# Crawler Side

## 1 Description

The crawler part in this project is in charge of obtaining data from various data sources—to be specific—Twitter and AURIN APIs. In order to support the research topic sound and well, the crawler itself should be able to retrieve data within a certain range of specified keywords and location coordinates, which guarantee a higher effectiveness of data usage and sentimental analysis afterwards. In addition, it is critical to accumulate a large amount of raw data, which lessens or eliminates the possibilities towards wrong conclusions during analysing process and contributes the final goal of this project.

## 2 Crawling Tweets

Tweeter is researcher-friendly and provides a number of APIs for retrieving tweets from its database. But still there are lots of problems and challenges having been met during the crawler’s development.

### 2.1 Requirements Analysis

### 2.2 Challenges & Solutions

#### 2.2.1 Outdated Documents

Although the Twitter gives out a range of APIs to researchers, the Documents of them, largely, do not contain the newest features about. Thus for the usage of Twitter APIs in the crawler side in the early stage, only the most basic and simple functions are put into, which only provides poor efficiency in both data retrieving and possibility of analysing afterwards.

To solve the challenge above, a great range of projects on GitHub and information in various forum have been viewed, for extra information of Twitter APIs new features. And finally, one of the API: *tweepy.api.cursor()* has been found, which covers the most of functions of previous attempts, enhancing the fetching performance to a great extent at the same time.

#### 2.2.2 Limitation of Data Requests

To constraint the total data throughputs, Twitter limits the rate of requests invoked by API users—180 times per 15 minutes (1 request every 5s) by default, which makes the process of accumulating tweets problematic.

Solution to this problem should lie in not only breaking the limitation of requests rate, but also improving the percentage of valid tweets retrieved in each reply from Twitter.

Possible solutions to these issues are:

* 1. Get tweets as many as possible within one request invoked in crawler and sent to Twitter.
  2. Describe the searching request to Twitter as explicit as possible, to ensure the majority of tweets retrieved in each request are valid and ready to be used.
  3. Register more Twitter authentication keys that allows sending tweets requests simultaneously to Twitter server, which would multiply the efficiency of data retrieving process.

In this project, altogether 3 Twitter Application keys are applied and used, and with the help of *tweepy.api.cursor()*, which provides sufficient arguments for describing each query request in detail and will package up to 500 tweets in each reply, in total the tweets crawler would output a throughput nearly 100 valid tweets per second on average.

The theoretical final throughput should be 1 req / 5s \* 500 tweets per req \*3 keys = 300 tweets/s, the duplicated tweets and data constraints lessen this number to just below 100, but still a great improvement when compared to 1- key crawling.

#### 2.2.3 Duplicated tweets

Duplicated tweets, apart from distinct tweets, would do harm to the sentimental analysis afterwards. To be specific, it would amplify the positive or negative emotions to the analysing process, which affects the final threshold of judgement. Therefore, there is a necessity for the removal of duplicated tweets.

Situation of duplication in tweets is common, especially when fewer constraints are defined while querying with Twitter APIs. On analysing the replies from Twitter APIs after each request, it is obvious to point out that there are two main types of duplication:

1. Duplication in both user *screenname* and message *text*
2. Duplication only in message *text*

For the cases above, (a) is regarded as the duplicated tweets that would do harm to the whole system and must be removed. To filter the tweets that contains same *screen\_name* and message *text*, a new *identifier* = *screen\_name + text* is defined for each tweet, which helps to fix the issue above. On defining this *identifier*, only the tweets with same *screen\_name* and *text* can be filtered out, but not the ones with same *screen\_name* only or *text* only.

At this stage, it is not a sound filter method, but a primary one, to give the first treatment to duplicated tweets, and some more complicated filter methods is implemented in the database and analysis sides, which are relevant to context and sentimental level. The meaning of the primary filter is to leverage the analysis afterwards, because the duplication here is syntax-relevant and easy to handle, which saves calculation resources for following parts for other analysing needs.

#### 2.2.4 Search Exactly

In the beginning stage, the search could only base on the keyword—all the returned results are only based on the given keyword. With this searching strategy, there is no problem in the number of returned tweets, but the percentage of tweets’ validity—only a small range of tweets contains the coordinate, within which only a tiny group of tweets are inside Australia. Although it can provide tens of thousands tweets every half of a day, but the efficiency of each request is too low to accept, which is a waste of resources and give a large pressure to the database and relevant parts.

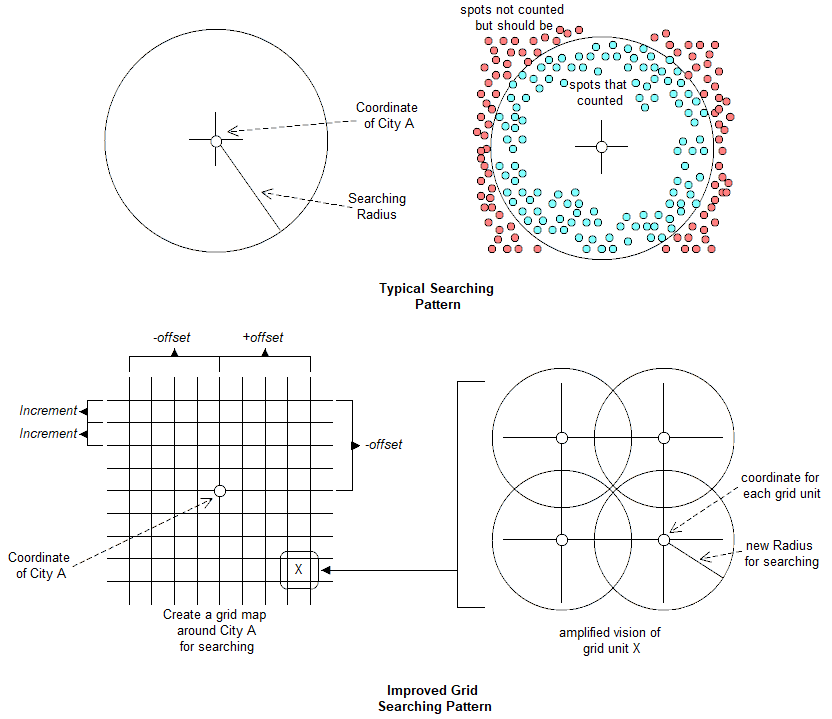
To solve this, *geocode* is found to be helpful, which defines an area by giving the coordinate of the centre of a circle—(latitude, longitude), and the radius of the searching circle, in *km* or *mi*. After defining this parameter for each request, the replies of Twitter would give tweets that contain keyword and are in exact circle of areas.

But by applying this, other challenges arouse here:

1. For the most of the tweets that returned by Twitter, they do not even contain the information of location, or exact coordinates, which is significantly unfavourable to the following sentimental analysis and statistic work.
2. For the tweets accumulation process, the greater the radius of searching is set, the more tweets would be returned within this area, whereas the more vagueness in location are given to these tweets at the same time.

Therefore, the searching strategy should be modified—instead of searching only based on the given centre coordinates and searching radius for each main city in Australia, another searching pattern is adopted here—creating a grid of points which could cover the adjacent areas around the specific city, and for each point, we can deploy a typical circle search. To describe the Grid exactly, two parameters—*offset* and *increment* are defined. *offset* confines the total area of the Grid map, and the distance of each point in the grid is assigned by parameter *increment*.

The Typical Searching and Grid Searching processes can be illustrated by following figure:



*Figure 1: Comparison between Typical Searching and Grid Searching.*

From the comparison of two searching patterns above, it is obvious that:

1. The Typical Searching Pattern searches in a round area, which would not meet the data retrieving needs in the corner areas of the city (if required). Whereas the Grid Searching Pattern would cover this drawback.
2. For each small searching circle in Grid Searching Pattern, it is so small and exact enough (below 10km, typically 5 – 8 km for searching radius) that could be regarded as the exact coordinate information for each returned tweet in this area (if this tweet does not contain origin location information from Twitter), which contributes to the accuracy of data retrieval process with AURIN Database afterwards.
3. The Grid Searching Pattern generates lots of overlapped areas in each searching rectangular, which would cause duplication of tweets. However, the mechanism of processing duplicated tweets in the previous section would handle this well.

Practically, based on the Grid Searching Pattern and after testing parameters, the offset is set at 0.4 and the increment is set at 0.1, which is proved to be ideal enough for this project’s data needs. That is—it would search in 81 ( ((2\*0.4/0.1)+1)^2 ) small circle of areas around each main city across Australia.

#### 2.2.5 Scalability

One of the core requirement of crawler side in this project is scalability. In specific situations, once there are new nodes put into the cluster, or requiring new instances for the tweeter crawler, all the instances of the Twitter crawler should work independently and correctly, without popping any collision issues.

The most possible issue here is the duplication of tweets—the same tweets would be replied in a group of crawler instances by Twitter, for it regards all the instances as different user requests for they have different keys and IPs.

To solve this challenge, one of the CouchDB’s feature is used—the attribute *\_id* for each item in a database should be assign uniquely. In case of the same user sending different message or different users sending same message, and these situations should not be regarded as duplications in tweets, a new method is taken into account for generating distinct *\_id* for each unique tweets—hash the combination of user name and his message text ( hash(usr\_name + usr\_text) ). By doing this, supposing that if two instances of crawler get the duplicated tweets, the *\_id* field in each instance for this tweet should be the same, and once one of the instance has finished its insertion to CouchDB, the other instance by no means could do it again—just simply rejected by CouchDB. Besides, the crawler should capture this remote exception invoked by CouchDB and simply discard current insertion and move to the next tweet.

### 2.3 Data Extraction from Raw Tweets

After receiving raw tweets from Twitter APIs, extracting necessary field of data in each tweet becomes next. To finish this part, following steps should be taken into consideration:

1. Acknowledging the type of data that needed: based on following sentimental analysis’ requirements, which can be tabled as following:

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Attribute** | **Type of Value** | **Description** |
| 1 | \_id | String | Required by CouchDB for indexing |
| 2 | name | String | Tweet’s user *screen\_name* field |
| 3 | usr\_id | String | Tweets’s user id |
| 4 | status count | String | Number of user’s total tweets counts |
| 5 | location | String | City name of current tweet located in |
| 6 | time zone | String | Time zone of current tweet |
| 7 | data time | String | Time when current tweet posted |
| 8 | retweeted | Boolean | If current tweet retweeted from others |
| 9 | keyword | String | By which keyword filtered this tweet |
| 10 | msg | String | Message field of current tweet |
| 11 | coordinate | String | Coordinate that current tweet located |

1. Once the extraction needs are clarified, next would be finding the exact fields that match these requirements. Some of the fields can be filled with invoking Twitter APIs (e.g. name = tweet.user.screen\_name, etc), whereas others only available through text analysis. Regular Expression is used for this certain analysis work. On checking the raw data of tweets, following regular expression are proved useful on finding each field of data:

|  |  |
| --- | --- |
| **Extract Target** | **Regular Expression** |
| usr\_id | id\_str=u\'([0-9]+)\' |
| status count | statuses\_count=([0-9]+) |
| time zone | time\_zone=u\'([a-zA-Z]+)\' |
| date time | datetime\.datetime\(([0-9, ]+)\) |
| retweeted | retweeted=True |
| coordinate | coordinates=\[\[\[(-?[0-9]+\.[0-9]+), (-?[0-9]+\.[0-9]+)\], \[-?[0-9]+\.[0-9]+, -?[0-9]+\.[0-9]+\], \[(-?[0-9]+\.[0-9]+), (-?[0-9]+\.[0-9]+)\], \[-?[0-9]+\.[0-9]+, -?[0-9]+\.[0-9]+\]\]\] |

Although the RE method in python does take time to analyse text, the bottleneck of this crawler instance is not restricted by RE matching, but the request rate limited by Twitter side. Therefor, the usage of RE method would not be problematic in performance.

1. Finally, the crawler should insert trimmed data to remote CouchDB.

It is convenient for python to communicate with CouchDB through the package “pycouchdb”. Once the necessary information of connection is given, including username. password, remote IP address and remote Port number, by using following code can the python program establish a remote server object locally:

*db\_server = pycouchdb.Server(ex\_db\_info, authmethod = "basic")*

where *ex\_db\_info* contains the authentication info mentioned above.

After connect to remote CouchDB server successfully, before inserting data, a database should be assigned, by using following instructions:

*db\_name = …*

*try:*

*rmt\_db = s.database(db\_name.lower())*

*except:*

*rmt\_db = s.create(db\_name.lower())*

*rmt\_db …*

And finally, when it comes to insertion data to remote CouchDB, *save* clause is used as following:

*rmt\_db.save(dict(attr1 = \_\_, attr2 = \_\_, …, attr n = \_\_ ))*

## 3 Crawling AURIN

Australian Urban Research Infrastructure Network (AURIN) could be regarded as a platform, which provides entries to various open databases from data providers across Australia. It allows researchers to retrieve data from its numerous databases through open APIs, after authenticating users’ access keys. This feature facilitates the most researches on AURIN Platform in this project to a great extent.

### 3.1 Requirements Analysis

In this project, data from AURIN play a role as a proof, that proves the conclusion derived from previous sentimental analysis, which based on the retrieved tweets from Twitter. Thus, the overall requirements for AURIN data retrieval could be concluded as following:

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

### 3.2 AURIN Open API

#### 3.2.1 Communicating with AURIN APIs

Instead of using the AURIN recommended tool—QGIS IDE, in this project we choose python program as the main part of data retrieval method with AURIN Platform, on concerning the following issues:

1. QGIS needs extra installation on the target server, which complicates the processes of automatic configuration and data retrieval—QGIS works as a mid-tier between users and AURIN Platform. Although it provides various kinds of visual demonstrations of each dataset and attribute, the functions like those at this stage are not required, especially in this project.
2. There are 2 options for filtering data in QGIS—one is to filter data manually, which obviously slow and subjective; the other is to using the integrated WFS function, sending wfs query requests to AURIN Platform remotely. None of the functions meets the data retrieval needs in accuracy and performance.
3. Querying with AURIN Platform is mainly based on WFS requests, to be specific, a HTTP type of requests. Python program (or script) could support this requirement well (urllib2) and is a rather light-weighted alternative when compared to the bigger QGIS IDE.
4. In this project, it is not only about crawling data from AURIN, but also primary trim and normalization are needed at this stage—it is unaffordable for the database server to hold all data that from AURIN datasets. Thus, by programming with python scripts, it gives a flexibility both in retrieval of data as well as the regulation of data.

#### 3.2.2 Authentication & Submitting Request to AURIN

AURIN Open API requires a username and a password for authentication, before receiving requests from users. The username and password used in this project, provided by AURIN Platform Admin Group, are shown as following:

*User name = research*

*Password = 9g6P7DkT*

Owing to the feature of WFS requests, each request sent should contain the authentication information. Following codes illustrate this process:

\* parameter *url* holds the WFS http request, which should be formed up previously.

*def AURIN ( url ):*

*password\_manager = urllib2.HTTPPasswordMgrWithDefaultRealm()*

*password\_manager.add\_password(None, url, username, password)*

*auth\_manager = urllib2.HTTPBasicAuthHandler(password\_manager)*

*opener = urllib2.build\_opener(auth\_manager)*

*urllib2.install\_opener(opener)*

*req = urllib2.Request(url)*

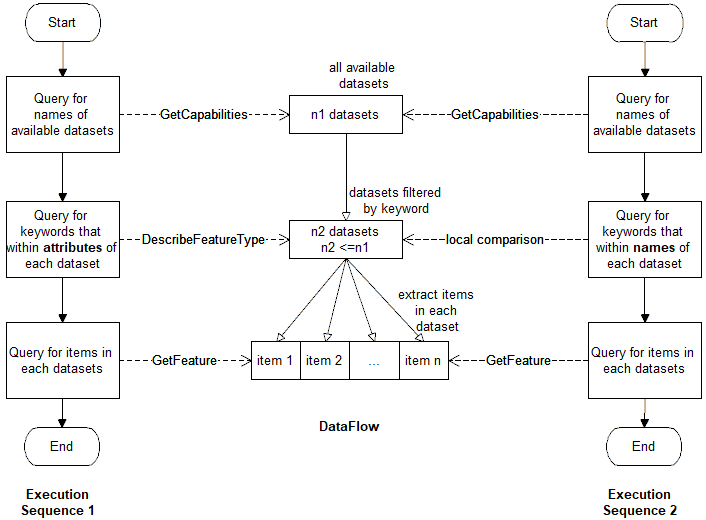
*handler = urllib2.urlopen(req)*

*return handler.read()*

This bunch of codes above shows how the request *url* is sent along with user authentication information, and the process of receiving the replies from AURIN.

#### 3.2.3 Different Crawling Patterns

In this project, 2 models of searching pattern is defined and tested, shown in following figure:



*Figure 2: Two attempts in datasets retrieval*

The differences between these two execution patterns could be concluded as following:

1. For Execution Sequence 1 (ES1), after get a names list of available datasets, it compares the keyword with each dataset’s attribute, which based on WFS request: *DescribeFeatureType*.
2. Whereas in the other Execution Sequence 2 (ES2), once it has established the list of datasets names, it only compares the keyword with that names list.

This slight difference results in total different outcomes, which makes these two models distinctive:

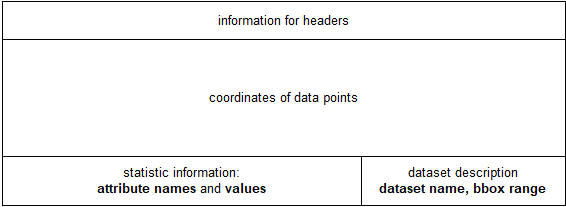
1. Execution time for ES1 is longer than ES2—the processes of querying for each attribute name in each dataset is super time consuming, each *DescribeFeatureType* request could only deal within one dataset each time. Thus if there are thousands of datasets available at a time (exactly 2379 datasets), the cost in time would be large.
2. Result datasets of ES1 is smaller than ES2—ES1 judges the appearance of the keyword in attributes of each dataset, but the fact is that, many datasets use abbreviations of words as attributes name, for example: income—inc, salary—slr, household—hshld, etc., whereas for the name of datasets, this problem disappears. Therefore, only matching the name of attributes would be problematic—some datasets do contain the information of the keyword but they are filtered out, that tells why the outcomes of ES1 is fewer than ES2 largely.

After we exploring the structure of datasets and finding the facts above, ES2 is selected as the final pattern of crawler for AURIN, which provides better performance and outcomes.

### 3.3 Data Extraction from AURIN Data forms

The item of each selected AURIN dataset could be returned by using the *GetFeature* WFS request, where only a limited number of fields are relevant to the research. Thus, a method, that could extract the necessary data out of the raw AURIN data form, is required at this stage, before saving data to remote CouchDB.

By exploring the structure of data forms returned by AURIN API *GetFeature*, the data structure could be visualized as following figure:



*Figure 3: Visualization of AURIN’s data form*

This kind of data forms can easily accumulate to size of GBs. Hence fetching AURIN data blindly would cause the server soon runs out of local storages, which would trigger unexpected error to the crawling server.

From the figure above, it is obvious that the data concerned in this research densely aggregate in the bottom of each data form: dataset’s name, attributes names and values, bbox range. On the contrary, the data located in most of top and middle part is unnecessary and should be discarded before saving to database.

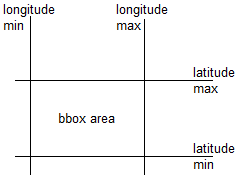
Like the methodology taken in Twitter Crawler, Regular Expression is used again in this scenario. Following table illustrates the RE for each concerned field:

|  |  |
| --- | --- |
| **Extract Target** | **Regular Expression** |
| Dataset name | <Name>aurin:([^<>]+)<\/Name> |
| Attribute\_Value pair | \"([a-z\_0-9]+)\":([0-9]+), |
| Bbox | ,\"bbox\":\[([+-]?[0-9]+\.[0-9]+),([+-]?[0-9]+\.[0-9]+),([+-]?[0-9]+\.[0-9]+),([+-]?[0-9]+\.[0-9]+)\]}}] |

After extracting data from the AURIN raw data forms, the python script finally forms an item of dictionary defined by package pycouchdb, and then send to remote database. This process is quite same as the usage of CouchDB in previous Twitter Crawler section.

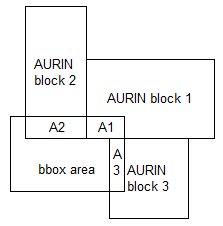
### 3.4 Scaling AURIN’s data

By defining the parameter *bbox* in the WFS *GetFeature* query, AURIN gives back the block that has the overlapped area with this specific *bbox*.



*Figure 4: Boundary of bbox definition*

However, usually, the data form for each block returned by AURIN could be much larger than the given bbox area, or only a small amount of area is shared between these two. Considering the following scenarios:



*Figure 5: Defined bbox area and AURIN data blocks*

In the figure above, the bbox area is submitted by WFS request, which is relevant to temporary research. As a result, AURIN would return AURIN blocks like block 1, block 2, block3 etc., as long as the block has shared area with bbox (A1, A2 and A3 in this scenario). Therefore, using the whole data in each returned AURIN block for the specified bbox is meaningless. Possible solution to this is scaling down the block’s data based on the area of sharing in between:

*Ab = Area of bbox*

*As = Area of sharing*

*Aa = Area of AURIN block*

*Scaling ratio = ( Ab / As ) \* ( As / Aa ) = Ab / Aa*

Thus, for each use of AURIN block’s data, this scaling ratio should be multiplied. This work has been done by the AURIN Crawler before it sending results to remote CouchDB, for it has the direct access to the information of bbox and retrieved AURIN blocks.