An approach to a Disaster Preparedness System

by

Mohammed Faisal Khan

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

UNIVERSITY OF NEW HAVEN

December 2018

Author . . . .
Department of Computer Science
May 1, 2018

Certified by . . . .
Amir Esmailpour
Associate Professor
Thesis Supervisor

Accepted by . . . .
Ronald S. Harichandran
Chairman, Department Committee on Graduate Theses

An approach to a Disaster Preparedness System

by

Mohammed Faisal Khan

Submitted to the Department of Computer Science
on May 1, 2018, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

In this thesis, I designed and implemented a disaster management platform which collects data from social media platform like twitter in real time using stream listener and stores the information in MongoDB, a NoSQL database, which is used to parsed the tweets. Hadoop map and reduce was used to extract related and meaningful data from the tweets which can be used directly using the provided API as well as have various applications using Machine Learning algorithms for modelling the various disaster scenarios. The data extracted was used to provide information about the resources available during a disaster like shelter capacity, food provisions and safe zones.

Thesis Supervisor: Amir Esmailpour
Title: Associate Professor

Acknowledgments

This is the acknowledgements section. You should replace this with your own acknowledgements.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Micro-optimization is a technique to reduce the overall operation count of floating point operations. In a standard floating point unit, floating point operations are fairly high level, such as "multiply" and "add"; in a micro floating point unit (μFPU), these have been broken down into their constituent low-level floating point operations on the mantissas and exponents of the floating point numbers.

Chapter two describes the architecture of the μFPU unit, and the motivations for the design decisions made.

Chapter three describes the design of the compiler, as well as how the optimizations discussed in section 2.1.1 were implemented.

Chapter four describes the purpose of test code that was compiled, and which statistics were gathered by running it through the simulator. The purpose is to measure what effect the micro-optimizations had, compared to unoptimized code. Possible future expansions to the project are also discussed.

1.1 Motivations for micro-optimization

The idea of micro-optimization is motivated by the recent trends in computer architecture towards low-level parallelism and small, pipelineable instruction sets [?, ?]. By getting rid of more complex instructions and concentrating on optimizing frequently used instructions, substantial increases in performance were realized.

Another important motivation was the trend towards placing more of the burden of performance on the compiler. Many of the new architectures depend on an intelligent, optimizing compiler in order to realize anywhere near their peak performance [?, ?, ?]. In these cases, the compiler not only is responsible for faithfully generating native code to match the source language, but also must be aware of instruction latencies, delayed branches, pipeline stages, and a multitude of other factors in order to generate fast code [?].

Taking these ideas one step further, it seems that the floating point operations that are normally single, large instructions can be further broken down into smaller, simpler, faster instructions, with more control in the compiler and less in the hardware. This is the idea behind a micro-optimizing FPU; break the floating point instructions down into their basic components and use a small, fast implementation, with a large part of the burden of hardware allocation and optimization shifted towards compile-time.

Along with the hardware speedups possible by using a μFPU, there are also optimizations that the compiler can perform on the code that is generated. In a normal sequence of floating point operations, there are many hidden redundancies that can be eliminated by allowing the compiler to control the floating point operations down to their lowest level. These optimizations are described in detail in section 2.1.1.

1.2 Description of micro-optimization

In order to perform a sequence of floating point operations, a normal FPU performs many redundant internal shifts and normalizations in the process of performing a sequence of operations. However, if a compiler can decompose the floating point operations it needs down to the lowest level, it then can optimize away many of these redundant operations.

If there is some additional hardware support specifically for micro-optimization, there are additional optimizations that can be performed. This hardware support entails extra “guard bits” on the standard floating point formats, to allow several unnormalized operations to be performed in a row without the loss information[1]. A discussion of the mathematics behind unnormalized arithmetic is in appendix **??**.

The optimizations that the compiler can perform fall into several categories:

1.2.1 Post Multiply Normalization

When more than two multiplications are performed in a row, the intermediate normalization of the results between multiplications can be eliminated. This is because with each multiplication, the mantissa can become denormalized by at most one bit. If there are guard bits on the mantissas to prevent bits from "falling off" the end during multiplications, the normalization can be postponed until after a sequence of several multiplies[2].

As you can see, the intermediate results can be multiplied together, with no need for intermediate normalizations due to the guard bit. It is only at the end of the operation that the normalization must be performed, in order to get it into a format suitable for storing in memory[3].

1.2.2 Block Exponent

In a unoptimized sequence of additions, the sequence of operations is as follows for each pair of numbers (m_1,e_1) and (m_2,e_2).

1. Compare e_1 and e_2.
2. Shift the mantissa associated with the smaller exponent $|e_1 - e_2|$ places to the right.
3. Add m_1 and m_2.
4. Find the first one in the resulting mantissa.

[1]A description of the floating point format used is shown in figures **??** and **??**.

[2]Using unnormalized numbers for math is not a new idea; a good example of it is the Control Data CDC 6600, designed by Seymour Cray. [**?**] The CDC 6600 had all of its instructions performing unnormalized arithmetic, with a separate `NORMALIZE` instruction.

[3]Note that for purposed of clarity, the pipeline delays were considered to be 0, and the branches were not delayed.

5. Shift the resulting mantissa so that normalized

6. Adjust the exponent accordingly.

Out of 6 steps, only one is the actual addition, and the rest are involved in aligning the mantissas prior to the add, and then normalizing the result afterward. In the block exponent optimization, the largest mantissa is found to start with, and all the mantissa's shifted before any additions take place. Once the mantissas have been shifted, the additions can take place one after another[4]. An example of the Block Exponent optimization on the expression X = A + B + C is given in figure **??**.

1.3 Integer optimizations

As well as the floating point optimizations described above, there are also integer optimizations that can be used in the μFPU. In concert with the floating point optimizations, these can provide a significant speedup.

1.3.1 Conversion to fixed point

Integer operations are much faster than floating point operations; if it is possible to replace floating point operations with fixed point operations, this would provide a significant increase in speed.

This conversion can either take place automatically or or based on a specific request from the programmer. To do this automatically, the compiler must either be very smart, or play fast and loose with the accuracy and precision of the programmer's variables. To be "smart", the computer must track the ranges of all the floating point variables through the program, and then see if there are any potential candidates for conversion to floating point. This technique is discussed further in section **??**, where it was implemented.

The other way to do this is to rely on specific hints from the programmer that a certain value will only assume a specific range, and that only a specific precision is

[4]This requires that for n consecutive additions, there are $\log_2 n$ high guard bits to prevent overflow. In the μFPU, there are 3 guard bits, making up to 8 consecutive additions possible.

desired. This is somewhat more taxing on the programmer, in that he has to know the ranges that his values will take at declaration time (something normally abstracted away), but it does provide the opportunity for fine-tuning already working code.

Potential applications of this would be simulation programs, where the variable represents some physical quantity; the constraints of the physical system may provide bounds on the range the variable can take.

1.3.2 Small Constant Multiplications

One other class of optimizations that can be done is to replace multiplications by small integer constants into some combination of additions and shifts. Addition and shifting can be significantly faster than multiplication. This is done by using some combination of

$$\begin{aligned}
a_i &= a_j + a_k \\
a_i &= 2a_j + a_k \\
a_i &= 4a_j + a_k \\
a_i &= 8a_j + a_k \\
a_i &= a_j - a_k \\
a_i &= a_j \ll m\text{shift}
\end{aligned}$$

instead of the multiplication. For example, to multiply s by 10 and store the result in r, you could use:

$$\begin{aligned}
r &= 4s + s \\
r &= r + r
\end{aligned}$$

Or by 59:

$$t = 2s + s$$

$$\begin{aligned} r &= 2t + s \\ r &= 8r + t \end{aligned}$$

Similar combinations can be found for almost all of the smaller integers[5]. [?]

1.4 Other optimizations

1.4.1 Low-level parallelism

The current trend is towards duplicating hardware at the lowest level to provide parallelism[6]

Conceptually, it is easy to take advantage to low-level parallelism in the instruction stream by simply adding more functional units to the μFPU, widening the instruction word to control them, and then scheduling as many operations to take place at one time as possible.

However, simply adding more functional units can only be done so many times; there is only a limited amount of parallelism directly available in the instruction stream, and without it, much of the extra resources will go to waste. One process used to make more instructions potentially schedulable at any given time is "trace scheduling". This technique originated in the Bulldog compiler for the original VLIW machine, the ELI-512. [?, ?] In trace scheduling, code can be scheduled through many basic blocks at one time, following a single potential "trace" of program execution. In this way, instructions that *might* be executed depending on a conditional branch further down in the instruction stream are scheduled, allowing an increase in the potential parallelism. To account for the cases where the expected branch wasn't taken, correction code is inserted after the branches to undo the effects of any prematurely

[5]This optimization is only an "optimization", of course, when the amount of time spent on the shifts and adds is less than the time that would be spent doing the multiplication. Since the time costs of these operations are known to the compiler in order for it to do scheduling, it is easy for the compiler to determine when this optimization is worth using.

[6]This can been seen in the i860; floating point additions and multiplications can proceed at the same time, and the RISC core be moving data in and out of the floating point registers and providing flow control at the same time the floating point units are active. [?]

executed instructions.

1.4.2 Pipeline optimizations

In addition to having operations going on in parallel across functional units, it is also typical to have several operations in various stages of completion in each unit. This pipelining allows the throughput of the functional units to be increased, with no increase in latency.

There are several ways pipelined operations can be optimized. On the hardware side, support can be added to allow data to be recirculated back into the beginning of the pipeline from the end, saving a trip through the registers. On the software side, the compiler can utilize several tricks to try to fill up as many of the pipeline delay slots as possible, as seendescribed by Gibbons. [**?**]

Chapter 2

Design

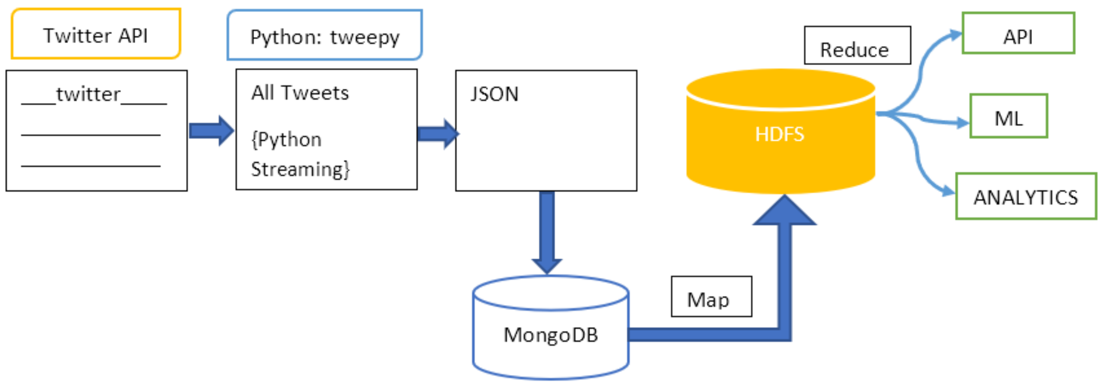


Figure 2-1: A simple caption

2.1 Twitter

2.1.1 Twitter API

Twitter as a platform has a lot of users and it generates a lot of data that can be used for analysis. Data can include user tweets, user profiles, user friends and followers, whatâĂŹs trending, etc. This data can be extracted using twitter Application Programming Interface(API). There are three methods to get this data: the REST API, the search API, and the Streaming API. The Search API is retrospective and allows you search old tweets [with severe limitations], the REST API allows you to collect user profiles, friends, and followers, and the Streaming API collects tweets in real time as they happen. As we are going to do a real time analysis of the data we find the Streaming API most suited to our needs.

The Twitter API requires a few steps:

1. Authenticate with OAuth
2. Make API call
3. Receive JSON file back
4. Interpret JSON file

2.1.2 Authenticate with OAuth

OAuth is an open standard for access delegation, commonly used as a way for internet users to grant websites or applications access to their information on other websites or applications access to their information on other websites but without giving them the passwords (Gordon, 2012). Authentication with OAuth on Twitter requires you to get keys from the Twitter developers site using a Twitter developer account. There are four keys (Consumer Key, Consumer Secret Key, Access Token, and Secret Token) that are required to access the API and need to be used during a handshake, once authenticated the program can make API calls.

2.1.3 Make API call:

When making an API call to twitter it has parameters incorporated into the URL, the wrapper looks like this:

```
https://stream.twitter.com/1.1/statuses/filter.json?track=twitter
```

This call is being done through the streaming API, where it is asking to connect to Twitter and once connection is established it will track the keyword âĂŸtwitterâĂŹ. We can specify our own keywords to track and if we do it carefully we can filter a lot data at an early stage that is irrelevant in our research.

Since, we will be using a python library called tweepy the working of an API call is abstracted from the user, nevertheless understanding the working of the making a Twitter API call is necessary to extract the relevant data.

2.1.4 Receive JSON file back:

JavaScript Object Notation (JSON) is an open standard file format that has data formatted as attribute-value pairs. The format is language independent and is commonly used for asynchronous browser-server communication for a data request.

JSON files is also the data structure that Twitter returns when an API call is made. The amount of data returned depends how we define our keywords but is usually rather comprehensive and needs to be parsed.

2.1.5 Interpret JSON file:

JSON file can be stored as raw file or can be stored using a SQL/NoSQL database. Since the data received is unstructured the most logical way to store the data would be using a NoSQL database. We will use MongoDB to store the tweets we receive and parse through. We will then run queries based on keywords we need.

2.2 Python:

While there are many different programming languages that can be used to interface with the API, the flexibility and huge community support behind python as well as its relevance in data science makes python the ideal choice for our research. Python has many libraries that has different use cases, we are going to use tweepy to stream our data from twitter.

2.3 Tweepy:

Python is a versatile language with adaptability to various use cases. These are done by extending the language by using libraries which are community created. One of these libraries is tweepy. Tweepy is open-sourced, hosted on GitHub and enables Python to communicate with Twitter platform and use its API (NovaliÄĞ, 2013). This makes it easier to access the platform to collect and monitor tweets for analysis.

2.3.1 Using tweepy:

Command to install the tweepy library:

$ pip install tweepy

Tweepy supports OAuth authentication. Authentication is handled by the tweepy.AuthHandler class. (Roesslein, 2011) A consumer token and a secret key is needed to connect with the twitter stream API, we can use the keys we generated after we created a twitter developer account.

These keys are a pair of private and public (secret and non-secret) keys and used to maintain security. The consumer key pair authorizes your program to use the Twitter API, and the access token essentially signs you in as your specific Twitter user account. This framework makes more sense in the context of third party Twitter developers like TweetDeck where the application is making API calls but it needs access to each user's personal data to write tweets, access their timelines, etc. (Dolinar, 2015)

We can import the tweepy library as below:

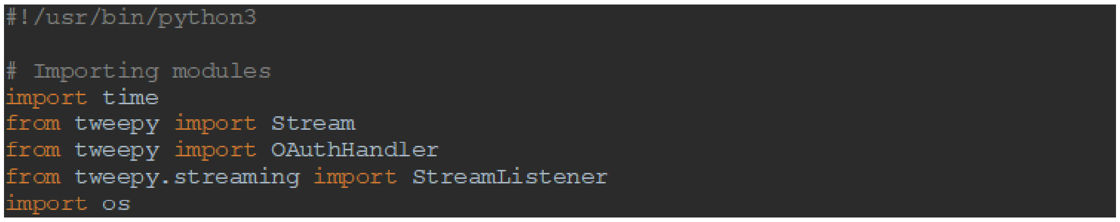
- from tweepy import Stream
- from tweepy import OAuthHandler
- from tweepy.streaming import StreamListener

The above tweepy class imports will be used to construct the stream listener.

2.4 Diving into the code:

2.4.1 Importing the modules:

Apart from the three tweepy class imports that we use to construct the stream listener, the time library will be used to create a time-out feature for the script, and the os

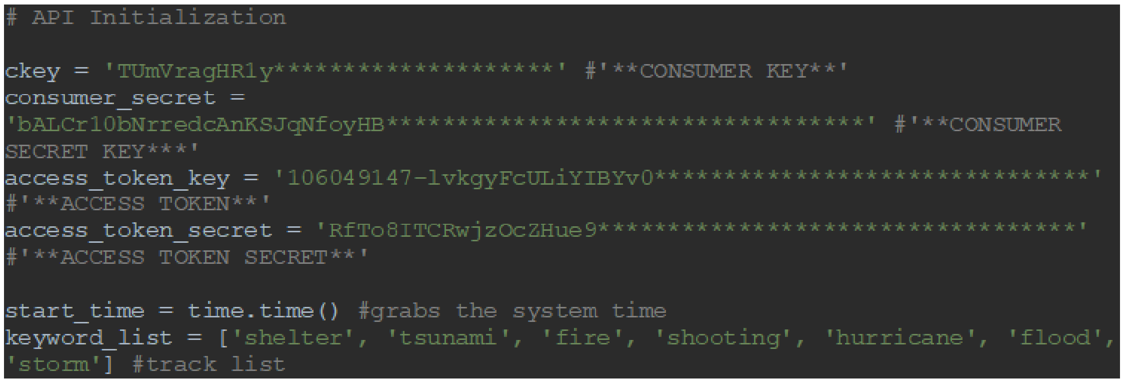


```
#!/usr/bin/python3

# Importing modules
import time
from tweepy import Stream
from tweepy import OAuthHandler
from tweepy.streaming import StreamListener
import os
```

Figure 2-2: A simple caption

library will be used to set your working directory.

2.4.2 Setting the Variables:



```
# API Initialization

ckey = 'TUmVragHR1y*******************' #'**CONSUMER KEY**'
consumer_secret =
'bALCr10bNrredcAnKSJqNfoyHB**********************************' #'**CONSUMER
SECRET KEY***'
access_token_key = '106049147-lvkgyFcULiYIBYv0******************************'
#'**ACCESS TOKEN**'
access_token_secret = 'RfTo8ITCRwjzOcZHue9**********************************'
#'**ACCESS TOKEN SECRET**'

start_time = time.time() #grabs the system time
keyword_list = ['shelter', 'tsunami', 'fire', 'shooting', 'hurricane', 'flood',
'storm'] #track list
```

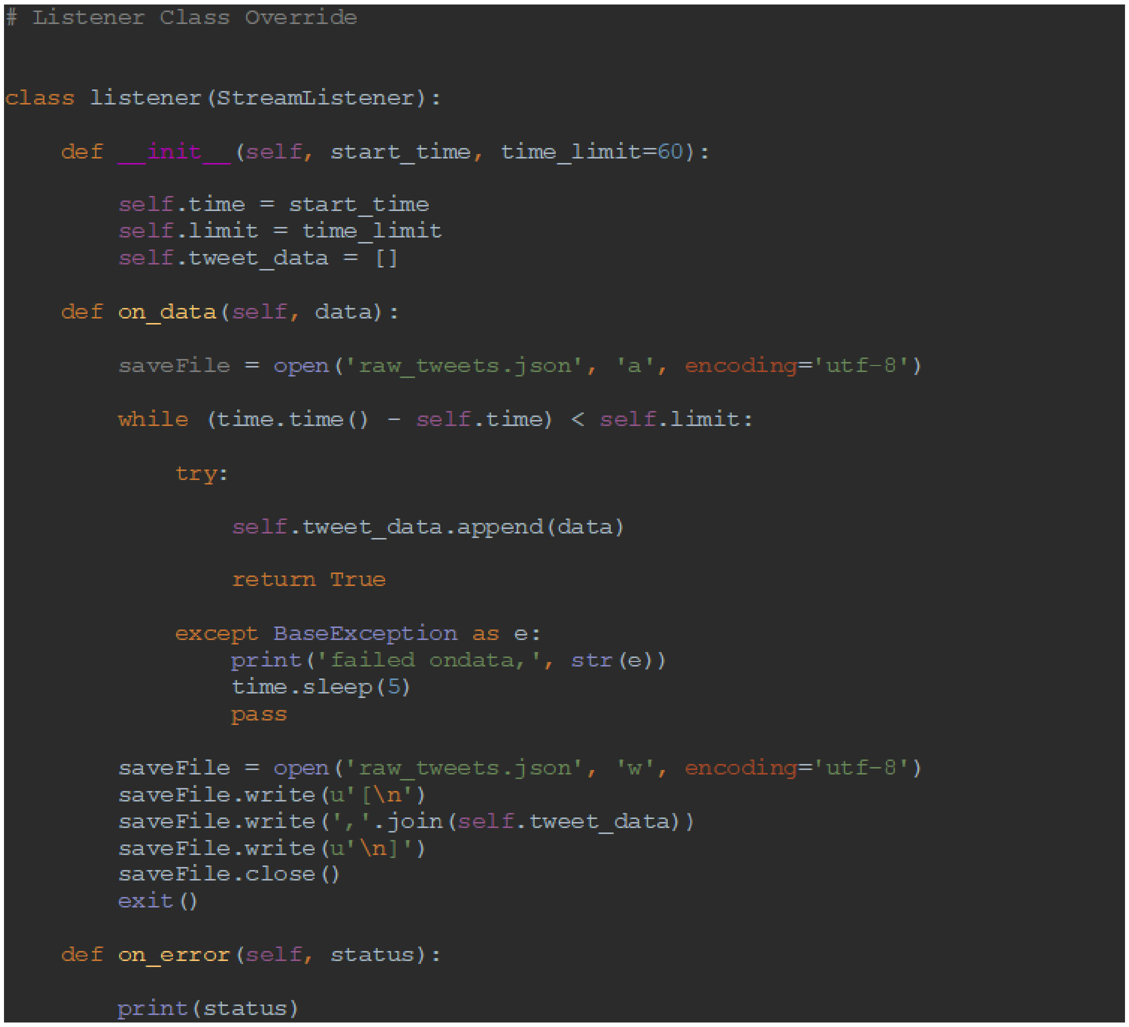
Figure 2-3: A simple caption

We have to set the above variables, which will be used in the stream listener by being fed into the tweepy objects.

2.4.3 Using and Modifying the Tweepy Classes:

The code shown below does the following:

- Creates an OAuthHandler instance to handle OAuth credentials
- Creates a listener instance with a start time and time limit parameters passed to it
- Creates an StreamListener instance with the OAuthHandler instance and the listener instance



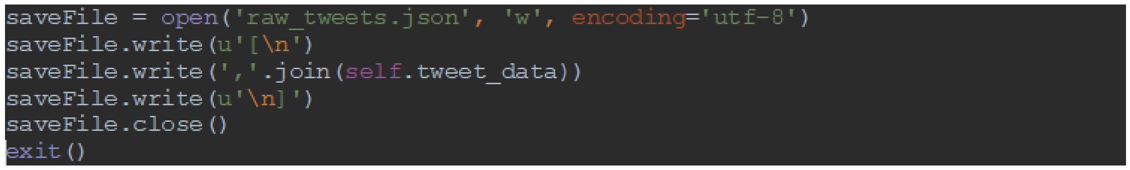
```
# Listener Class Override

class listener(StreamListener):

    def __init__(self, start_time, time_limit=60):

        self.time = start_time
        self.limit = time_limit
        self.tweet_data = []

    def on_data(self, data):

        saveFile = open('raw_tweets.json', 'a', encoding='utf-8')

        while (time.time() - self.time) < self.limit:

            try:

                self.tweet_data.append(data)

                return True

            except BaseException as e:
                print('failed ondata,', str(e))
                time.sleep(5)
                pass

        saveFile = open('raw_tweets.json', 'w', encoding='utf-8')
        saveFile.write(u'[\n')
        saveFile.write(','.join(self.tweet_data))
        saveFile.write(u'\n]')
        saveFile.close()
        exit()

    def on_error(self, status):

        print(status)
```

Figure 2-4: A simple caption

Before these instances are created, we have to "modify" the StreamListener class by creating a child class to output the data into a .csv file.

We will output the data into MongoDB by reading this csv file.

To elaborate more on the writing of the data to a file after the StreamListener instance receives data:



```
saveFile = open('raw_tweets.json', 'w', encoding='utf-8')
saveFile.write(u'[\n')
saveFile.write(','.join(self.tweet_data))
saveFile.write(u'\n]')
saveFile.close()
exit()
```

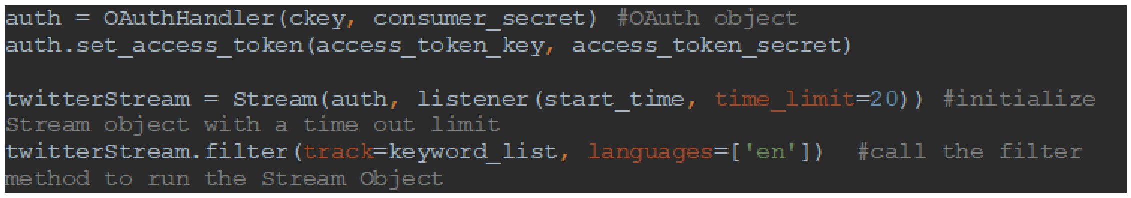
Figure 2-5: A simple caption

This block of code opens an output file, writes the opening square bracket, writes the JSON data as text separated by commas, then inserts a closing square bracket, and closes the document. This is the standard JSON format with each Twitter object acting as an element in a JavaScript array. If you bring this into Python built-in parser and the json library can properly handle it.

This section can be modified to or modify the JSON file. For example, we can place other properties/fields like a UNIX time stamp or a random variable into the JSON. We can also modify the output file or eliminate the need for a .csv file and insert the tweet directly into a MongoDB database. As it is written, this will produce a file that can be parsed by Python's json class.

After the child class is created we can create the instances and start the stream listener.

2.4.4 Calling the Stream Listener:



```
auth = OAuthHandler(ckey, consumer_secret) #OAuth object
auth.set_access_token(access_token_key, access_token_secret)

twitterStream = Stream(auth, listener(start_time, time_limit=20)) #initialize
Stream object with a time out limit
twitterStream.filter(track=keyword_list, languages=['en'])  #call the filter
method to run the Stream Object
```

Figure 2-6: A simple caption

Here the OAuthHandler uses your API keys [consumer key and consumer secret key] to create the auth object. The access token, which is unique to an individual user [not an application], is set in the following line. This will take all four of your credentials from the Twitter Dev site. The modified StreamListener class simply called listener is used to create a listener instance. This contains the information about what to do with the data once it comes back from the Twitter API call. Both the listener and auth instances are used to create the Stream instance which combines the authentication credentials with the instructions on what to do with the retrieved data. The Stream class also contains a method for filtering the Twitter Stream. The parameters are passed to the Stream API call.

2.5 MongoDB

Storing JSON tweets as a .csv file works well, but they donâĂŹt always make good flat .csv files as not every tweet has the same structure nor do every tweet contain the same fields. Some data is well nested into the JSON objects. It is possible to write a parser that has a field for each possible subfield, but this can take a lot of time as involves a lot of considerations and will also create a large .csv file or SQL database.

NoSQL databases like MongoDB greatly simply tweet storage, search and recall which eliminates the need to use an extensive tweet parser.

2.5.1 What is MongoDB?

It is a document-based database that stores data using documents rather than using tuples in tables like traditional relational databases. These documents are similar in structure to JSON objects using key-value pairs and are called BSON (Binary JSON). JSON and BSON have similar properties as JS objects and Python dictionaries.

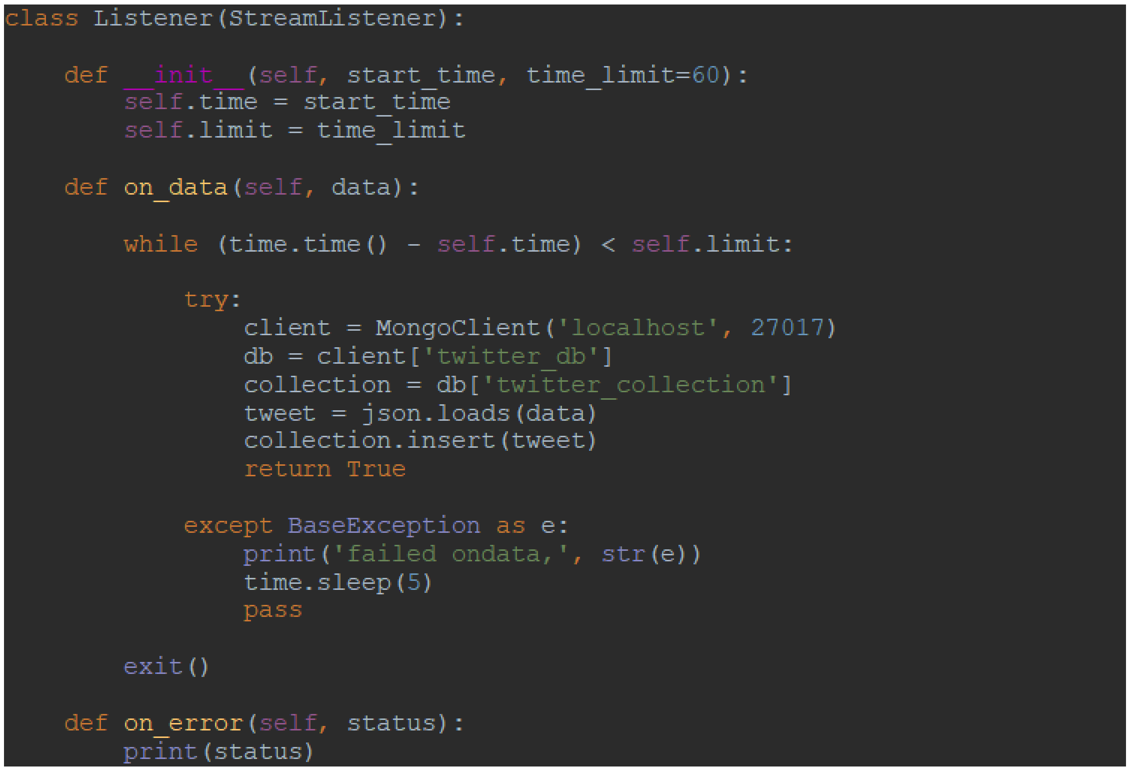
2.5.2 Why store in MongoDB?

Storing tweets in MongoDB makes sense as BSON and JSON are so similar and that makes putting the entire content of a tweetâĂŹs JSON string into an insert statement and executing that statement to store the data. This also makes recalling and searching for tweets simple although it does require a change in thought process of rather executing traditional SQL commands to treating data as OOP structures.

2.6 Storing Tweets in MongoDB:

Once MongoDB is installed and configured storing tweets is simple using the Python stream listener. Modifying the code shown above we have to import pymongo and json libraries. The json library is the default python library and will be available to import, pymongo needs to be set up using the following command:

```
$ pip install pymongo
```



```
class Listener(StreamListener):

    def __init__(self, start_time, time_limit=60):
        self.time = start_time
        self.limit = time_limit

    def on_data(self, data):

        while (time.time() - self.time) < self.limit:

            try:
                client = MongoClient('localhost', 27017)
                db = client['twitter_db']
                collection = db['twitter_collection']
                tweet = json.loads(data)
                collection.insert(tweet)
                return True

            except BaseException as e:
                print('failed ondata,', str(e))
                time.sleep(5)
                pass

        exit()

    def on_error(self, status):
        print(status)
```

Figure 2-7: A simple caption

The main changes in the code that I had to do was in the listener child class as shown below.

$MongoClientcreatestheMongoClientinstancewhichinterfaceswiththedatabase.Theclient[twitter$

2.7 Recalling Tweets from MongoDB:

The function to retrieve any document from a MongoDB database is collection.find(). Here, I can specify what I want or leave it black to get all the documents returned, in my case it will be all the tweets.

Calling using the .find() method, Python returns a MongoDB cursor, which can be iterated through by putting it in a for loop. The for loop will run the loop for each object in the iterator.

2.8 Hadoop:

Appendix A

Tables

Table A.1: Armadillos

Armadillos	are
our	friends

Appendix B

Figures

Figure B-1: Armadillo slaying lawyer.

Figure B-2: Armadillo eradicating national debt.

Bibliography

https://www.pythoncentral.io/introduction-to-tweepy-twitter-for-python/
$https://github.com/tweepy/tweepy/blob/v3.6.0/docs/auth_tutorial.rst$ https://lifehacker.com/591
oauth-what-happens-when-you-log-into-a-site-with-google-twitter-or-facebook