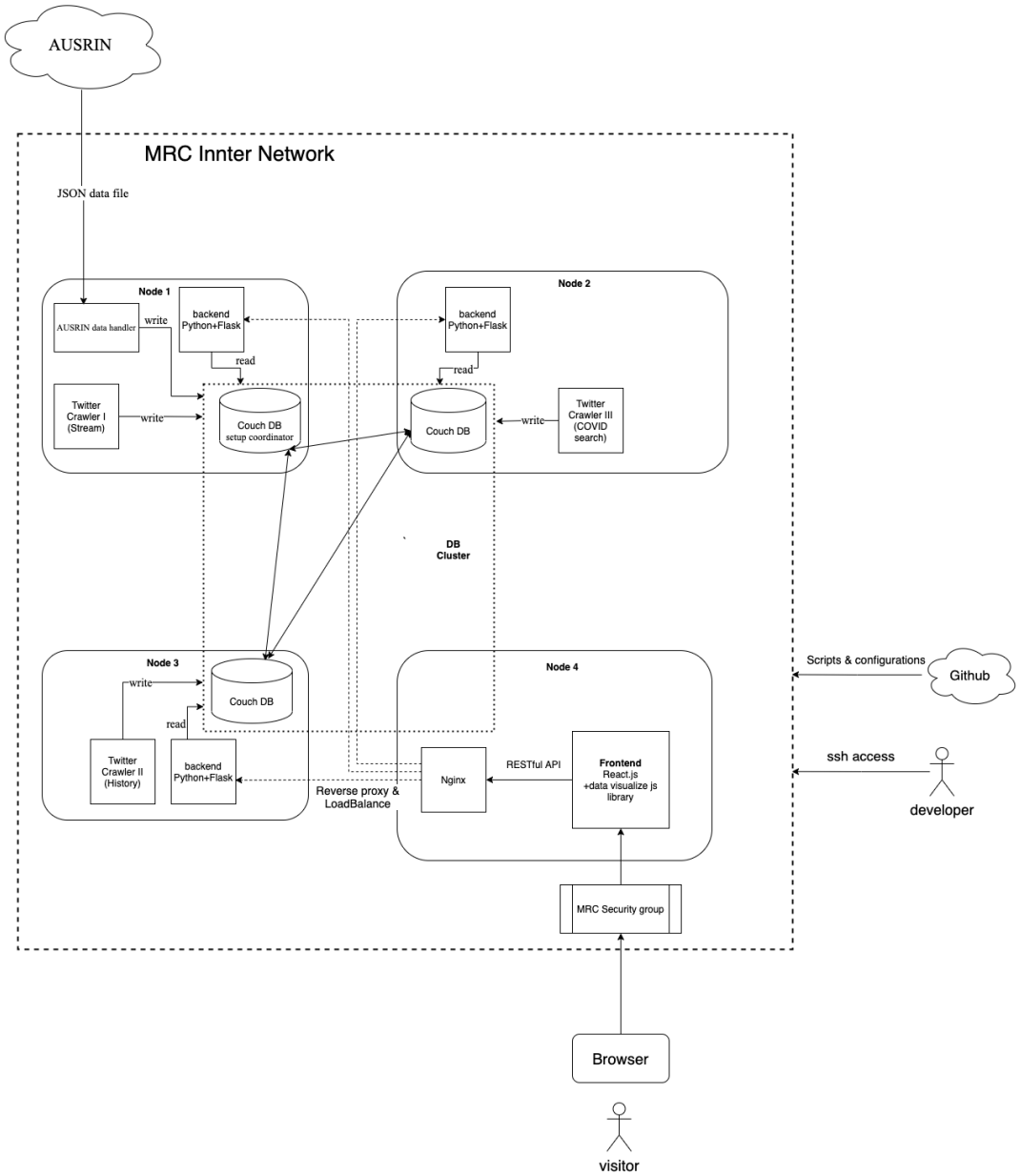

Figure 1.1.1 System Deployment Architecture

As is shown in Figure 1.1, we use three MRC instances in the project. Node 1, Node 2 and Node 3 serve as a back-end cluster that provides data fetching, data persistence and calculation services. Node 4 acts as a front-end server, which provides data visualization for service for data sets and scenarios.

We use a GitHub repository to publish system scripts and configurations to all nodes, and run different modules of the system predefined by Ansible scripts. Using GitHub repository helps deployment because it can be accessed by all server nodes, and git clone operation is easy to achieve.

We have three types of Twitter Crawler: Stream Crawler, Search Crawler and Historical Crawler:

- Stream Crawler: Continuously fetch latest tweets in Melbourne via Twitter Stream API. Developers can define keywords and other behaviors in configuration file;
- Search Crawler: Search COVID-19 related tweets in Melbourne via Twitter Lookup API. Developers can define keywords in configuration files.
- Historical Crawler: Traverse all tweets in Melbourne between 2014-2017. Developers can define keywords in configuration files.

We will run these 3 different crawlers in 3 instances to balance server load and guarantee performance. Crawlers will run in the Docker container to unify the run-time environment.

As for data persistence, a 3-node CouchDB cluster is deployed to enable data sharing in the distributed system. Single Leader Replication pattern is used to guarantee high consistency of data input, i.e. we write data on CouchDB node in server node 1. On the other hand, We can read data from any node of couchDB to ensure accessibility of the database service. For security, the front-end server is not deployed with the CouchDB node because the front-end server node is usually exposed to outside visitors and can be fragile on data leakage .

As for the back-end application, it is deployed in 3 server nodes to ensure accessibility and balance server pressure. It reads data from the database cluster via Round-Robin pattern to control the load of the database, because the database will perform map-reduce calculation. A Nginx reverse proxy is deployed in the front-end server to spread requests to the back-end application equally, so that it can also balance the load of the back-end application and guarantees accessibility.

Front-end application is deployed in the server Node 4. RESTful API is used to achieve communication between front-end application and back-end service. A JavaScript library for data visualization is used to increase efficiency of development.

We use data from Australian Urban Research Infrastructure Network (AURIN) to help analyze scenarios. We deploy a handler application to parse JSON file from AURIN, analyze data and store them to Database for data sharing with high performance.

1.2 Deployment Details

In this project, we are given access to Melbourne Research Cloud (MRC), a largely Infrastructure-as-a-Service (IaaS) managed by University of Melbourne Research Computing Services, to deploy our system. Our system can be deployed with script through Ansible on a remote host.

1.2.1 Resource Allocation

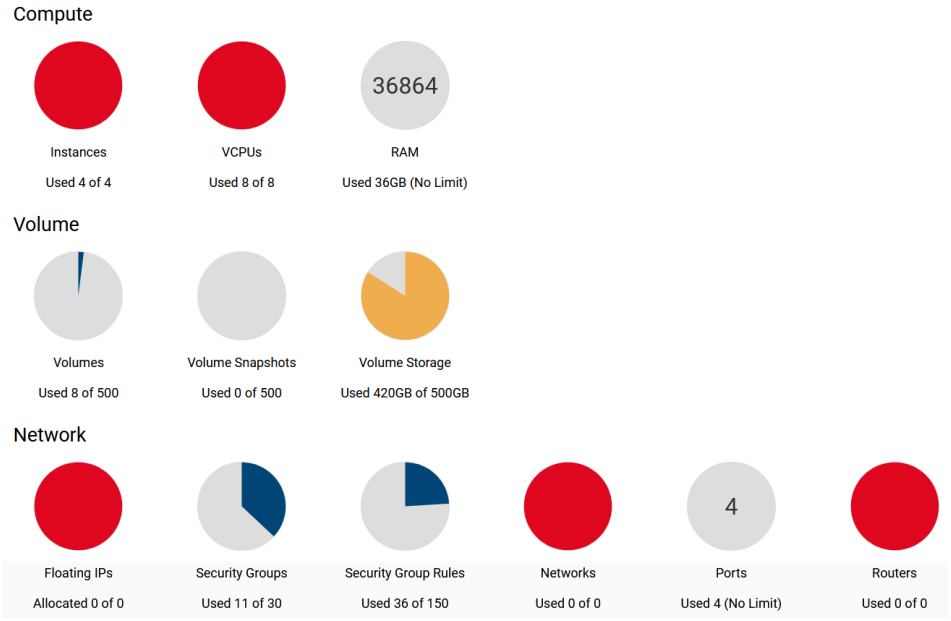

Figure 1.2.1 Resource Usage

As is shown in figure 1.2.1, MRC provides us with 4 instances, 8 virtual CPUs, unlimited RAM, 500GB of volume storage in total. We fully utilize most of the resources except part of the storage for backup. As stated in part 1.1, our system is composed of 1 frontend node and 3 backend nodes. Each node is given 2 VCPUs and 9 GB of ram. Because the CouchDB cluster, AURIN data, and historical data are all on the backend, we allocate 120GB of volume to each backend node and 60 GB to the frontend node.

1.2.2 Discussion of MRC

In terms of costs, MRC provides free resources for researchers and students. It is funded by the university and mainly for research and education. MRC provides most of the essential functionalities equivelent to commercial IaaS for free. So, it is beneficial to students.

```
export OS_AUTH_URL=https://keystone.rc.nectar.org.au:5000/v3/
# With the addition of Keystone we have standardized on the term **project**
# as the entity that owns the resources.
export OS_PROJECT_ID=68370a1695224f7cbbafb6ad86d88e3e
export OS_PROJECT_NAME="unimelb-COMP90024-2022-grp-52"
export OS_USER_DOMAIN_NAME="Default"
if [ -z "$OS_USER_DOMAIN_NAME" ]; then unset OS_USER_DOMAIN_NAME; fi
export OS_PROJECT_DOMAIN_ID="default"
if [ -z "$OS_PROJECT_DOMAIN_ID" ]; then unset OS_PROJECT_DOMAIN_ID; fi
# unset v2.0 items in case set
unset OS_TENANT_ID
unset OS_TENANT_NAME
# In addition to the owning entity (tenant), OpenStack stores the entity
# performing the action as the **user**.
export OS_USERNAME="zhang.r9@student.unimelb.edu.au"
# With Keystone you pass the keystone password.
echo "Please enter your OpenStack Password for project $OS_PROJECT_NAME as user $OS_USERNAME: "
read -sr OS_PASSWORD_INPUT
# If your configuration has multiple regions, we set that information here.
# OS_REGION_NAME is optional and only valid in certain environments.
export OS_REGION_NAME="Melbourne"
# Don't leave a blank variable, unset it if it was empty
if [ -z "$OS_REGION_NAME" ]; then unset OS_REGION_NAME; fi
export OS_INTERFACE=public
export OS_IDENTITY_API_VERSION=3
```

Figure 1.2.2 OpenStack RC File

MRC is built upon OpenStack which is one of the most popular IaaS platforms. Like many other IaaS, we have the dashboard to manage instances, images, volumes, networks, security groups, etc. In addition, MRC supports OpenStack API to enable remote command line or automation tools for management. Figure 1.2.2 shows the OpenStack RC file for remote command line control over the cloud resources.

Most of the resources in MRC are very scalable and flexible. We can choose different flavors for every instance. There are also a large variety of instance images available. Different security groups can be generated with specified directions, ports, and ip ranges. They can also attach to different instances. Volumes with specified storage can attach to or detached from instances.

However, MRC does have some drawbacks. Different from commercial IaaS like AWS with fancy GUI and many plug-and-play functionalities, MRC provides just the essential resources. Users must build most of the complex functionalities on MRC from scratch themselves. During deployment in our project, we also find that the resources have to be carefully allocated at first. The size of volumes is difficult to scale up/down once created. Therefore, we spare some storage for data transfer in case of instance failure.

Although the MRC is not as easy to use as commercial IaaS, it turns out to help us get familiar with cloud infrastructures with just enough resources.

1.3 Software Architecture

1.3.1 Twitter Crawler

The architecture diagram for the Twitter Crawler is shown in figure 1.3.1.

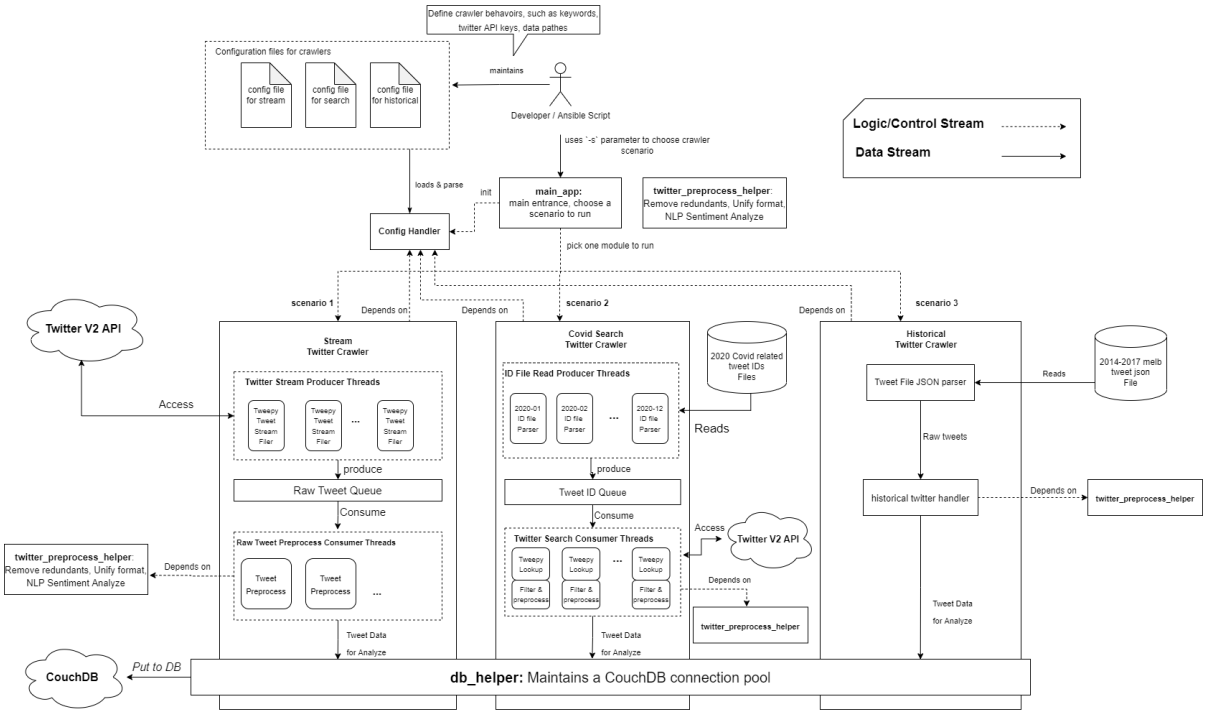

Figure 1.3.1 Architecture of Twitter Crawler

We designed a multi-functional Twitter Crawler application associated with CouchDB for the project. It can be deployed in different MRC instances, and have different behaviors based on parameters and configuration files. The crawler will harvest tweets in 3 ways:

- Stream Twitter Crawler;

- Search/lookup Twitter Crawler;
- Historical Twitter Crawler.

Stream Twitter Crawler: Stream Twitter Crawler uses Tweepy library to query Filtered Stream API of Twitter API V2. Since we only have essential access, the twitter API's functionalities are extremely limited:

- Rate limitations: 25 requested per 15 minutes makes it inefficient to harvest tweets.
- Functional limitations: Can only use core operators, and can not access archived tweets, which make it challenging to fetch tweets based on geographic location, and also impossible to fetch historical tweets.

To overcome rate limitation, we need to increase efficiency of the Tweet harvesting operation. The multithread and producer-consumer patterns are applied to ensure performance. We make Tweepy tweet harvest logic into multiple threads as producers, and raw tweet preprocess logic into multiple threads as consumers. In this multi-thread model, raw tweet preprocess logic will not block tweet harvest procedure. To achieve time and space decoupling between producer and consumer, we used a blocking queue to store raw tweet data.

To overcome functional limitations, multiple solutions are tried. As for geographic location, since we can not get access to the geo filter of the stream API, keyword matching is used as the alternative. i.e. "#Melbourne" keyword filter is applied to fetch melbourne related tweets. We tried to manually parse raw tweets' geo field to get tweets in melbourne but it is impossible, because there are enormous amounts of tweets around the world that will be streamed in, as a result, parsing them one by one is not acceptable.

To dynamically define behavior without manipulating source codes of the Stream crawler, a yaml configuration file is used. As is shown in the Figure 1.3.2:

```yaml
# for twitter streaming , select api version 1 or 2. Now 2 only.
api-level: 2
# is the api key has Academic Research access? true or false
got-academic-access: false

api-key: bfa9kSteSfb1SXyVCfxfr1mWs
api-secret: bYwtGehL0L3eNremqAk0nBOcXurwRIGvLvfNjgcDh0FSVXn0y8
api-token: AAAAAAAAAAAAAAAAAAAAAMp1bgEAAAAAqWJoKo%2F7IgG7Z7%2FKIq
access-token: 1515881246039359488-lMhG3MeErJvfPqpfe0CG9qXGd8rksj
access-token-secret: kbeccnUWeo4ZuwKUhT47M08yWiRVcTmGiDJ7KMec0sAg

# don't input to many
# if key-word is an empty list, i.e. [], get all twitter
key-word-list:
  - covid

target-db-name: stream_debug_db

# 0~100, very loose to very strict match
key-word-match-degree: 40

# true or false
is-fetch-english-tweet-only: false

# melbourne place box
target-box-points:
  - longitude: 144.59374
    latitude: -38.43385
  - longitude: 145.51252
    latitude: -37.51127



db:
  username: admin
  password: 0F9YUJIgGoxIhgRWfHKS
  port: 5984
  host-list:
```

Figure 1.3.2 Configuration file snippet of Stream Crawler

In the yaml configuration file, we can define twitter API level used (currently we only implemented

API v2), twitter API keys, filter keywords, geo box, database information and so on. After deployment of the crawler application, users can easily modify the configuration file to harvest tweets they want.

Search/lookup Twitter Crawler: The crawler can harvest all COVID-19 related tweets of melbourne in 2020. To combine the historical data from AURIN with COVID-19 related tweets, we need to search tweets in some historical periods such as 2020, however, we only have essential access level to Twitter API, we can only perform recent searches.

To achieve COVID-19 related tweet search in 2020, we turned to open-source projects for help. There is an open-source project that continues searching COVID related tweets' IDs around the world starting in 2019 (https://github.com/echen102/COVID-19-TweetIDs). To harvest COVID-19 tweets in Melbourne, tweet ID files of 2020 are downloaded from the open-source project. crawler will parse the ID, and call the Twitter API's lookup functionality to harvest tweets, then the crawler will filter the tweets both on the geo location of Melbourne and the keyword "#Melbourne". Then we can search COVID-19 related tweets of Melbourne in 2020.

What's more, Twitter Official has a looser rate limitation on Lookup API, compared to Search API, so search tweet IDs seems to be a more efficient way of harvesting COVID-19 related tweets.

In the crawler, we also used the multi thread model and producer-consumer pattern to ensure performance. Since file I/O is expensive, we use a multi thread model in ID file parsing as a producer. As for the consumer, multi thread based lookup logic is implemented. Similar to Stream Crawler, A blocking queue of Tweet ID is used to decouple time and space.

A yaml configuration file is used to control the crawler's behavior, such as keywords, database information and so on.

Historical Twitter Crawler: A large corpus of historical twitter data for melbourne is provided. The Historical Twitter Crawler will parse the data file, preprocess it and put the data to the database for analysis.

Also, we have multiple reusable modules in the crawler application, such as config handler, twitter_preprocess_helper and db_helper.

Config Handler: A singleton helper class to parse the yaml configuration files.

Twitter Preprocess Helper: A set of static helper functions that can remove redundant data of raw tweets, unify tweet format and perform NLP sentiment analysis.

Removing redundant data can save storage space, and make database query operation faster because of lower data payload.

Unifying tweet format is also important because we have 3 different incoming raw tweets, and having a unified format in database documents can make it easy for backend to implement data analysis logic.

NLP sentiment analysis can get polarity value from the tweet text. The crawler follows steps below to perform sentiment analysis:

1. Use regular expressions to discard special characters of tweets, like urls, "RT", special characters, etc;
2. Correct typos;
3. Tokenize sentence into words;
4. Remove stop words such as "oh", "he" and "the";
5. Standardize words, i.e. remove redundant tense information;
6. Reconstruct sentence;
7. perform sentiment analysis and give the polarity value.

DB helper: A singleton object that maintains database connection, and provides the implementation of

putting data to the CouchDB.

1.3.2 AURIN Data Handler

For the AURIN data handler, it is basically a one-time-run python script that reads csv files, extracts potential necessary data from the AURIN system, and then puts it into the local node of the database. Due to the design of our system, the CouchDB cluster can be randomly accessed and synchronized in a short time, so the script can be run in any nodes of the MRC server as an initiation of AURIN databases.

It mainly consists of 3 parts:

Config: There is a python file storing the path of the csv files downloaded from the AURIN system and giving the database name of different types of the data for database initialization. Moreover, we select some data columns from the whole csv file and change their name to another shorter and understood way as shown in Figure 1.3.3

```
# 2014 - 2017 house/unit sale/rent price data
HOUSE_PRICE_PATH = '/data/Aurin/House/'
HOUSE_PRICE_DB = 'aurin_house_price_db'
HOUSE_PRICE_TIME = ['2014/', '2015/', '2016/', '2017/']
HOUSE_PRICE_INFO = {
    'datemonth': 'month',
    'dateyear': 'year',
    'propertycategorisation': 'type',
    'sa42016code': 'sa4_code',
    'sa42016name': 'position',
    'for_rent_home_lease_averageprice': 'rent_avg',
    'for_rent_home_lease_maximumprice': 'rent_max',
    'for_rent_home_lease_medianprice': 'rent_mid',
    'for_rent_home_lease_minimumprice': 'rent_min',
    'for_sale_both_auction_private_treaty_averageprice': 'sale_avg',
    'for_sale_both_auction_private_treaty_maximumprice': 'sale_max',
    'for_sale_both_auction_private_treaty_medianprice': 'sale_mid',
    'for_sale_both_auction_private_treaty_minimumprice': 'sale_min'
    }
```

Figure 1.3.3 Config File of Selected Names of AURIN CSV File

Data: Data part is used to store the download AURIN data csv files, there are 4 types of these files, “Employment”, “House Price”, “Income” and “Population” data, in the early design, there are also education data and RAI data ready to be applied, however, finally only important and the data that can be suitably sociated with our Twitter data are chosen.

Helper: Helper part contains a path helper and a database helper, the first one can be called when the AURIN handler needs to get the root path of this module. The second helper is used to save processed data into the database.

With these 3 parts, the main function can run one time to read data from the Data part and process it by using the data-info constants, then send it to the database helper to save the files. Then the script is over.

1.3.3 Database System

We use a 3-node CouchDB cluster as our database system. The database system has three roles:

1. **Store data from Twitter Crawlers**: CouchDB allows data to be stored in JSON format and issues rich queries against the data.
2. **Remove duplication**: CouchDB can remove duplicated data based on the document ID of

each record. For this project, we use tweet ID as a unique ID to remove duplication.

3. **Handle back-end MapReduce requests and create views**: CouchDB provides HTTP APIs for creating and accessing views. The back-end application can use HTTP queries to create different views to meet the needs of different scenarios.

Database MapReduce

MapReduce is a processing technique for distributed computing. A MapReduce program is composed of a map method, which performs filtering and sorting, and a reduce method, which performs a summary operation. In CouchDB, a functionality View is used to execute MapReduce operation. By creating views, we can perform statistical calculations on the data according to different scenarios.

Pros and Cons of CouchDB

Pros:

- The **Replication** function of CouchDB is very convenient to use. By defining source database and target database, one can easily implement data replication. There are two modes of replication to choose from, one time or continuous.
- CouchDB can **remove duplicates** automatically based on a unique ID defined by the user.
- CouchDB provides a web-based interface **Fauxton**. It's convenient to use since it provides a basic interface to the majority of the functionality, including the ability to create, update, delete and view documents and design documents. It provides access to the configuration parameters, and an interface for initiating replication.

Cons:

- **Consistency** between nodes is not very well maintained.

1.3.4 Back-end System

As it is mentioned above, the backend will provide RESTful APIs for the frontend to obtain the data of visualization. For a quick developing application, our team choose to use Flask framework which is famous for "micro", it keeps simple core functions for client-side applications to access and replies very fast and runs stably. This is dependent on the that our backend is used for communicating with the database and doing some data packing, and the main calculation jobs are embedded in the CouchDB with its strong map-reduce view processing ability. Additionally, it can also perform as a web template render providing the views on the browser.

Config: Same as the Twitter Crawler module, we can modify some config of the database in the app_config.yaml file, at this file, the host IP addresses of the database cluster can be configured, and then the password and the username of the database can also be changed in the same place. Finally, the names of the Twitter databases can be easily altered to meet the different keyword of the data collections and prevent some errors occurs. Except the config part, there are different part of backend to make the functionality become true.

Util: util part provides useful fundamental tools for the whole backend, config_handler.py is a config file reader, it reads keys and values from the app_config.yaml and parses them to the python variables and saves them as a class. After it is called, it will create a config instance with get-property methods exposed, so that we can make use of the values written in the config files. Next is constants.py, which stores the maps aimed to transform the specific words in the URL to the names of map-reduce functions in the CouchDB, in addition, also the default values of group-level of the reduce functions are stored here. Also, we have a simple database balancer in the util part, it just simply gets the next database address in a way of Round-Robin, which will take advantage of our database cluster, offering the high availability.

Service: service part is a low-level module to interact with the CouchDB, it is also a class that expose methods with parameters for the controllers (which will be introduced later), for initiation, this class calls the config part to get some database information, including a request string for accessing the CouchDB design documents, which provides all the map-reduce calculating results known by "views" in the CouchDB, thanks for the B+ tree structure stored in the database, our backend is free of dealing with the complex figures, sums or counting with different groups.

Controller: if the service part is exchanging information between databases and backends, the controller is designed to handle the request from the client-side or called the frontend in our system. Besides, the APIs provided are designed as RESTful styles, and of course, our frontend is only used for obtaining data and display, so only the GET API methods are offered. RESTful APIs are clear and easy to get what the client needs, in our 4 different scenarios the API will simply be named as "languages" (and "languages-month" for details data), "house-price", "covid" and "stream". In the developing phase, the backend also provides 2 other useful APIs type named “twitter” for getting twitter data and “aurin” for getting AURIN data, so the API formats are like:

- http://<host>:<port>/api/twitter/<type>/<view> (for debugging and developing phase)
- http://<host>:<port>/api/aurin/<type>/<view> (for debugging and developing phase)
- http://<host>:<port>/api/scenario/<type> (for demo phase)

As introduced above, the first two APIs will give clients original data from the map-reduce of the CouchDB directly, it is basically a JSON format data produced by the database and shown to the developer so the developer can do some preparations for data visualization. Moreover, these APIs that are possibly useful in the future are preserved and still open in the demo phase in case for need. As for the third type of API, it is used for demo and the backend will collect different data from different databases, then process, name and organize them and send it to the frontend so that the frontend makes easy use of it and save more time.

Design-doc-upload: The final part is a simple script used for uploading all the map-reduce functions to the database, for each database, the functionality of map-reduce will be broken without these design documents. As a result, we write these documents as a string, use JSON encoding and send them to the CouchDB. Similar to the AURIN saving scripts, it only needs to run one time before starting the backend service.

1.3.5 Front-end System

Front-end system uses React.js, TypeScript, Material UI, ANTV to build data visualization system. Users can specify the analysis scenario they want, set filter criterias like including which sub-dataset, which time period they want to see, then they can get the corresponding data visualization result.

React.js: React.js is a frontend framework that helps users to manage the state of their application conveniently. With the help of React, the application is more manageable, users can only focus on the business logic codes of each state, and leave the job like how to transfer application’s state without side-effect to React.

TypeScript: Typescript can be seen as an enhancement to JavaScript. It adds strong type checking and annotation to JavaScript, making Javascript much easier to read and maintain. In our project, data are usually very complicated, hence it is a must for use to integrated typescript to imply the future developers the information of the data and check whether things like input and output are accord to the rule. On the other hand, as a new technique, typescript may not supported perfectly by some old libraries.

ANTV: ANTV is the data visualization library that we used in this project. It provides almost all types of data visualization panels and gives lots of customized options. In this project specifically, we used the area chart, heat map, line chart, bar chart, pie chart to visualize our data.

Material UI: Material UI is the most popular React component library, which is maintaining by Google. It provides most basic components to set up a web page like modal, button, slider. Users can implemented the majority of their application in a short time and put more attention to the business part.

2 System Usage

While most parts of our system can be deployed in automation with Ansible scripts, our system also has modular functionalities in command. This section introduces the system usage in detail.

To deploy and use the system on a remote host, the users need the following prerequisites:

1. Installation of Ansible to run the deploying scripts.
2. OpenStack RC file and the OpenStack API password, both generated on MRC, to access and manage MRC resources remotely.
3. The private key specified for SSH to the instance created.
4. A text editor to modify the config files.

2.1 Instance Deployment

2.1.1 Deployment

The MRC instances are fully deployed with Ansible scripts. We use the built-in Ansible OpenStack module for configuration management. The OpenStack RC file (provided by MRC) defines environment variables to access our assigned MRC resources. Configurations of the instance are defined in the `nectar.yaml` file. Ansible is idempotent, which means with one set of configurations, the deployment script only needs to run once, since repeatable execution would have the same effect.

2.1.2 Usage

1. Define configurations for the cloud resources in the `nectar.yaml` file.
2. Run `create-instances.sh` script to create volume, security groups, and instances.
3. During script running, enter the OpenStack API password for OpenStack API authentication
4. During script running, enter localhost's root user password to grant privileges for installing Ansible's dependencies.

2.2 Database and Backend Deployment

2.2.1 Deployment

We use the CouchDB cluster as our database system. Each one of the backend nodes has a CouchDB node running on it. First node of the generated backend nodes in the `host_file.ini` is the masternode. The three CouchDB nodes of the cluster run in docker containers. Therefore, we need to install docker and mount volume to the CouchDB container. We use the latest ` ibmcom/couchdb3` as our CouchDB image. The CouchDB configurations are defined in the `backend.yaml` file. Since the database deals with big data, we mount volume to path `/data`. In addition, Default ports required by the CouchDB documentation (https://docs.couchdb.org/en/latest/setup/cluster.html) are open for the containers. Lastly, we use Ansible built-in uri module to send http requests to the masternode to finish the cluster setup.

Backend applications are also dockerized. We run the backend docker container with our own docker image. The Dockerfile to build the docker image are provided in the `backend` folder.

All the source code is managed in a git repository. We use Ansible script to checkout the code in the

cloud instance.

2.2.2 Usage

1. Define `all`, `masternode`, and `othernode` in the `host_file.ini` file based on the instance ip generation in part 2.1 instance deployment.
2. Edit DB configurations in the `backend.yaml` file.
3. Run the script `prepare-instance.yaml` to install dependencies, mount volumes, and pull Git repository.
4. Run the script `deploy_backend.yaml` to run CouchDB docker containers, create the CouchDB cluster, and run backend application Docker containers.

2.3 Frontend Deployment

2.3.1 Deployment

Frontend application is built with `npm run build` command , through Ansible built-in shell module. Then the build website is deployed on a dockerized Nginx server. We use Ansible built-in template module to replace the environment variable and generate the `nginx.conf` file, where reverse proxy and load balancing are defined. What's more, npm needs to upgrade with command `n latest` to make sure the build runs. So we also provide Ansible shell commands to do it.

2.3.2 Usage

1. Edit the `frontend.yaml`, `nginx.conf.j2`, `Dockerfile.j2` files to modify configurations needed.
2. Run the script `deploy_frontend.sh` to install the dependencies and run dockerized Nginx server.

2.4 Crawler Deployment

2.4.1 Deployment

Twitter Crawler Application can be run directly with python in the terminal, it can also run in the Docker container. Stream Crawler needs continuous running, because it will stream the latest tweets in CouchDB for data analyses. Search Crawler for COVID can run continuously, because there is a large amount of tweet IDs that is needed to query. Historical crawler runs by actual demands, because the number of tweets within 2014-2017 is limited and can finish query work in 1 hour.

According to the system architecture, we have three backend nodes. The three types of crawler are deployed separately on these three server nodes to balance load of servers.

2.4.2 Usage

Run crawler without Docker:

1. Install Dependencies with `pip install -r requirements.txt`;
2. Edit configuration files in `/config` to define the crawler behavior;
3. Prepare data files that are needed, such as tweet ID files and the historical tweet file, then save the path in the configuration files.
4. Run the `main_app.py` script with `-s` argument to declare which scenario to run.

Run crawler with Docker:

1. Edit configuration files in `/config` to define the crawler behavior;
2. Prepare data files that are needed and save the file path to configuration files;
3. Pick one `.sh` file to run based on the scenario needed. For example, if you would like to run Stream Crawler, run `run_stream_crawler.sh` to start the application with Docker.

2.5 AURIN Handler Usage

AURIN handler is a run-once python script and can be executed by the terminal or shell scripts. Before running it, run `pip install CouchDB` first. There are serval ways to make it work. Firstly, it can run in the local environment if the user changes the database host IP address and corresponding username and password. Secondly, it can also be executed locally in the local database without any change. Then the user can use the replication function provided by the CouchDB to send the data to where the user needs it. The last one is default usage for our system, in which the script can be executed in the MRC servers only once. Our team think scripts do not need to use a docker container due to being a bit wasting the storage.

2.6 Back-end Application Development

2.6.1 Design

As discussed in Section 1.3.4, the backend application is based on the Flask framework with serval functional parts, the basic idea in the backend is listening to requests, and checking for the routes, once matches that, the corresponding method will be called.

In each method, a connection with one database will be established by using our simple database balancer module at the beginning, and then it calls the different methods of database.py and waits for the response. In the database.py, a string interpolation design is applied, so the parameters will become a part of the request link for the CouchDB API, for instance, the names of the views in the design documentation, and which group level should be selected. Then it sends the request and receives data texts, and it needs to be parsed as a python object, returning to the controller methods. Then the API method may need to pick these data up and rearrange them, then produce a new data object. The API methods of the scenario controller (used for handling the demo APIs) may ask for the database 4 or 5 times for a data combination in one single scenario, however, the data eventually sent to the client-side (frontend) is all in one JSON format.

2.6.2 Local Test

1. Adjust the app_config.yaml to set correct database addresses, username and password.
2. Run `npm install -r requirement.txt` to install the dependencies.
3. Run `python upload_design_doc.py` to upload design documents.
4. Run `python app.py` or `sh ./docker-run.sh` (using docker).
5. Check the browser “http://localhost:5000” to see a “hello world”.
6. Send HTTP Get methods with our API formats to get data.

2.7 Front-end Application Development

2.7.1 Design

Based on the design, frontend will display the corresponding data visualization panel based on the selected scenario and criterias, hence those are actually the global state of the whole system and is managed through context api provided by React.js. When the selected scenario changes, because each

panel uses different data, front-end needs to re-fetch data using the new api and generate the corresponding data visualization panel. However, if only the filter criterias change, because all data is stored locally when loading an scenario's data visualization panel, there is no need to re-fetch the data, it will only filter data based on the new criteria then update the panel, which can significantly reduce network loading and make the application more efficient.

2.7.2 Local Test

1. `Cd` into the frontend folder.
2. Run npm install to install all dependencies of the frontend application.
3. Run npm run start to start the frontend application.

3 System Functionality

3.1 Scenarios

In our system, we mainly focus on the Greater Melbourne, which is the area within the Urban Growth Boundary to help locate the AURIN data with Statistical Areas Level 4 (SA4) Standard and associated with the Twitter data that has coordinates. In such a way, we can aggregate and then analyze a specific area. Our team wants to explore 4 scenarios on the project, they are "tweets languages and overseas migration analysis", "house price and attitudes", "policy and decisions made by local government and people feeling during Covid" and "real-time feelings on the social platform".

3.1.1 Scenario tweets languages and overseas migration analysis

a. Description

In this scenario, our team extract different tweets from our databases storing Twitter data and counts the number of different languages in these tweets. On the other hand, we also have the AURIN data about net overseas migration, total migration, and total population in Melbourne. After an easy calculation, we will see a rough proportion and can compare it with the English tweets' scale.

The reason why our team choose it is due to some considerations, first, if a city can attract more foreigners deciding to live, strong evidence will be thrown out that the city is comfortable to live in. Secondly, people usually have a sense of belonging when they are used to the local life, the culture, and the food, thus they become one of the residents. So do the languages they use, once the overseas people will use local languages to share lives on the social platform and make new friends, we can say that the livability of a city can be confirmed because it can attract people all over the world.

b. Visualization

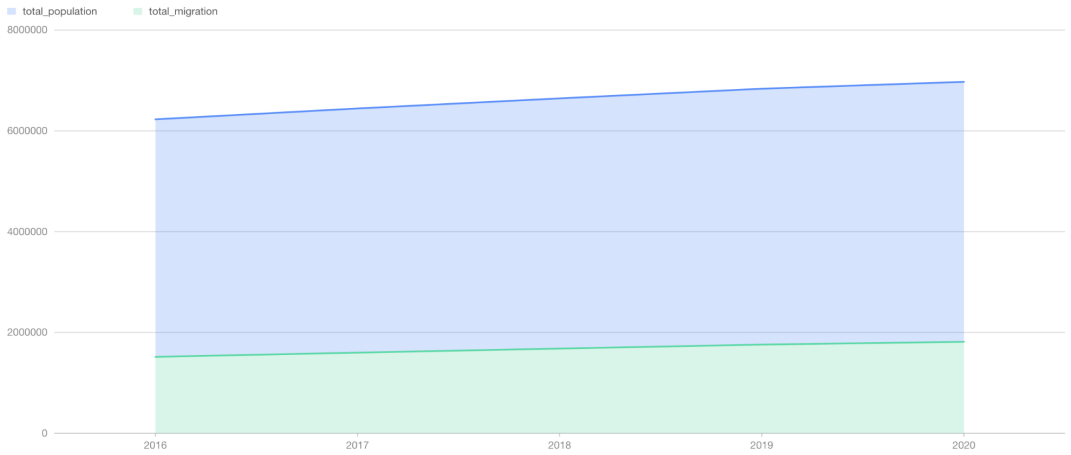

Figure 3.1.1.1 Overseas Migration Number and Total Population Number

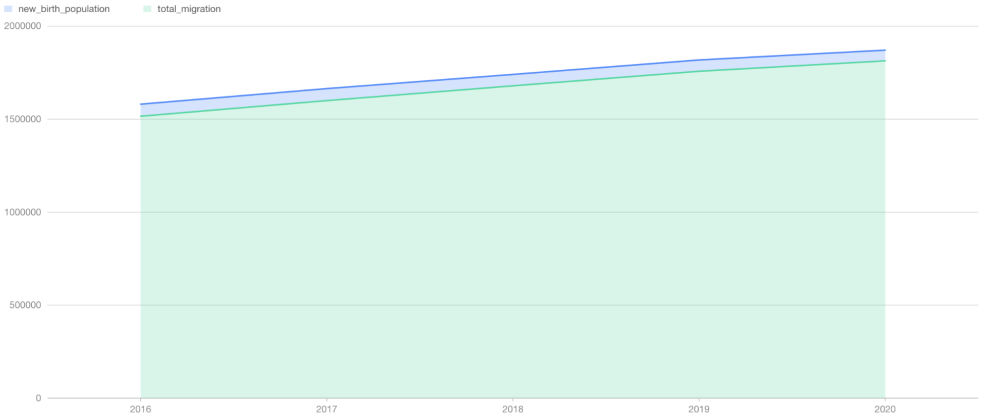

Figure 3.1.1.2 New Overseas Migration Number and New Birth Population Number

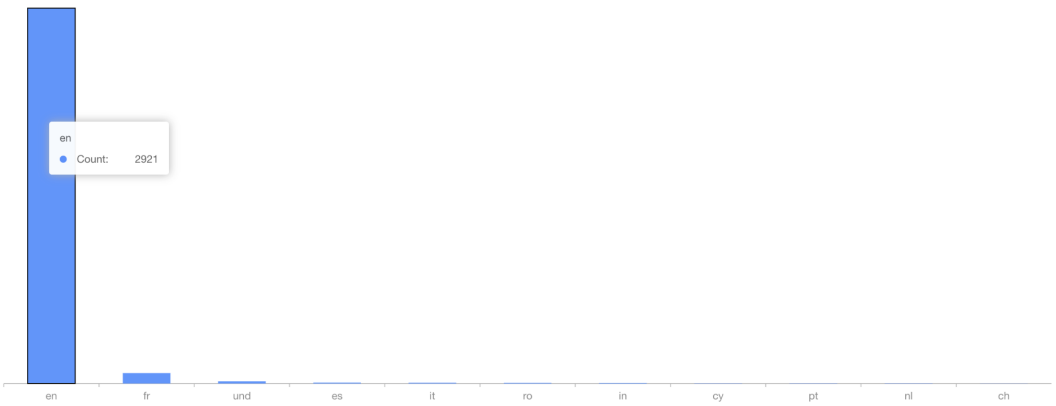

Figure 3.1.1.3 Different Languages Usage From 2015 To 2021

3.1.2 House price and attitudes

a. Description

In this scenario, our team initial design to combinates the historical twitter data, house price information (including units and houses), and also an income information in AURIN to give people's attitudes about these prices in different areas (SA4) in Melbourne. Unfotunately, for some reason, these data are not displayed at final stage, but our team will still show our data processing ability with the geometric coordinates data (here we use AURIN house price data), and set it on the map.

The reason is simple, the guarantee of living conditions is an important criterion for evaluating the quality of life. Thus, the house price and income level in a city will decide how difficult people can find their favorite house. As a result, our team can observe the attitudes toward house prices on the Twitter platform and explore the relevance of attitudes, incomes, and house prices in that area.

b. Visualization

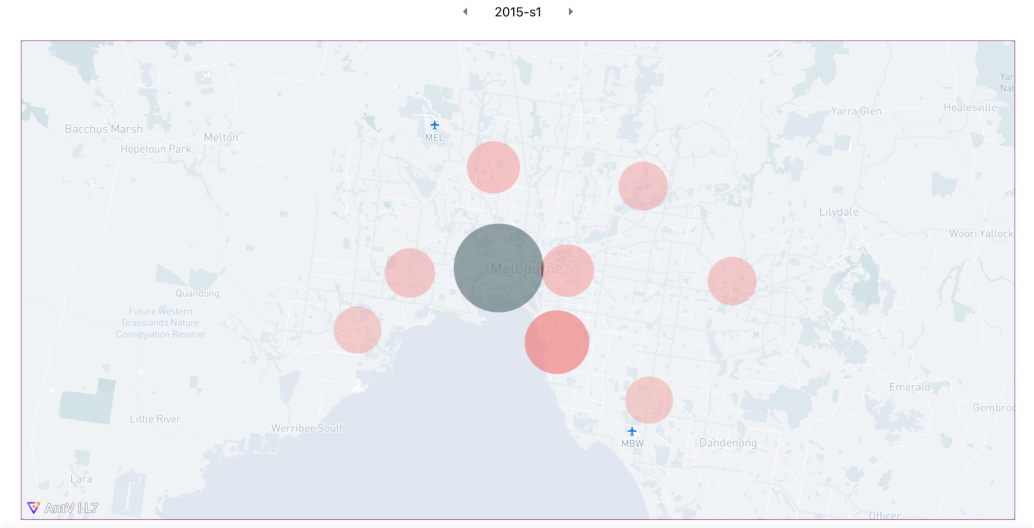

Figure 3.1.2.1 An Example of Displaying 2015-S1 Unit Type Max Price in SA4 Standard

3.1.3 Policy made by local government and people feelings during Covid

a. Description

In this scenario, our team wants to observe the how Melbourne and local government deal with emergency situation, Covid-19. We collect covid tweets about Melbourne and calculate their average values during year 2020 by month and give a detail of the special "lockdown" period from about July to September. To illustrate why people will have some emotional volatility, we will use the employment, unemployment data from 2019 to 2021 by quarter to seek the answer.

In general, the livability of a city is inseparably linked to local government. The decisions it makes, the policy it releases will have a huge impact on the lives of people. Therefore, our team wants to see how the decision of lockdown will influence the city, and whether the government and people is prepared well for this. If the policy is effective and positive, so the corresponding data will tell us the result.

b. Visualization

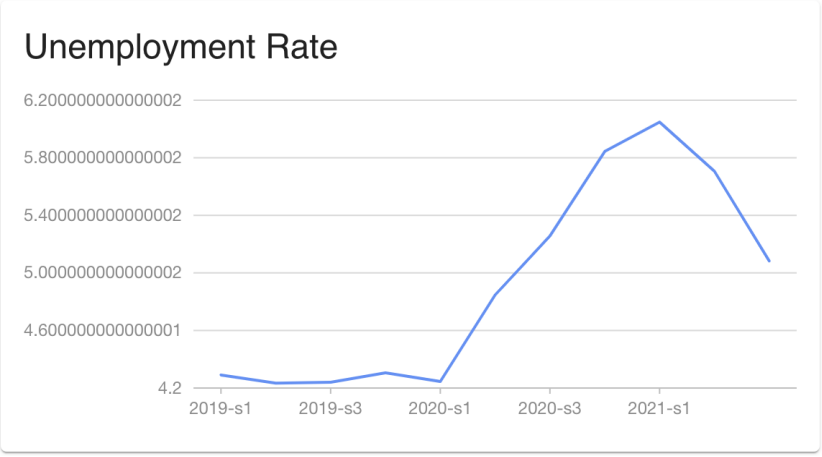

Figure 3.1.3.1 Unemployment Rate From Year 2019 to 2021 By Quarter

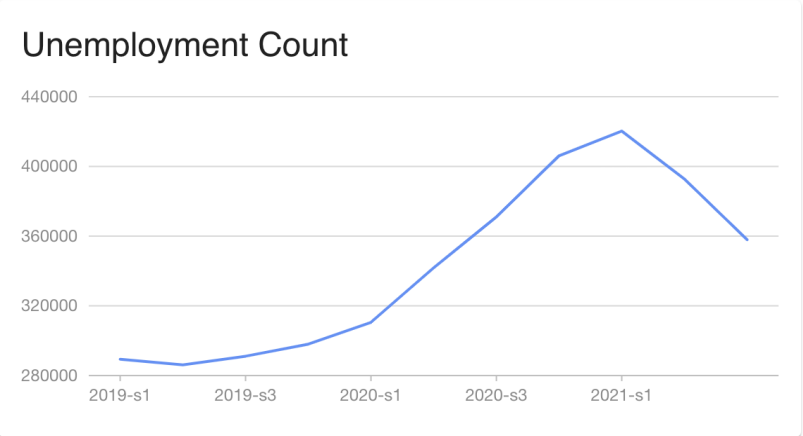

Figure 3.1.3.2 Unemployment Count Number From Year 2019 to 2021 By Quarter

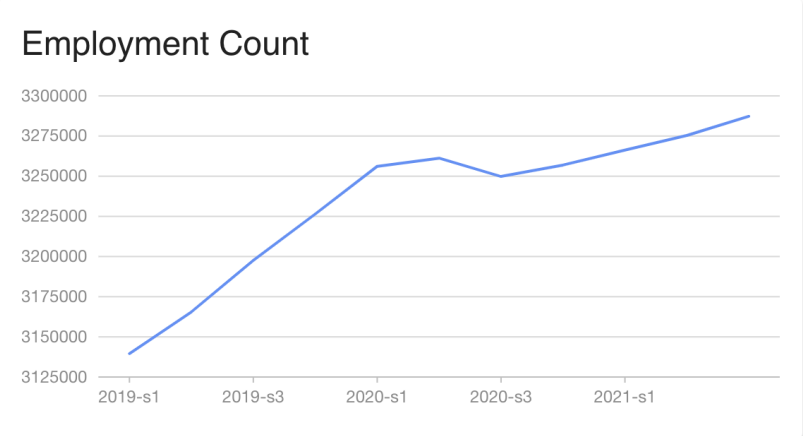

Figure 3.1.3.3 Employment Count Number From Year 2019 to 2021 By Quarter

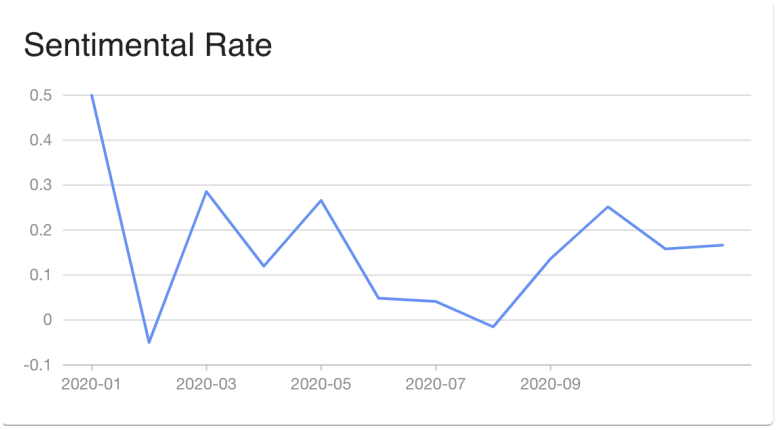

Figure 3.1.3.4 Twitter Sentiment Analysis During Year 2020 By Month

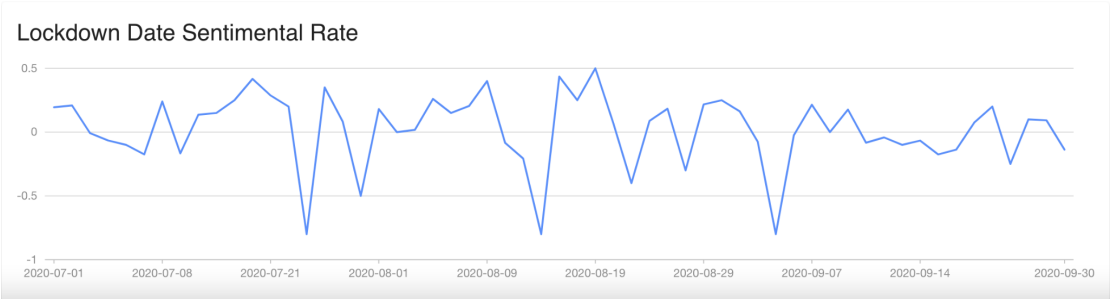

Figure 3.1.3.5 Twitter Sentiment Analysis From July to September

3.1.4 Real-time feelings on the social platform

a. Description

In the final scenario, our team will show our ability of collecting the live tweets about Melbourne and give how people feels about Melbourne by different keywords which is set on the Stream Twitter Crawler. In the display phase, we will show the basic total attitudes and subjectivities of people, and each is distributed in 5 level from "very positive or subjective" to "very negative or objective", in addition, we will also show the 10 latest sample tweets on our page with detail information like tweet text, post time, languages and post source.

For the final scenario, we want to explore the attitudes of people via Twitter platform directly, it is intuitive to show the words of people live in or talk about a city, the more positive of these tweets from the people, the stronger the evidence in support of Melbourne's livability. In demo, we are simply use the keyword "food" for display, but the users can use any words to explore if they want.

b. Visualization

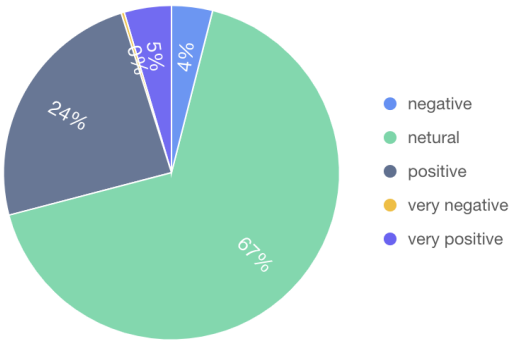

Figure 3.1.4.1 General Stream Twitter Sentiment Analysis Information (Polarity)

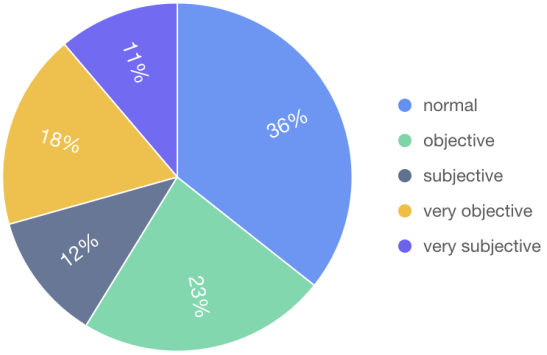

Figure 3.1.4.2 General Stream Twitter Sentiment Analysis Information (Subjectivity)

Figure 3.1.4.3 Sample Tweets and Information of It

4 Discussion

4.1 Error Handling

As for Twitter Crawler's Error Handling, the most common error is the Network error.

Twitter API is not always available due to server maintenance, in that case all responses from the Twitter API will be 503 that may cause failure on Twitter Crawler application. What's more, we are also inspired by the concept of checked exceptions in Java, in other words, we believe that the network exception can not be handled by our codes. As a result, we used the "try … catch …." pattern in every network related logic to catch exceptions and handle them graciously.

The other challenge is that there are three different sources of tweets. If we use a unified pre-process handler, it is possible to crash because of missing fields or mistaken keys. Luckly, there are only three sources of tweets, and differences between them are not obvious, except for several missing fields. In this case, we still use a single unified pre-process handler with many "if … else …" to detect missing fields.
However, for scalability in the future, we believe that it is good to apply a strategy pattern for this business rule. We could define an abstract tweet preprocess handler class, then implement several concrete tweet preprocess helpers.

In the Backend, the error mainly occurs in the bad request for 404 error, since the request link may not be correct in the testing phase. Therefore this handler is used to tell the user there are some spelling mistakes in the request so that the backend cannot find a matching API, in a more friendly way.

In our "stream" API design, the method will search for recently 10 updated documents as a sample data and return, sometimes we make changes on the views, as we may add new map-reduce functions or want to delete useless views, then the method will fetch that modified design document and cause the format mistakes, in this case, we just catch the KeyError the skip the document, so that we can correctly deliver 9 tweets samples. Nevertheless, the error only arises in the developing phase, and there nothing to worry in demo.

4.2 Availability

For the Database system, we are using the CouchDB Cluster, which provides Replication model and Sharding model to ensure Availability and Scalability. The increasing number of server nodes could increase availability of the database because if one node is down, there will be data replications in the others. Also, CouchDB clusters can be read from each node because the data will be in the final consistency, which provides availability on the database reading. Although there will be latency in the node synchronization, our business rules are not required to have a high consistency of data.

For the Backend system, we deployed the backend application in 3 server nodes, and there is a reverse agent between frontend application and backend nodes. If we configure the reverse agent properly, it can help us forward the request to the backend node that is alive.

However, we only have one node for the reverse agent, which means if the agent node is down, the frontend will lose contact with all backend nodes. It is better to build up a reverse agent cluster as well to guarantee availability of the backend communication.

To sum up, availability is ensured in the database system and backend system upon most occasions. On the other hand, availability of the frontend application is not guaranteed. Single node deployment for the frontend is because we want to make server nodes have a clear division of labor, and also, deploying a frontend module with backend in the same server node may bring risk to the backend system and data since the frontend is usually accessed by public.

4.3 Scalability

For the Database system, CouchDB Cluster supports adding nodes via a GET query with configurations about number of shards and replicas. CouchDB will use incremental replication to ensure the final consistency of data.

For the backend system, each node will run a replication of the backend application. Based on the system design, the backend application will be idempotent and stateless. The frontend only performs get queries to backend, and all data needed for data analysis are stored in the database cluster. As a result, extending the backend application will be easy, which means that we can deploy more backends to new nodes without any modification.

For the crawler application, it can improve the multi-core performance on one server node by increase the worker thread number in the configuration file, or modify `xxx_thread_num` parameter in the `main_app.py`. For multi-node performance, users can let different nodes search different keywords with Stream Crawler and Historical Crawler. For COVID Search Crawler, users can let different nodes read IDs from different months or days to increase performance of ID looking up. However, we need all server nodes to have the same historical tweet file and COVID tweet ID files.

For the frontend system, there is only one server node. We believe that the front end can also have scalability. For example, we can deploy multiple frontend servers and provide a gateway server for the public to visit. As for the reverse agent, Nginx also supports cluster deployment.

To sum up, the scalability of the whole system is guaranteed.

4.4 Containerization

To deploy the system in the same environment and make it easy to migrate and achieve isolation, we are using virtualization technology in the project. There are several virtualization technologies such as Hyper-V, KVM and so on. Virtual machines can provide the application with a unified runtime environment and make it isolated from potential inference, however, virtual machines are “fat” in terms of system requirements, which is not acceptable in limited server resources. Also configure virtual machines can be troublesome, because we need to install Operating system and runtime dependencies on each virtual machine..

As an alternative, we use Containerization technologies to achieve virtualization. Containerization is efficient because it uses a shared operating system and can be tiny and neat to achieve our requirements for this project. containerization is also easy to operate on. While building a container, the application can be easily added to the container and expose ports to the external world for service.

We choose Docker as the solution of containerization in this project. As a popular containerization solution, we can easily find documents and references online, which make the development process less technically risky. What’s more, Docker enables developers to pack the application easily as a lightweight container, which precisely meets the project requirements.

As a result, we use Docker to pack applications for the project because it is easy to use and efficient in server resources, and if we meet any difficulties, the document and references are also available online. Using Docker brings advantages for system scalability and isolation.

5 Individual Roles and Contribution

Xiaotian Li 1141181:

- System Architecture;
- Twitter Crawler Development;
- Provides tweet data in CouchDB;
- Report: System architecture, Crawler software architecture, Crawler usage, Discussion.
- Team motivation, decision making and project schedule control.

Rui Zhang 1294310:

- System deployment(Ansible scripts);
- CouchDB setup;
- Nginx setup;
- Report: MRC Deployment detail, Deployment in System Usage.

Bocan Yang 1152078:

- Collecting and saving AURIN data (with python scripts);
- Part of map-reduce functions;
- RESTful API design and Backend development;
- Report: AURIN and backend system design, usages of AURIN scripts and backend test, Scenarios, Abstract.

WeiFeng Wu 1232129:

- Front-end UI design
- Front-end development
- Solve front-end network problems like cors
- Report: Frontend system design, frontend application development, frontend usage

Huahu Wen 1215209:

- Usage of CouchDB functionalities;
- Participate in scenarios design and design documents writing;
- Report: Database System

6 Appendix

6.1 GitHub Repository link

https://github.com/Blackmesa-Canteen/COMP90024-Cluster-and-Cloud-Computing-Assignment-2

6.2 Frontend Link

http://172.26.135.17:80

6.2 Video Links

6.2.1 System Deployment and Configuration

https://www.youtube.com/watch?v=gCR_GMbe9tI&list=PLsPXub5kWbqojCqGz-1Zgdk4m6YC821l-&index=5

6.2.3 Database Utilization

https://www.youtube.com/watch?v=1bhOiZqTO1M&list=PLsPXub5kWbqojCqGz-1Zgdk4m6YC821l-&index=3

6.2.4 Crawler Usage

Part1:
https://www.youtube.com/watch?v=6e5Q-WBf8KI&list=PLsPXub5kWbqojCqGz-1Zgdk4m6YC821l-&index=1

Part2:
https://www.youtube.com/watch?v=9vwR2HPDHtY&list=PLsPXub5kWbqojCqGz-1Zgdk4m6YC821l-&index=2

6.2.5 Back-end Usage and Test

https://www.youtube.com/watch?v=qxpLwm6HOCI&list=PLsPXub5kWbqojCqGz-1Zgdk4m6YC821l-&index=4

6.2.6 Front-end and Live Demo

https://www.youtube.com/watch?v=VGyziLEugSA&list=PLsPXub5kWbqojCqGz-1Zgdk4m6YC821l-&index=6