

# CS 172 Final Project Report: **Birdie**

Eric Ong, John Shin, Erin Wong

## Part 1 – Twitter API

### Collaboration Details:

John and Erin worked on the backend, setting up the crawler and indexing. John set up the collection of tweets using Twitter API. Erin assisted in the retrieval of HTML link titles.

Erin has significantly less experience in backend development than John, so much of the work was done collaboratively on John's computer as he helped Erin learn.

### Overview of System:

#### a. **Architecture**

Tweets are streamed in batches using the Twitter API. Live streaming tweets caused the Twitter API to end the session prematurely due to the time elapsed. Tweets are stored, then parsed, and qualifying tweets are stored separately.

Tweets that contain HTML links are handled to retrieve the <title> tag of the HTML page.

#### b. **Data collection strategy**

We stream about 50,000 tweets using the Twitter API and store them in a file. We then discard all tweets that do not qualify for our purposes: tweets not in English, tweets with no location, etc. About 5,000 tweets remain after discarding non-qualifying tweets; these are stored in another file.

We also used urllib3 to retrieve the <title> tags from each HTML page if there was one in the tweet object. Often times, users would put URLs to other tweets or Instagram posts which would essentially be a retweet, so in order to avoid those, if the URL contained Twitter or Instagram, it would automatically be filtered out. The processed title tags would also have trailing hex characters so those were removed as well.

#### c. **Data structures employed**

Formatted tweets and streamed tweets are stored in separate jsons. The formatted json is then bulk inserted into Elasticsearch which is hosted in a Bonsai cluster.

#### d. **Other**

Because only about 50,000 tweets are pulled at a time, only about 5,000 qualify for our purposes. As a result, there are often few or no tweets returned when a particular term is searched for. In the future, we would like to stream more tweets, time permitting, so that there is a larger pool of qualifying tweets to search.

### Limitations of the system

Memory would be the biggest limitation where the number of tweet objects stored is determined by the Bonsai cluster size. It also does take time to not only stream the tweets but the tweet formatting script does take a bit to execute.

Instructions on how to deploy the crawler:

## Screenshots

### Twitter stream

```
15 class TwitterStreamer():
16     def __init__(self):
17         pass
18
19     def stream_tweets(self):
20         listener = StdOutListener()
21         auth = OAuthHandler(config.api_key, config.api_secret_key)
22         auth.set_access_token(config.access_token, config.access_token_secret)
23         stream = Stream(auth, listener, tweet_mode='extended')
24
25         stream.sample()
26
27 class StdOutListener(StreamListener):
28     def __init__(self):
29         pass
30
31     def on_data(self, data):
32         try:
33             tweet = json.loads(data)
34
35             with open("tweets.json", 'a') as tf:
36                 global count
37                 print(count)
38                 count += 1
39
40             json.dump(tweet, tf)
41
42             return True
43
44     except BaseException as e:
45         pass
46
47     return True
```

### Title parser

```
32 def scrapePage(urls, newObject):
33     url = urls[0]["expanded_url"]
34
35     try:
36         if "twitter" not in url and "instagram" not in url: #don't want urls to other tweets
37             webpage = urllib.request.urlopen(url).read()
38             title = str(webpage).split('<title>')[1].split('</title>')[0]
39
40             if len(title) > 0:
41                 title = title.translate(str.maketrans('', '', string.punctuation))
42                 printable = set(string.printable)
43                 filter(lambda x: x in printable, title) #gets rid of hex values in titles
44
45                 newObject["title"] = title
46                 newObject["url"] = url
47     except:
48         print("failed url parse")
49         pass
```

## Part 2 - Indexer

### Collaboration Details:

#### 1. Architecture

Elasticsearch split into 3 shards and 2 replicas. The bulk insertion is placed into Bonsai. John did this.

#### 2. Index Structures

Elasticsearch by default indexes every inserted field in the "\_source" key, so in order to prevent that, the fields that weren't going to be queried were set to false.

John and Erin did this together.

```
{
  "tweet172" : {
    "mappings" : {
      "properties" : {
        "content" : {
          "type" : "text",
          "fields" : {
            "keyword" : {
              "type" : "keyword",
              "ignore_above" : 256
            }
          }
        },
        "coordinates" : {
          "type" : "text",
          "fields" : {
            "keyword" : {
              "type" : "keyword",
              "ignore_above" : 256
            }
          }
        },
        "date" : {
          "type" : "object",
          "enabled" : false
        },
        "entities" : {
          "type" : "text",
          "fields" : {
            "keyword" : {
              "type" : "keyword",
              "ignore_above" : 256
            }
          }
        },
        "title" : {
          "type" : "text",
          "fields" : {
            "keyword" : {
              "type" : "keyword",
              "ignore_above" : 256
            }
          }
        },
        "user" : {
          "type" : "object",
          "enabled" : false
        }
      }
    }
  }
}
```

### 3. Search Algorithm

Erin and John set up the Flask endpoints for the Elasticsearch query.

```
41     if searchBy == 'Title':
42         print("searching by title")
43         query = {
44             "query": {
45                 "bool" : {
46                     "must" : {
47                         "match" : { "title": input }
48                     }
49                 }
50             }
51         }
52     else:
53         query = {
54             "query": {
55                 "bool" : {
56                     "must" : {
57                         "match" : { "content": input }
58                     }
59                 }
60             }
61         }
62     }
```

### Limitations of System:

The query by default returns 10 results, but changing the return size also slows down the search and delays the frontend as well. This limits how many tweets the user wants to define. Storage is also another limitation, so if more tweet objects were inserted into Elasticsearch, the query itself may slow down.

### How to Deploy the System:

python3 retweet.py (this will get the initial twitter stream of 50,000)

python3 format.py (formats the 50,000 tweet objects and retrieve only the necessary fields)

python3 es.py (bulk inserts the tweetParsed.json into Elasticsearch)

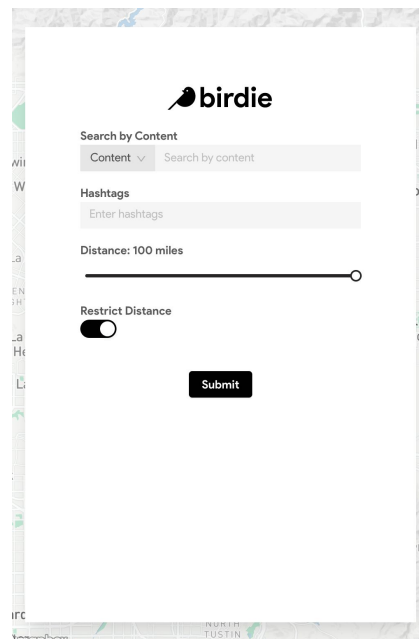
python3 app.py (starts the flask app allowing the frontend to hit the endpoints)

## Part 3 – Extension: Web-Based Interface

Our application is comprised of both a frontend and a backend. The frontend is developed using React, while the backend is developed using Flask, which connects to the Elasticsearch cluster.

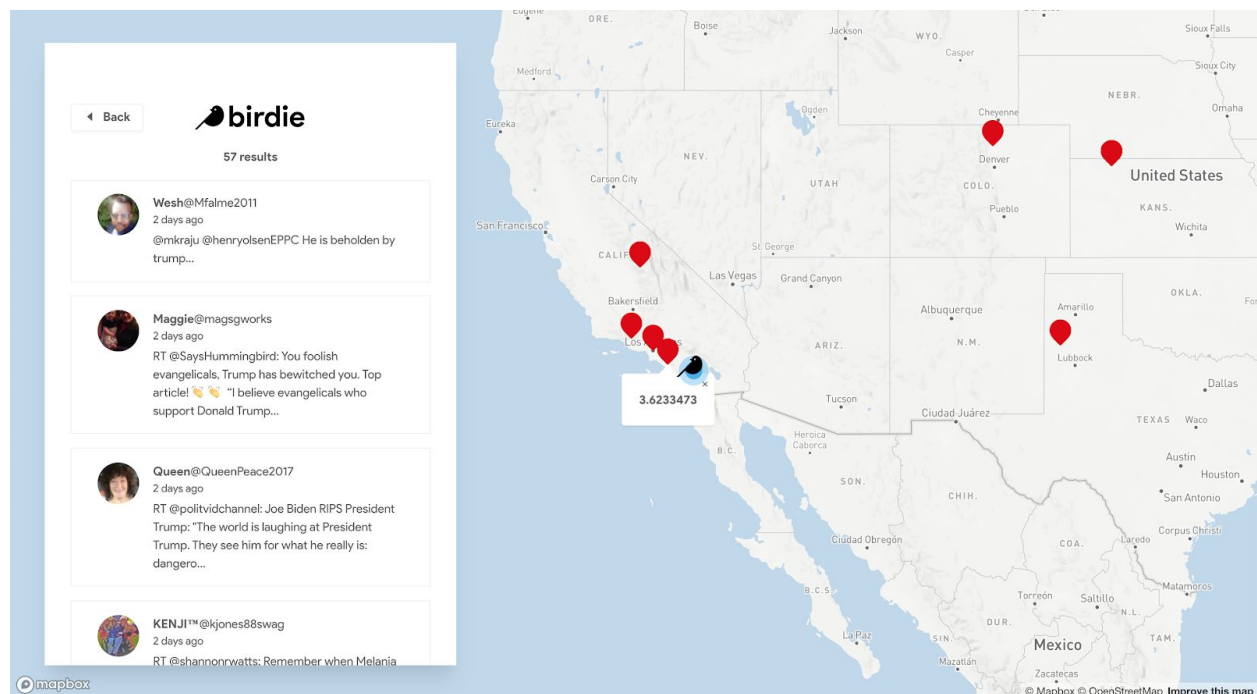
The frontend is made up of 2 main components: the Panel component and the Map component. The Panel component is where the user can query his/her desired tweets. The user can choose to search by Title or Content for the tweets, include hashtags to filter by, limit the distance of where to search around, and a toggle to

denote whether or not the location should be restricted (if this toggle is off, that means the user doesn't care about location and just wants to return all the results).



The image shows a search interface for 'birdie'. It features a search bar with a dropdown menu set to 'Content' and a 'Search by content' button. Below this is a 'Hashtags' section with an 'Enter hashtags' input field. A 'Distance' slider is set to '100 miles'. There is a 'Restrict Distance' toggle switch which is currently turned off. A 'Submit' button is at the bottom.

The Map component displays all the markers of where the tweets are located. Clicking on a marker will display a small popup that shows the score from Elasticsearch from the backend. Dragging the “birdie” will allow the user to update the location to search around by.



The image displays the Birdie search results and map interface. On the left, a sidebar shows the Birdie logo, a 'Back' button, and '57 results'. Below this, four tweet results are listed, each with a user profile picture, name, and text. The first tweet is from Wesh@Mfalme2011, the second from Maggie@magsgworks, the third from Queen@QueenPeace2017, and the fourth from KENJI™@kjones88swag. On the right, a map of the United States and Mexico is shown with several red location markers. A popup window is visible over the Los Angeles area, displaying the score '3.6233473'. The map is powered by Mapbox and OpenStreetMap.

As mentioned previously, our backend is implemented using Flask, which contains 1 main POST endpoint, */search*, this endpoint takes in the method to search by ("Content" or "Title"), the actual search query, the distance around the "birdie" object, and an array of "hashtag" strings. In addition to the endpoint, our Flask backend is the main gateway to the Elasticsearch cluster.