VIRGINIA POLYTECHNIC INSTITUTE
AND STATE UNIVERSITY

CS 5604 INFORMATION STORAGE AND RETRIEVAL

Collection Management Tweets Final Report

Payel Bandyopadhyay, Md Momen Bhuiyan, Farnaz Khaghani, Anika Tabassum, Junkai Zeng

INSTRUCTOR: PROF. EDWARD A. FOX
Department of Computer Science, Virginia Tech
Blacksburg, Virginia - 24061

January 17, 2018

Abstract

This report documents the work by the Collection Management Tweets (CMT) team, which is a part of the bigger project on building a state-of-the-art information retrieval and analysis system in support of the IDEAL (Integrated Digital Event Archiving and Library) [1] and GETAR (Global Event and Trend Archive Research) [2] projects.

The mission of the CMT team has two parts: Loading and cleaning 6.2 million tweets from two events named "Solar Eclipse" and "Las Vegas Shooting" in 2017 into HBase, an open source, non-relational, distributed database that runs on the Hadoop distributed file system for further use; and Building and storing a social network for the tweet data using a triple-store.

For the first part, our work includes:

- Making use of the work done by the previous group, where incremental update was done, to introduce a faster development process of data collection and storing
- Improving the performance of work done by the group from last year. Previously, the cleaning part, e.g., removing profanity words, plus extracting hashtags and mentions, utilized Python. This becomes very slow when the dataset scales up. We introduce parallelization in our tweet cleaning process with the help of Scala and Hadoop clusters and make use of different Natural Language Processing libraries for stop word and profanity removal.
- Along with tweet cleaning we also identify and store Named-Entity-Recognition (NER) and Part-of-speech (POS) tagging over the tweets which were not done by the previous team.

The cleaned data in HBase from this task is provided to the Classification team for spam detection and to the Clustering and Topic Analysis team for topic analysis. Collection Management Webpage team uses the extracted URLs from the tweets for further processing. Finally after the data is indexed, the Front-End team visualizes the tweets to users.

In addition to the aforementioned tasks, our responsibilities also include building a network of tweets. This entails doing research into the types of database that are appropriate for this graph. For storing the network, we use a triple-store database to store different types of edges and relationships in the graph. We also research methods ascribing importance to nodes and edges in our social networks once they are constructed, and analyze our networks using these techniques.

Contents

List of Figures 5

List of Tables 7

1 Overview 8

2 Literature Review 11

2.1 Overview of Previous Group's Work 11

2.2 Overview of Matthew's Thesis 11

2.3 Social Network for Large Dataset 13

3 Requirements 14

3.1 Data Preprocessing 14

3.2 Tweet Data Cleaning 16

3.3 Social Network Building 17

4 Design 19

4.1 Approach 19

4.2 Data Transferring to HBase 19

4.3 Extracting and Cleaning Data in HBase 20

4.3.1 HBase Schema 20

4.3.2 Data Cleaning 20

4.4 Building Social Network for Large Data Set 22

4.4.1 Resource Description Framework 22

4.4.2 Social Network Fuseki Server 23

5 Implementation 24

5.1 Timeline of the Project 24

5.2 Dataset 24

5.3 Modified Pipeline 24

5.3.1 Transferring data from JSON to HDFS 25

5.3.2 Cleaning data in HBase 27

5.3.3 Benchmark . . . 28
5.4 Social Network . . . 29
5.4.1 Data Collection . . . 30
5.4.2 Creating the RDF . . . 30
5.4.3 Transferring RDF data to Social network server . . . 31
5.4.4 Performance Evaluation . . . 31

6 User Manual **33**
6.1 Transferring Data from JSON to HBase . . . 33
6.1.1 JDK . . . 33
6.1.2 JSON4S . . . 33
6.2 Cleaning Data . . . 34
6.2.1 Stanford CoreNLP . . . 34
6.2.2 Building Matthew's framework . . . 35
6.3 Running Matthew's Framework . . . 36
6.4 Social Network . . . 37
6.4.1 Fetching list of followers and friends . . . 37
6.4.2 Creation of RDF data . . . 38
6.4.3 SPARQL Query . . . 38
6.4.4 Transferring RDF data to server . . . 39

7 Developer Manual **40**
7.1 Setting up Virtual Machine . . . 40
7.2 Adding HDD space to VM . . . 42
7.3 GETAR Server Architecture . . . 43
7.3.1 Hardware architecture . . . 43
7.3.2 Software Architecture . . . 44
7.3.3 Tweet Collection Module . . . 44
7.3.4 Apache Hadoop . . . 44
7.3.5 HDFS . . . 45
7.3.6 HBase . . . 46
7.3.7 Spark . . . 47

7.4 Social Network 48
7.4.1 List of Softwares for social network 48
7.4.2 Conversion of JSON to CSV file 49
7.4.3 Fetching Twitter followers and friends with Twitter API 51
7.4.4 Conversion of CSV to N-Triple file 52
7.4.5 SPARQL Query 54
7.4.6 Interface to Front End Team 55

8 Future Work **56**

9 Acknowledgements **57**

10 References **58**

List of Figures

1 RDF triple-store semantic relationship . . . 9
2 The flow of data through the framework . . . 12
3 Data before pre–processing. . . . 15
4 Data after pre-processing. . . . 16
5 Pipeline of tweet processing . . . 19
6 Pipeline of building the social network . . . 23
7 Workflow of the project . . . 24
8 Pipeline of Previous Team for tweet processing . . . 26
9 New pipeline for tweet processing . . . 27
10 Sample processed data in HBase . . . 28
11 Sample of N-Triple file format . . . 31
12 Sample JSON query in Fuseki based on the data we uploaded . . . 33
13 Sample code for transferring JSON value in HBase . . . 34
14 Stanford CoreNLP features . . . 35
15 Sample annotations by CoreNLP . . . 36
16 Sample code for cleaning data in HBase . . . 36
17 Sharing turned on in local server. . . . 37
18 Sample output from Matthew's framework. . . . 38
19 Check Virtualization Support in Server . . . 40
20 Add user and install virtual machine manager . . . 41
21 Start Networking for KVM . . . 41
22 Download KVM image . . . 41
23 Unzip KVM image . . . 41
24 Install KVM image . . . 42
25 Add available space to VM . . . 42
26 Run fdisk to change partition . . . 42
27 Print current partition in VM . . . 42
28 Delete the last current partition in VM . . . 43
29 Create a new partition in VM . . . 43
30 Physical volume resize . . . 43

31 Logical volume resize 43
32 Map-Reduce task flow in Hadoop 45
33 HDFS architecture 46
34 HBase architecture 47
35 Spark architecture 48
36 Part of JSON data for file oreclipse.json 50
37 Part of equivalent CSV data for file oreclipse.json 51
38 Twitter library code for fetching the followers list 52
39 Packages to import in JAVA to create N-Triple data files 52
40 Changing memory size to avoid java heap space error 54
41 Changing memory type of data set on Fuseki server 54
42 Example of a SPARQL query based on our dataset 55
43 A screen shot of dataset we created in Fuseki 56
44 A screen shot of dataset we created in Fuseki 56

List of Tables

1	Role of each member in the CMT team	9
2	Schema for metadata column family	20
3	Schema for tweet column family	21
4	Schema for clean-tweet column family	22
5	Details about the dataset for tweet processing	25
6	Benchmark for transferring data to HBase	29
7	Benchmark for data cleaning in HBase	29
8	File size of each NT file in solar eclipse and las vegas shooting events	32
9	Approximate time to convert to NT file and upload in server	32
10	URI maintained for creating each relationship	38
11	SPARQL Query	39
12	List of software routines and their version	49

1 Overview

The aim of the Collection Management Tweets team was to collect all the tweet data, clean raw tweet texts, dump the tweet data into HBase, and build a social network database to show relationships among the users and the tweets. To meet the goals of the project, the CMT team has been provided with 6 million raw tweets on the Solar Eclipse 2017 and .18 million raw tweets on the Las Vegas Shooting event, all in JSON data format. However, the JSON files were huge and contain a lot of redundant information. The CMT team has been assigned the tasks of collecting necessary information from the JSON data, processing that for further analysis by other teams, and finally, providing the information for the SOLR team to index the tweet data so users can fetch important tweet pages through the front end interfaces.

At first, we were working on the tweets the previous team had worked on. By the end of interim report 1 submission, the GTA has uploaded a new 50GB data set of tweets regarding Solar Eclipse 2017 to work on. Our first task was to convert that dataset into CSV format and remove all the redundant data and columns which are not required by other teams. For cleaning raw tweet texts our task is to:

- tokenize and lemmatize them
- remove all the stop words, profanity, and punctuation marks from the text
- remove hashtags, retweet flags, and user mentions from the text and store them separately
- find Part-of-Speech (POS) tags
- find Named-Entity-Recognition (NER) tags

To fulfill the tasks, we encountered a great challenge at the beginning, since we had only three team members. As for hardware resource, we initially had only one MacBook Pro with enough RAM for testing ideas, which forced us to perform carefully project management in order to keep everything manageable. This included:

- Leveraging every resource as much as possible, for example, Matthew's framework built on top of Spark to process tweet data, and the source code from the CMT group of last year for porting datasets into different formats;
- Carefully splitting tasks and organizing small group meetings aside from class meetings;
- Preparing report and presentation slides collaboratively using Google Doc and Overleaf.

Because of the shortage of members and the workload at the beginning, we decided to first study a tweet data processing framework, developed by a former student of Virginia Tech as Master's thesis, and put social network building as a next-phase task. We noticed that the source code was written in Scala, which was unfamiliar to all of us, and we also noticed that the code wasn't completely bug-free, which brought us a significant issue to build the tweet cleaning system on top of it. After submitting interim report 1, with Dr. Fox's help two more members joined our team which reduced our workload and we were able to work on the requirements the other teams asked for. The final role of each member of our team can be found in Table 1. We also asked for a shared virtual machine from Dr. Fox which was set up by the end of the

Table 1: Role of each member in the CMT team

Member	**Role**
Payel Bandyopadhyay	Formatting tweets from JSON to CSV and partially for running Matthew's framework
Md Momen Bhuiyan	Loading CSV files into HBase and cleaning tweets
Farnaz Khaghani	Collecting additional data from Twitter for the Social Network
Anika Tabassum	Loading data into the triple-store for the Social Network
Junkai Zeng	Modifying and running Matthew's framework to fetch information

sixth week. Working on processing tweets from JSON to CSV was a time-intensive process, as different teams have asked for different columns in HBase. It took almost 24 hours at a stretch to process Solar Eclipse tweets into CSV format. We were facing hurdles when different teams start requesting a new set of information or column to put in HBase. It was taking very long just to process them into CSV format. So, after the submission of interim report 3, we planned to use a Spark parser developed in Scala which would retrieve necessary column information directly from JSON files and put them into HBase, which has made our process faster than that of the previous team.

For the social network building, during previous reports, we planned to store edges representing different relationships between tweet and user data. After consulting with the Front End team we have decided to show only different relationships among users:

- One user replying to another user in a tweet
- One user mentioning another user in a tweet
- One user re-tweeting another user's tweet
- One user following another user
- One user followed by another user

In order to build such a graph, we conducted a study on the concept of the triple-store database, also known as Resource Description Framework (RDF) database, and found two popular systems, namely Virtuoso and Apache Jena. The RDF database, often called a semantic graph database, is able to build a schema and formal description of the data. This description, which is stored in the triple-store format, is referred to as **RDF statement** and is able to describe any subject or concept and connect it to any other object by using a predicate (verb), as is shown in Figure 1. For example, 'Selena Gomez follows Coach' can be stored in an RDF statement in the triple-store with the following components: Selena Gomez is the subject, Coach is the object, and the predicate 'follows' shows the relationship between subject and object.

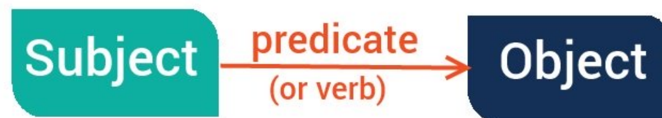


Figure 1: RDF triple-store semantic relationship

For storing the semantic relationship of users in the triple-store database, we planned to fetch the first three relationships from the JSON data we were provided as those were already in JSON files. As the last two types of relationship information was not provided in the JSON data, we planned to use the Twitter API in Python to fetch the list of friends and followers of all the users who were present in the JSON data. However, during our interim report 2 submission work, we identified that the Twitter API has a rate limit for the number of requests, which makes it very time consuming to fetch followers and friends of each user. So, next we planned to fetch the followers and friends of the top N users. Hence, we intended to identify the top N users who have the highest statuses, followers, friends, and favorites count.

2 Literature Review

For the literature review, we started by reading a key chapter of the textbook [3] which is highly relevant to our work. The content gave us the basic knowledge of tokenization, lemmatization, and stop words which we need for our cleaning of the text of tweets. We prioritized surveying background literature, which mainly includes the final report from the CMT group of last year [4] and Matthew Bock's [5] thesis on the framework for processing tweet data. The goal was to understand what related task already has been finished by the previous team. We also had to look into the social network analysis frameworks to learn how to store a large graph in a database.

2.1 Overview of Previous Group's Work

The previous group used several programming languages including Pig Latin, Java, and Python to implement the data formatting and cleaning tasks. Their pipeline had the following stages:

- First, the new tweet data is collected into CollectDB, a MySQL database. Then the data are incrementally updated into ArchiveDB, another MySQL database, using pt-archiver. At the same time, the data are inserted into a CSV file.
- For MySQL to HDFS incremental update, csv2avro is being used to convert the CSV file to an AVRO file, a format for serializing data supported by HDFS.
- Then a bash script is used to move the AVRO file and merge with another file to save disk space as the block size in the HDFS is 256 megabytes.
- After that using Stanford's CoreNLP suite, lemmatization is performed and data are stored in HBase. This is done batchwise without using the Hadoop map-reduce framework.
- Then Apache Pig Latin and Python are used to scan the HBase table, remove stop-words, and extract information like hashtags and mentions, etc.

For the social network, their pipeline was in Python using the tweepy library to collect information and apply their own algorithm for ranking users. For the visualization, they used NetworkX, a Python library. Their ranking algorithm takes information about the user like count of followers, friends, status, etc. and then calculated scores. Then they scaled the data and ordered them for visualization.

The cleaning part of this work is not leveraging the power of parallel computing, which makes the cleaning process not scalable. The social network using a single collection will not scale well with the total number of users and cannot provide insight into the different types of relationships between objects (e.g., users and tweets).

2.2 Overview of Matthew's Thesis

Besides their work, Matthew's tweet data processing framework is also helpful. In this framework, a high-level design of data structure is implemented so that users can avoid the hassle of dealing with raw data. The flow of data is shown in Figure 2.

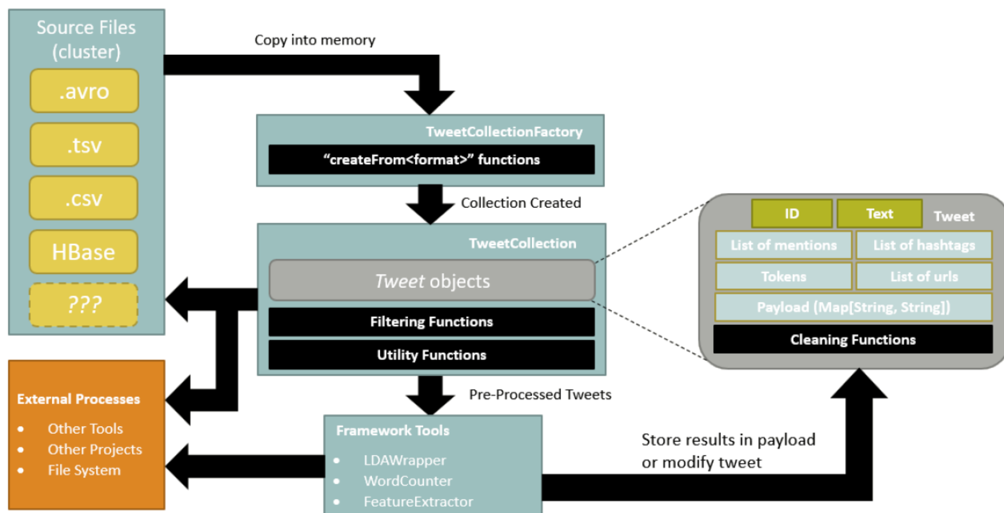


Figure 2: The flow of data through the framework

The flow is:

- First, the data sets from different sources are read from the disk. Examples include: .avro files, .csv files, and .tsv files.
- The framework will then use Spark to read the contents of the file and parallelize its content across the cluster. In this step, tweet data are wrapped into the Tweet object, which is a data structure in which hashtags, mentions, and URLs are stored.
- After the tweet collection is created, it can be cleaned by the user-defined function to filter the dataset. The framework is able to handle packaging the function and distributing it across the cluster.
- Once the collection is cleaned, various tools can be used for data analysis. For example, for topic analysis, there is a LDAWrapper already implemented in the framework. The framework provides several tools and ways to pass on data files to other external tools.

2.3 Social Network for Large Dataset

In the previous group's work, the social network was considered as a directed network. There is useful information in the underlying social network that social network analysis (SNA) [6] can uncover. Advances have been done in the SNA domain, e.g., Pajek [7] , Ucinet [8], and the SNA packages of R [9]. In this work, we are going to consider a relational network and store semantic facts and descriptions between nodes. For this, we have found a triple-store database as most suitable for our purpose. The Resource Description Framework (RDF) fulfills our requirements to create a social network with the following features:

- store and represent multiple types of data, and the semantic relationship between objects,
- query diverse and evolving data from different sources with different formats, and
- extract information and enrich content from unstructured data by text mining.

Reviewing the literature in the Semantic Web domain, we found alternatives for creating the social network in our project, and classified them, along with their limitations:

- For obtaining information from Twitter, there are several libraries created to use with the Twitter API. Some examples are: twython [10], a Python wrapper for the Twitter API, and tweepy [11], one of the most advanced and easy to use Python libraries for the Twitter API. We choose tweepy for data extraction tasks due to its powerful tools and methods for accessing the Twitter API.
- For network and social network analysis, there are specific SNA software tools. Pajek [7] is one of the most popular and complete tools with an extensive set of analysis algorithms and visualization options. The R statistical software also has packages to deal with network data. Although these tools provide useful elements for data manipulation, there are limited data management options since the data storage is based on text files only. Hence, we narrowed down our options to RDF.
- For the social network data model, there are requirements to be met. One of the most comprehensive models for this purpose is Freeman's *maximal structure experiment*: one or more kind of relations, one or more types of levels of social units, attributes of social units, and attributes that change. These attributes and features are the main reason for more elaborate models than the simple graphs.

To give us an overview about RDF data and its functionality a website [12] helped us a lot. In [12] they properly explain how to create a relationship between two nodes, and how to create relationships among the nodes in a database. They have also uploaded a demo source code in Github which helped us a lot to create our own social network database using RDF. We will discuss the details of our model in the next sections.

3 Requirements

In this section, we mention the expected final result of this project. We describe our methodologies and the libraries we used to fulfill our goal.

3.1 Data Preprocessing

A Solar Eclipse dataset was given to us. The initial dataset that was given to us was in JSON format. Below is a snippet of the JSON data:


```
{
  "contributors": "null",
  "truncated": "false",
  "is_quote_status": true,
  "id": 911027633986707457,
  "favorite_count": 5,
  ...
  ...
  ...
}
```


One of the most important tasks for our team was to pre-process the above dataset. Our data pre-processing was done in two parts: One part was for tweet cleaning and another for building a social network. For tweet cleaning we use an Apache Spark library in Scala to parse fields from a JSON file. For using Mathew's framework and building a social network we need to convert the JSON file to CSV files for storing particular fields regarding tweets.

Since the JSON file was way too large and Matthew's framework supported mostly CSV files, we converted the above dataset to CSV format. As our first task, we collaborated with other teams to gather their requirements. Since each team had their individual requirements and also the fact that their requirements changed with time we had to redo the above task multiple times.

We used a shell script to convert the JSON data to CSV format. We used the *json2csv* [13] API for the conversion. Below is an example usage. Suppose our JSON file to convert is:


```
{
        "user": {"name":"jehiah", "password": "root"},
        "remote_ip": "127.0.0.1",
        "dt" : "[20/Aug/2010:01:12:44 -0400]"
    }
{
        "user": {"name":"jeroenjanssens", "password": "123"},
        "remote_ip": "192.168.0.1",
        "dt" : "[20/Aug/2010:01:12:44 -0400]"
    }
```

```
{
        "user": {"name":"unknown", "password": ""},
        "remote_ip": "76.216.210.0",
        "dt" : "[20/Aug/2010:01:12:45 -0400]"
    }
```

We just need to use the command below:

```
json2csv -k user.name,remote_ip -i input.json -o output.csv
```

The above command will give us something like:

```
"jehiah","127.0.0.1"
"jeroenjanssens","192.168.0.1"
"unknown","76.216.210.0"
```

Now, the above API worked when we trimmed down the dataset to 100-10 rows for each JSON file. As we discussed, other teams need datasets with a million or more records; unfortunately for that the above API didn't work. Figure 3 shows a sample of our raw dataset:

contributors	truncated	is_quote_status	in_reply_to_status_i	id	favorite_count	full_text	entities__user_menti	entities__user_menti	entities__user_menti	entities__user_menti	entities__user_menti	entities__user_menti	entities__hash
null	FALSE	FALSE	null	89012333416751511	0	RT @NASASun: The	15102849	3	11	15102849	NASASun	NASA Sun & Space	
							11348282	119	124	11348282	NASA	NASA	
null	FALSE	FALSE	null	89012311488093791	0	RT @java: How to ge	125485258	3	8	125485258	java	Java	
null	FALSE	FALSE	null	89012252346728441	6	All that you touch. All that you see. All that you taste, all you feel. #Eclipse #PinkFloyd							
null	FALSE	FALSE	null	89012206172893591	0	#SpearandJackson's #Eclipse Plus 30 blades' tooth geometry has been designed to give a 30% faster cut. https://t.co/3LAR0rXqBM							
null	FALSE	FALSE	null	89012178854133351	0	How to Safely #Phot	85291227990550111	59	75	85291227990550111	Solar_Eclipse_9	SunMoonStars	
null	FALSE	FALSE	null	89012174914422781	0	RT @NASASun: The	15102849	3	11	15102849	NASASun	NASA Sun & Space	
							11348282	119	124	11348282	NASA	NASA	
null	FALSE	FALSE	null	89012072736979761	0	#codecanyon #Cat #Dog #Adventure #| #Buildbox #eclipse #| #Admob #Integration #Games https://t.co/fSURUsBOoe							

Figure 3: Data before pre–processing.

We used some Linux commands to trim down the required columns.

1. Use awk to display the field names and numbers:

   ```
   awk 'BEGIN{ FS="," }
        { for(fn=1;fn<=NF;fn++) {print fn" = "$fn;}; exit; }
        'oreclipse100.csv
   ```

2. Cut the required fields:

   ```
   cut -d, -f 5,6,7,19,22,23,41,42,43,44,160,162,429,458,464,466,
   473,477,488,489 oreclipse100.csv >oreclipse100Working.csv
   ```

3. Remove the blank columns:

```
sed -i "" '/^[[:space:]]*$/d' oreclipse100Working.csv
```

4. Add extra columns at the end:

```
awk -v RS='\r\n' 'BEGIN { OFS = "," } {print $0,
"twitter_search"}' oreclipse100Working.csv
> oreclipse100Workingout.csv
```

5. Add collection ID:

```
awk -v z="1005" '{sub(/$/,","z)} {print}'
oreclipse100Workingout.csv > oreclipse100Workingoutfinal.csv
```

Figure 4 shows a sample of our pruned dataset:

id	favorite_count	full_text	entities__urls__url	entities__urls__expa	entities__urls__displ	retweeted	coordinates__type	coordinates__coordi	coordinates__coordi	source	in_reply_to_user_id	user__id	user__utc_offs
9110276339867074	5	Better have those #E	https://t.co/Nsp6IYfO	https://twitter.com/ni	twitter.com/nickjonas	FALSE				<a href="http://twitter	null	50163968	-
9109691983434874	2	View from the Front, w/ serious projection lamp blinding me. #EclipseGlasses #SLV2(				FALSE				<a href="http://twitter	null	39746446	-
9107631048453447	0	RT @jackieevancho: Ready for #SolarEclipse2017 ! #EclipseGlasses ✔ #SunnyD ✔				FALSE				<a href="http://twitter	null	9104995997978828	null
9103904496337674	0	RT @Mel_Howes: Just put on my eclipse glasses. Whoa. #theylive #eclipseglasses #s				FALSE				<a href="http://twitter	null	4774985941	-
9103073303902044	0	#EclipseGlasses				FALSE				<a href="http://twitter	1083558361	1083558361	null
9101727432279203	0	You can use undam	https://t.co/nGKv1vo	http://ow.ly/Epl830fg	ow.ly/Epl830fgVSK	FALSE				<a href="http://www.h	null	1140925375	-
9101039698417295	2	Need my #eclipseglasses this morning □ @GoodDayAtlanta #fox5atl https://t.co/RVY				FALSE				<a href="http://twitter	null	54561501	-
9099613061737349	1	I want to back to the days when we all needed #eclipseglasses - First #Irma now #Mar				FALSE				<a href="http://twitter	null	346004773	-
9099096810191953	0	Some things are just	https://t.co/pDUNELz	https://www.instagram	instagram.com/p/BZ	FALSE	Point	-119.35	37.875	<a href="http://instag	null	126481753	-
9098787502166876	0	RT @joshgad: No matter how you wear it, we all come out looking like Marty McFly's				FALSE				<a href="http://twitter	null	586838326	-
9098554415883837	0	Eclipse Glasses Fail	https://t.co/NPZHDzp	https://goo.gl/EDHNT	goo.gl/EDHNTE	FALSE				<a href="http://twitter	null	2923059664	-
9097985589911306	0	RT @porterfiles: #eclipse2017 #eclipseglasses https://t.co/OfLH1Elmno				FALSE				<a href="http://twitter	null	8824129076564869	null
9097976138174832	1	https://t.co/hJ8o8HU	https://t.co/hJ8o8HU	https://hubs.ly/H08F_	hubs.ly/H08F_ck0	FALSE				<a href="http://www.h	null	3093013594	-
9097576129959198	0	You never know whe	https://t.co/IECqj6eq	https://www.instagram	instagram.com/p/BZ	FALSE				<a href="http://instag	null	32592624	-
9094997478403604	0	Find eclipse glasses	https://t.co/StorWeml	http://lmatfy.co/MQX	lmatfy.co/MQXdbUd	FALSE				<a href="http://lmatfy	null	1593504410	-
9094292511489638	0	RT @LaraLeaTrump: When you don't have #EclipseGlasses, you find the next best op				FALSE				<a href="http://twitter	null	7418347477734727	null
9092608663270113	0	RT @CAIRNSTACK_: Happy Eclipse Day! #eclipse2017 #safetyfirst #eclipseglasses h				FALSE				<a href="http://twitter	null	8421108448843776	null
9090696491778703	0	If you haven't yet thr	https://t.co/cy695rMx	http://ow.ly/gTYO30f	ow.ly/gTYO30f9HaM	FALSE				<a href="http://www.h	null	63464003	-
9090491813854945	0	RT @LaraLeaTrump: When you don't have #EclipseGlasses, you find the next best op				FALSE				<a href="http://twitter	null	17468666	-
9090488966886727	0	RT @wesleysnipes: Eclipse glasses? □□□ #EclipseSolar2017 #EclipseGlasses https:/				FALSE				<a href="http://twitter	null	321212887	
9089058635403468	0	RT @DragonOfAwes	https://t.co/yT1hubG	https://www.instagram	instagram.com/p/BY	FALSE				<a href="http://twitter	null	415089873	-
9088769410166087	0	#eclipse2017 #eclip	https://t.co/hbavF8aZ	https://www.instagram	instagram.com/p/BZ	FALSE				<a href="http://instag	null	86579760	-
9088715216318996	1	Smart ppl wear their	https://t.co/Eu44Z3F	https://twitter.com/th	twitter.com/thedonha	FALSE				<a href="http://twitter	null	435535517	-
9138132965198028	0	Michigan University	https://t.co/HkHERSj	http://www.leroyplast	leroyplastics.com/art	FALSE				<a href="https://ifttt.c	null	7576567610547281	-
9123869958334423	0	RT @awb_org: Give your #eclipseglasses 2nd chance! We will b collecting glasses 2				FALSE				<a href="http://buffer	null	601859624	-
9123784405563392	0	Did a Littleton Eyegl	https://t.co/L3xZEveF	http://bit.ly/2xrO4Qc	bit.ly/2xrO4Qc	FALSE				<a href="https://dlvrit	null	314145375	-
9123431718326026	0	Still holding onto th	https://t.co/Hq1Ae8S	https://www.instagram	instagram.com/p/BZ	FALSE				<a href="http://instag	null	1134259550	-
9123051465367879	0	RT @ESKTeffeteller: We are sending over 150 pairs of #eclipseglasses to @awb_org				FALSE				<a href="http://twitter	null	65518173	-
9122710805759467	0	RT @richinframes: □□ HONG KONG □□ before/ after #Hongkong #citynight #moon				FALSE				<a href="https://mobi	null	8833691442630696	null
9120215589227438	6	We are sending over 150 pairs of #eclipseglasses to @awb_org to be used in South A				FALSE				<a href="http://twitter	null	2724668910	null
9118987375548825	0	There's nothing I car	https://t.co/RnTzTjq2	https://www.instagram	instagram.com/p/BY	FALSE				<a href="http://twitter	null	9117916451877683	null
9117173432140431	0	(1) The more I think of it, the angrier I am. Selling someone bad #EclipseGlasses cou				FALSE				<a href="http://twitter	null	25303349	-
9117158819479265	0	@Walmart No refund yet on bogus #eclipseglasses you sold me a month ago. Order				FALSE				<a href="http://twitter	17137891	25303349	-

Figure 4: Data after pre-processing.

3.2 Tweet Data Cleaning

In the previous term's project, one of the main bottlenecks to data cleaning was using the native code without using Hadoop's parallelization capability. One of our main objectives was to use Matthew's framework which runs on Spark which is even faster than Hadoop's Map-Reduce framework as it does computation using in-memory processing. We also changed the data formatting pipeline by using Spark for that purpose. The following describes the list of fields we provided as per requested by other teams and not done by the previous team.

- We added some extra features to Matthew's framework. The regular expression rules for extracting the hashtag, mention and URL were changed for our needs. The sequence of those processing was also modified.

- One of the main contributions in tweet processing in this term is the inclusion of Named Entity Recognition and Parts of Speech tags in the data set. We used existing Spark APIs for the Stanford Core-NLP framework [14] for that purpose. This provided useful information for the FE team to visualize the data and Classification team to ease their work in identifying events.
- In the previous framework, there were a limited number of stop and profanity words, and those were kept in a list inside the code. Instead we used the standard list of stop-words provided by Spark.
- We removed punctuation, stop words, HTML markup, and profanity from the text. HTML markup is sometimes found in tweets while keeping UTF-8 characters.
- For profanity we were provided with a long list of profanities found by one of the team members of the Collection Management Web (CMW) team. This list can be expanded without changing the code.
- We also provided URL fields extracted from tweets, in both expanded and short form for those that are shortened. This is necessary for the Collection Management Webpages (CMW) team.
- We added the tokenized words for the Clustering and Topic Analysis (CTA) team.

3.3 Social Network Building

The main objective of creating a social network of Twitter users, followers, and tweets' attributes is to use data and relationships to rank the importance of the query results. We aim to mimic the PageRank algorithm used in search engines by using the following, followers, mentions, replying-to, and re-tweet counts and helping the Front End team to create visualizations.

The functional requirement for building the social network basically starts with the process of collecting information about users and tweets from Twitter. One possible solution for this step is using the existing library to extract the data from the Twitter API. One of these libraries is Tweepy, which is hosted on GitHub. Using the cursor methods of the library, the Tweepy library enables us to get the information necessary for building the social network through the Twitter API. As an alternative, it is possible to build a crawler from scratch in order to fetch account details like the number of followers or re-tweet and mentions counts. Using the Twitter API, it is also possible to fetch the followers, mentions, following, and both user and Twitter IDs retweeting a tweet. However, while using the Twitter API to fetch the followers of each user found in the solar eclipse 2017 dataset, we observed that it is taking a very long time to fetch them. For only 500 user nodes, it has taken more than 24 hours because of incorporating a rate-limit in the new policy for Twitter developers. Rate-limit means after fetching every 10-15 requests, the Twitter developer waits for some time to fetch new requests sent by the client. For a dataset of around 1M users, it will take a very long time. Thus, to avoid using the Twitter API for fetching records for a large number of users, we plan to collect the user IDs of the user mentions, user replying to, and user retweeting ID from the raw JSON file. The Twitter objects for each of the tweets that we require from the JSON file to create our database are:

- user_id: ID of the user object who posted the tweet
- user_mentions_id: ID of users who have been mentioned in the tweet
- in_reply_to_user_id: If the represented tweet is a reply, this field will contain the integer representation of the original tweet's author ID

- retweeted_status_user_id: If the represented tweet is a re-tweet, this field contains the user ID of the original tweet

For fetching followers and friends of each user, as there are no other options rather than using the Twitter API, we plan to fetch followers and friends of the top N influential users, i.e., who have the highest number of followers.

For the specific task of building the social network, we are not going to employ trivial graph approaches. We, instead, are going to use the RDF triple-store graph database, which enables us to store semantic facts and create a linked database with different types of nodes. The development of RDF has been motivated by the following uses, among others[12]:

- Web metadata: providing information about Web resources and the systems that use them
- Applications that require open rather than constrained information models
- Working internally with applications: combining data from several applications

Other than the objectives of RDF in web development, the RDF database has been used in other disciplines as well. Generally, the design of RDF is intended to meet the following goals:

- creating formal semantics and provable inference
- using an extensible URI-based vocabulary
- using an XML-based syntax
- supporting use of XML schema datatypes
- creating different types of relationships in one database

RDF provides a world-wide lingua franca for these processes. RDF is designed to represent information in a minimally constraining, flexible way. It can be used in isolated applications, where individually designed formats might be more direct and easily understood, but RDF's generality offers greater value from sharing.

To allow building an RDF graph database for social networks, Apache Jena provides an API to create an RDF triple-store database model. Also, Jena provides a server named Fuseki to store triple-store data. We intend to use Apache Jena for creating a triple-store database and store data.

4 Design

4.1 Approach

Our approach to the project involves understanding what the previous team did and how their framework works. We did the literature review of their work and also looked into the existing framework by Matthew which was suggested by the Instructor. We also contacted Dr. Shamimul Hasan for getting suggestions regarding existing tools for the triple-store. Figure 5 presents an overview of tweet processing we used in this project.

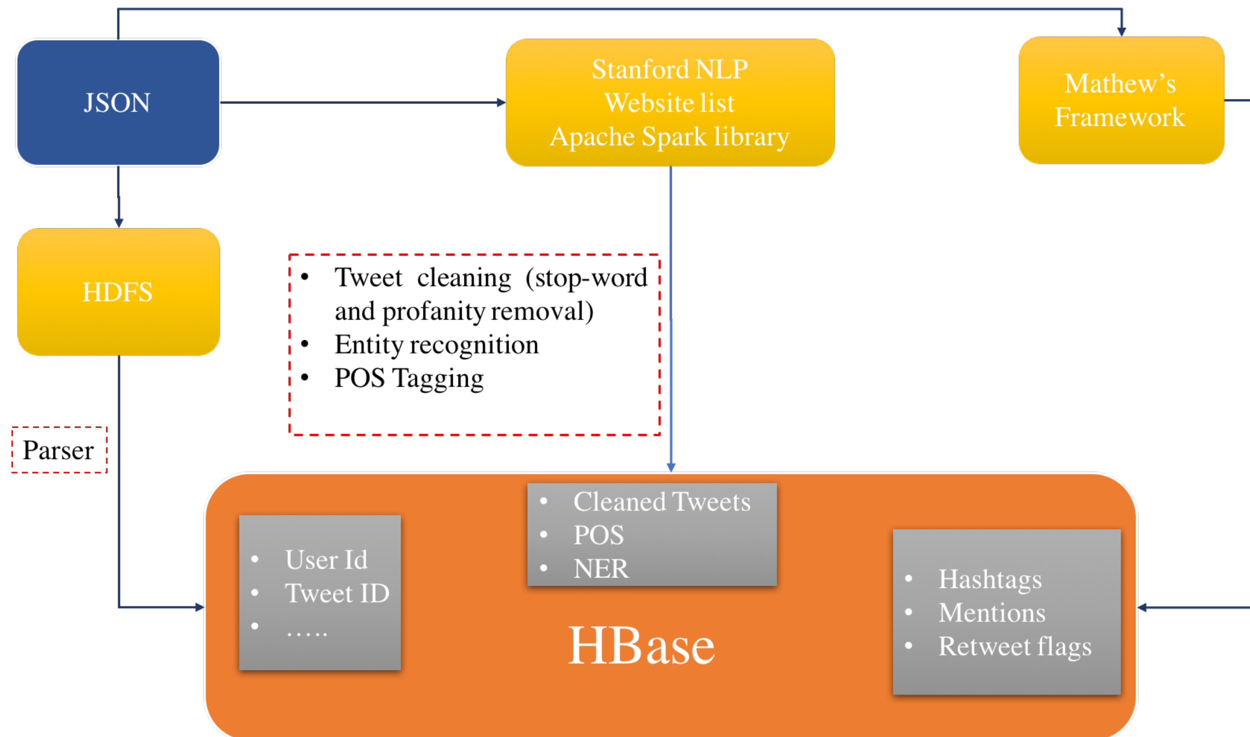


Figure 5: Pipeline of tweet processing

4.2 Data Transferring to HBase

Once the original JSON file is given we store the data inside HDFS first. Then a "scala" script was used for extracting necessary columns based on the requirements of the other team. Tweet IDs were used as row key for the HBase rows. Most of the information like tweet text, language, time, user information, URLs, etc. were extracted from the tweets. Others were extracted during the tweet cleaning step. For the purpose of social network building, JSON was converted to CSV files. As they needed a small amount of information it was appropriate. This included usernames that are mentioned or retweeted from, which we will be making use of in the future social network building step.

4.3 Extracting and Cleaning Data in HBase

After data from the JSON file is loaded into HBase it has to be cleaned so that other teams can use that. For this project, all of the teams have collaborated in creating an HBase schema. The design of the schema, as well as the process of cleaning data, are discussed below.

4.3.1 HBase Schema

As the data processing depends on the HBase schema, it is discussed first. All of the teams in this project collaborated on the database schema so that there is minimal knowledge required for us to interact. The process was initialized by the FE team who provided what their requirements were. Then we gave our feedback on what is possible. Based on that we agreed upon three column families in HBase: metadata, tweet, and clean-tweet.

The metadata column family provides the columns shown in Fig. 2. These columns provide metadata for the data. The tweet column family provides information about the tweet text along with other information like user and location information about the tweet. The clean-tweet column family provides columns that are needed by the other teams. Here clean-text-cla, clean-text-cta, and clean-text-solr are the columns that have been cleaned using the steps described in the next subsection.

Table 2: Schema for metadata column family

Column Family	Column-name
metadata	doc-type
metadata	collection-id
metadata	collection-name
metadata	dummy-data

4.3.2 Data Cleaning

After getting the initial data into HBase, a Scala script is used to clean them. The following steps are used for that:

- First, URLs and mentions are removed from the tweet text
- Then all punctuation is removed
- After that stop words are removed
- Finally they are tokenized and lemmatized

Note that if we removed the hashtags with the mentions the clean text could have been empty. For that reason the CLA team has asked us not to remove hashtags from the text.

Finally using Matthew's framework and Stanford CoreNLP, appropriate columns are populated into HBase like the columns in Table 4.

Table 3: Schema for tweet column family

Column Family	Column-name
tweet	tweet-id
tweet	like-count
tweet	comment-count
tweet	retweet-count
tweet	archive-source
tweet	source
tweet	text
tweet	screen-name
tweet	user-id
tweet	tweet-deleted
tweet	user-deleted
tweet	contributor-enabled
tweet	created-timestamp
tweet	created-time
tweet	language
tweet	geo-type
tweet	geo-0
tweet	geo-1
tweet	url
tweet	to-user-id
tweet	profile-img-url
tweet	long-url
tweet	user_mentions_id_str
tweet	user_mentions_name
tweet	user-name
tweet	user_location
tweet	user_followers_count
tweet	user_favourites_count
tweet	user_statuses_count
tweet	user_lang
tweet	user_friends_count
tweet	place_country_code

Table 4: Schema for clean-tweet column family

Column Family	Column-name
clean-tweet	clean-text-solr
clean-tweet	clean-text-cla
clean-tweet	clean-text-cta
clean-tweet	NER
clean-tweet	POS
clean-tweet	rt
clean-tweet	geo-location
clean-tweet	spatial-coord
clean-tweet	spatial-bounding
clean-tweet	solr_geom
clean-tweet	geom-type
clean-tweet	hashtags
clean-tweet	mentions
clean-tweet	long-url
clean-tweet	dates
clean-tweet	sner-people
clean-tweet	sner-organizations
clean-tweet	sner-locations
clean-tweet	tweet-importance

4.4 Building Social Network for Large Data Set

For building the social network for a large dataset, after collecting user IDs corresponding to different relationships using the Twitter API and .csv file, we need to create a model to store those data on a social network server. Figure 6 presents the pipeline of building the social network for the Twitter dataset we employed for this project. Our pipeline to build the social network is processed in four steps. First, we retrieve the mentions, re-tweets, and replies relations from JSON and convert those fields to a CSV file. So, for each user ID, it contains the corresponding users who are related to this user by the above-mentioned relationships. Second, we fetch the followers and friends of users using the Twitter API. Third, we convert the CSV file to triples or N-Triple format file which is compatible with the RDF triple-store database. Also we convert users' followers and friends into triples and store them in the N-triple file. Finally, we upload the N-Triple files into the RDF triple-store server, i.e., Fuseki server.

To follow the pipelines mentioned above, we explore different websites regarding RDF in [12], [15].

4.4.1 Resource Description Framework

Resource Description Framework (RDF) is a framework for representing information in the Web. It is advantageous to handle URI references, and for representing semantic web relations. Its modeling approach is similar to an entity-relationship class diagram, however, it represents every relation in *subject-predicate-object* triple format. An RDF model provides three different serializations or file formats to represent a

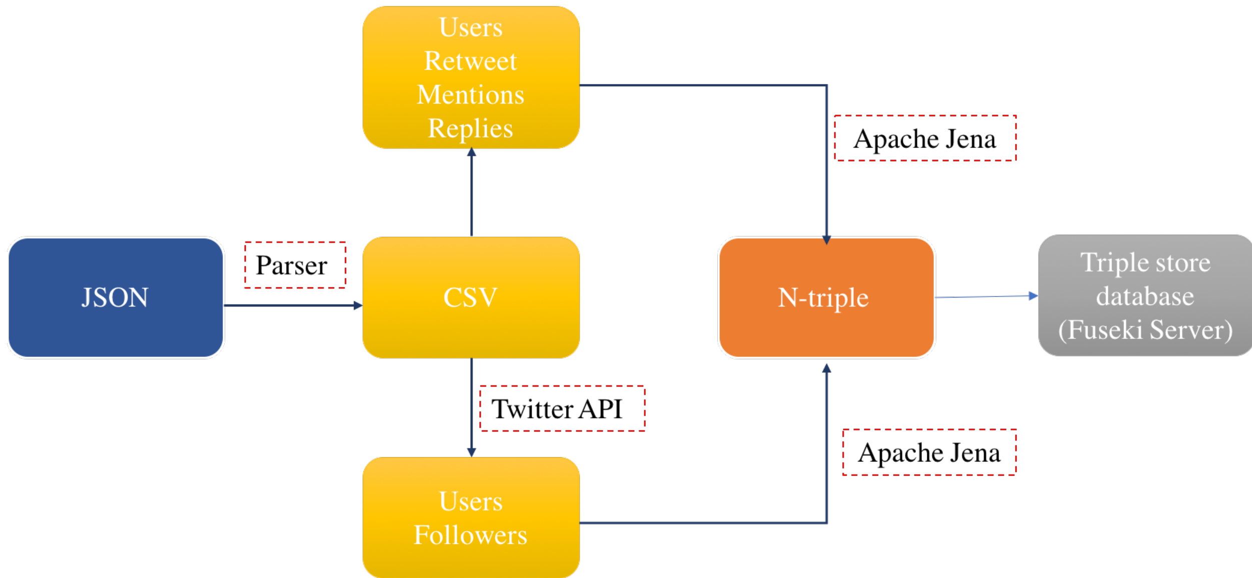


Figure 6: Pipeline of building the social network

model, e.g., Turtle (.ttl), N-Triple (.nt). For representing a graph structure, the N-triple format is more favorable to store the social network relationships, to represent each user as subject or object, and to render a relationship as an edge between them. For creating RDF data, we used the EclipseRDF4J framework and Apache Jena library. To have a proper knowledge about how to use these tools we had a lot of help from [16] and a Github demo project [12].

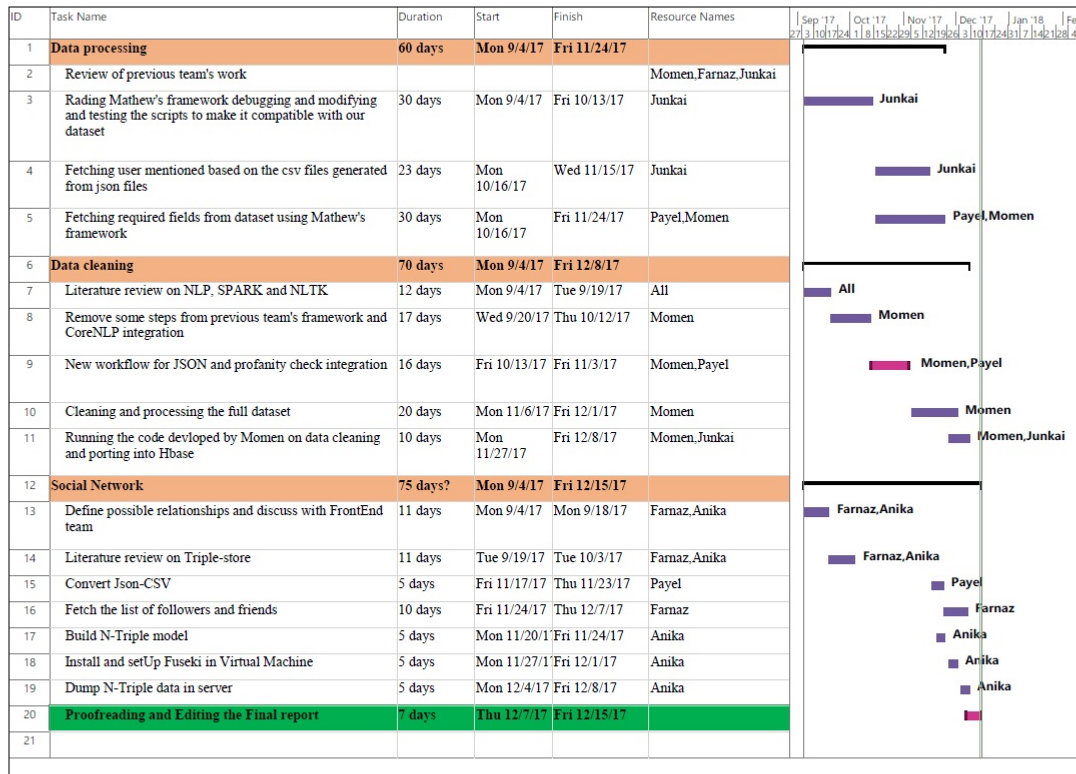
4.4.2 Social Network Fuseki Server

After fetching user-user relationships mentioned in Section 3.3, and creating RDF N-Triple format data mentioned in the previous section, we put all the data on a server. There are different servers available to store RDF data. We chose to use the Fuseki server provided by Apache Jena. Since it is necessary to have access to this server both by our team as well as the Front End team, we asked the GTA to give us access to an account in a Virtual Machine (VM). We installed the server in the VM provided by the GTA and uploaded the RDF N-Triple (.nt) files to the server. We provided an interface or URI to the Front End team to directly access the server and fetch results for their queries.

5 Implementation

5.1 Timeline of the Project

Figure 7 presents the work-flow and dependency of the tasks during the semester.



ID	Task Name	Duration	Start	Finish	Resource Names
1	**Data processing**	**60 days**	**Mon 9/4/17**	**Fri 11/24/17**	
2	Review of previous team's work				Momen,Farnaz,Junkai
3	Rading Mathew's framework debugging and modifying and testing the scripts to make it compatible with our dataset	30 days	Mon 9/4/17	Fri 10/13/17	Junkai
4	Fetching user mentioned based on the csv files generated from json files	23 days	Mon 10/16/17	Wed 11/15/17	Junkai
5	Fetching required fields from dataset using Mathew's framework	30 days	Mon 10/16/17	Fri 11/24/17	Payel,Momen
6	**Data cleaning**	**70 days**	**Mon 9/4/17**	**Fri 12/8/17**	
7	Literature review on NLP, SPARK and NLTK	12 days	Mon 9/4/17	Tue 9/19/17	All
8	Remove some steps from previous team's framework and CoreNLP integration	17 days	Wed 9/20/17	Thu 10/12/17	Momen
9	New workflow for JSON and profanity check integration	16 days	Fri 10/13/17	Fri 11/3/17	Momen,Payel
10	Cleaning and processing the full dataset	20 days	Mon 11/6/17	Fri 12/1/17	Momen
11	Running the code devloped by Momen on data cleaning and porting into Hbase	10 days	Mon 11/27/17	Fri 12/8/17	Momen,Junkai
12	**Social Network**	**75 days?**	**Mon 9/4/17**	**Fri 12/15/17**	
13	Define possible relationships and discuss with FrontEnd team	11 days	Mon 9/4/17	Mon 9/18/17	Farnaz,Anika
14	Literature review on Triple-store	11 days	Tue 9/19/17	Tue 10/3/17	Farnaz,Anika
15	Convert Json-CSV	5 days	Fri 11/17/17	Thu 11/23/17	Payel
16	Fetch the list of followers and friends	10 days	Fri 11/24/17	Thu 12/7/17	Farnaz
17	Build N-Triple model	5 days	Mon 11/20/1	Fri 11/24/17	Anika
18	Install and setUp Fuseki in Virtual Machine	5 days	Mon 11/27/1	Fri 12/1/17	Anika
19	Dump N-Triple data in server	5 days	Mon 12/4/17	Fri 12/8/17	Anika
20	**Proofreading and Editing the Final report**	**7 days**	**Thu 12/7/17**	**Fri 12/15/17**	
21					

Figure 7: Workflow of the project

5.2 Dataset

We processed tweet from two events, "Solar Eclipse" and "Las Vegas Shooting". The first event happened in Aug., 2017 and the second one in Oct., 2017. The tweets were collected using hashtags. The table 5 the details about the tweets in each event. In "Solar Eclipse" event approximately 5.7m tweets were unique and in "Las Vegas Shooting" about 173k tweets were unique.

5.3 Modified Pipeline

The previous CMT team did a great job in creating the pipeline for transferring data from MySQL to HBase. In this term we are improving upon that. This subsection discusses these improvements.

Table 5: Details about the dataset for tweet processing

Event	Collection Name	Collection Id	No. of Tweets
Solar Eclipse	#Eclipse2017	994	1365388
Solar Eclipse	#Eclipse	995	1189492
Solar Eclipse	#solareclipse	996	569352
Solar Eclipse	#August21	997	1376
Solar Eclipse	#solareclipse2017	1001	2572126
Solar Eclipse	#totalsolareclipse	1003	21187
Solar Eclipse	#eclipseglasses	1004	14171
Solar Eclipse	#oreclipse	1005	12882
Solar Eclipse	#totaleclipse	1006	478567
Las Vegas Shooting	#shooting #LasVegas	1024	21587
Las Vegas Shooting	#VegasShooting	1025	152816

5.3.1 Transferring data from JSON to HDFS

In the previous CMT team's pipeline initially they were given a CSV file. They first sanitized the raw text by doing some initial cleaning as well as formatting. Then they converted the data into AVRO format and transferred that data to HDFS. Finally they stored the data inside HBase. Figure 8 shows the previous architecture. These steps have several drawbacks:

- They did a major part of cleaning in HBase. So initial cleaning was an additional step.
- There were multiple intermediate data formats that didn't add anything to the pipeline.
- Normalizing newline removes extra information that can be further used by other teams.
- Removing all types of quotes and non-ASCII characters changes the whole information in tweets.
- As several steps were done without any parallelism, that reduced efficiency.

To address these problems we removed these step from the initial pipeline and performed those tasks in Spark. Figure 9 provides the current architecture of the pipeline. Here JSON file is first put into the HDFS. The "Put" command used here uses Hadoop's efficient parallel architecture. Then a Spark library named "json4s" is used to parse the JSON data and store into HBase. So there are no intermediate formats in this step.

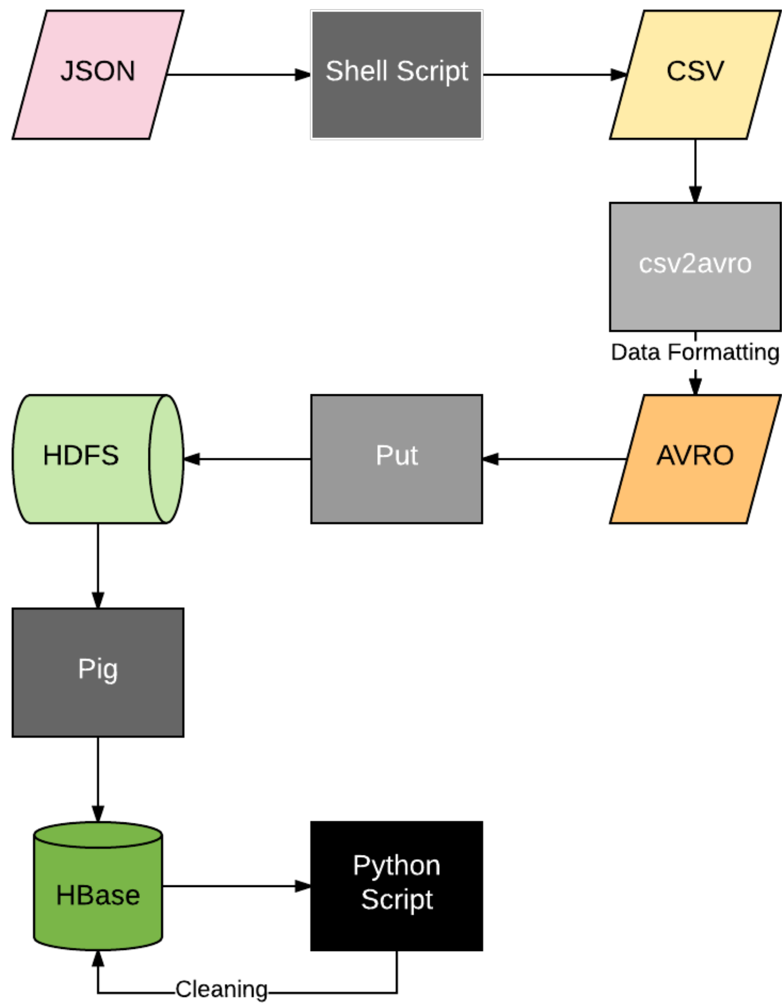


Figure 8: Pipeline of Previous Team for tweet processing

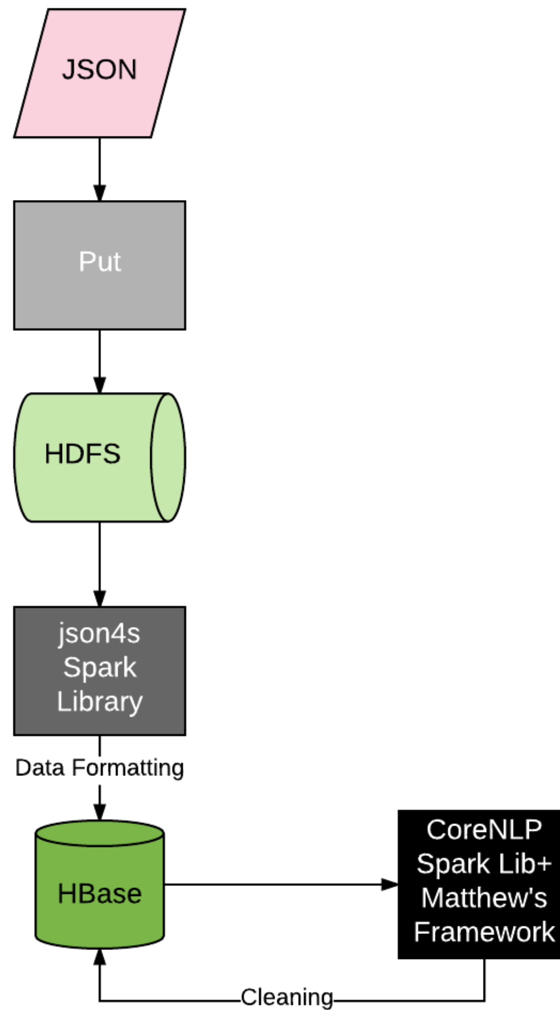


Figure 9: New pipeline for tweet processing

5.3.2 Cleaning data in HBase

The previous team's code for cleaning needed optimizing as they didn't use any parallelism. For that purpose we have integrated Matthew's framework and Stanford CoreNLP into our system for the purpose of speeding up the cleaning process. For our different purpose a different framework is used, which is described below.

- Matthew's Framework: For the purpose of reading the data from HBase, Matthew's framework is used. This framework is also used for extracting retweet, hashtag, and mention occurrences.
- Stanford CoreNLP: For the purpose of getting the named entities, lemmatization, and parts of speech tags, Stanford CoreNLP is used. Then the data is stored in the format the FE team wanted.

- Spark ML Library: Spark ML Library is used for the purpose of removing stopwords.
- Scala Code: Plain Scala code is used for the purpose of removing profanity as well as for the purpose of storing the data into HBase.

Figure 10 shows a single row of the data from HBase. In the figure the column value is the name of the column, timestamp is the timestamp of when it was last modified, and value is the value of that column. The schema has been modified to the one proposed recently.

```
COLUMN+CELL
column=clean-tweet:clean-text-cla, timestamp=1508204304198, value=rt vtbigevent : HAPPY SNOW DAY !!
e time  sign   tbe  https://t./aiphruqdnr https://t./gsz14qkaoy
column=clean-tweet:clean-text-cta, timestamp=1508204304197, value=rt vtbigevent : HAPPY SNOW DAY !!
e time  sign   tbe  https://t./aiphruqdnr https://t./gsz14qkaoy
column=clean-tweet:clean-text-solr, timestamp=1508204302945, value= RT @VTBigEvent: HAPPY SNOW DAY!
ch more free time to sign up for TBE @ https://t.co/aipHrUqDnr https://t.co/gSz14Qkaoy
column=clean-tweet:created-month, timestamp=1508204303050, value=01
column=clean-tweet:created-year, timestamp=1508204303050, value=2016
column=clean-tweet:geo-location, timestamp=1508123987989, value=
column=clean-tweet:hashtags, timestamp=1508204303050, value=
column=clean-tweet:lemmatized, timestamp=1508202484537, value=rt @vtbigevent : HAPPY SNOW DAY !! so
ore free time to sign up for tbe @ https://t.co/aiphruqdnr https://t.co/gsz14qkaoy
column=clean-tweet:mentions, timestamp=1508204303050, value=VTBigEvent
column=clean-tweet:readable-collection, timestamp=1508184017829, value=
column=clean-tweet:readable-lang, timestamp=1508184017829, value=
column=clean-tweet:rt, timestamp=1508204303050, value=1
column=clean-tweet:sner-locations, timestamp=1508202484537, value=
column=clean-tweet:sner-organizations, timestamp=1508202484537, value=
column=clean-tweet:sner-people, timestamp=1508202484537, value=
column=clean-tweet:tweet-importance, timestamp=1508123987989, value=
column=tweet:archivesource, timestamp=1508123987989, value=twitter-search
column=tweet:colnum, timestamp=1508123987989, value=709
column=tweet:created_at, timestamp=1508123987989, value= Sun Jan 24 20:23:23 +0000 2016
column=tweet:from_user, timestamp=1508123987989, value= steven_black28
column=tweet:from_user_id, timestamp=1508123987989, value= 2347438990
column=tweet:geo_coordinates_0, timestamp=1508123987989, value=0.0
column=tweet:geo_coordinates_1, timestamp=1508123987989, value=0.0
column=tweet:geo_type, timestamp=1508123987989, value=
column=tweet:iso_language_code, timestamp=1508123987989, value= en
column=tweet:profile_img_url, timestamp=1508123987989, value= http://abs.twimg.com/images/themes/th
.gif
column=tweet:source, timestamp=1508123987989, value= <a href="http://twitter.com/download/iphone" r
ollow">Twitter for iPhone</a>
column=tweet:text, timestamp=1508123987989, value= RT @VTBigEvent: HAPPY SNOW DAY!! So much more fr
 to sign up for TBE @ https://t.co/aipHrUqDnr https://t.co/gSz14Qkaoy
column=tweet:time, timestamp=1508204302945, value=1453667003
column=tweet:to_user_id, timestamp=1508123987989, value=
column=tweet:tweet_id, timestamp=1508123987989, value= 691355637544914944
column=tweet:url, timestamp=1508204302934, value=
```

Figure 10: Sample processed data in HBase

5.3.3 Benchmark

We benchmarked two stages separately. For the parsing step we ran the code on the following configuration:

- No. of Tweets: 173k

- File Size: 1.06GB
- RAM: 24GB
- CPU cores: 12

The table 6 shows the result for transferring data to HBase. It took about 2 minutes on average to parse and insert the JSON data into HBase.

Table 6: Benchmark for transferring data to HBase

Iteration	Time Taken
First	110.4 sec
Second	109.1 sec
Third	100.1 sec
Average	106.5 sec

For the cleaning stage we used about 21.5k tweets with the previous configuration for benchmarking. Table 7 shows the result for this step. The previous team didn't have most of the features in this step which is why we couldn't compare this with the previous team's timing.

Table 7: Benchmark for data cleaning in HBase

Iteration	Time Taken
First	32.96 min
Second	30.23 min
Third	31.95 min
Average	31.71 min

5.4 Social Network

The main objective of this task, as mentioned before, is to build a network based on the data and relationships of users and tweets. The ultimate goal is to assign an importance factor to the tweets using the interaction of users, user mentions, user replies, user retweets, followers, and friends. This task includes four major subtasks:

1. Data Collection
2. Converting data into RDF triple format
3. Transferring RDF data to social network server and providing interface to the Front End team for visualization
4. Assigning an importance factor for Twitter users and tweets

5.4.1 Data Collection

The first step is to collect the data necessary for building our social network. Since our approach includes the interaction of users and followers and also tweets and re-tweets and mentions, we need to extract all these entities and build a database accordingly. Mentions, re-tweets, and replying-to already existed and have been extracted in the tweet preprocessing step. Thus, our task in the data collection step is to collect the list of the followers and friends of the users. For this, we used the tweepy library. Code for collecting the followers of a user exists in the previous group's work. However, recently Twitter enforces SSL encryption for apps connecting to its API. Due to the change of this policy, it is not possible to use the previous team's code. We decided to build our own code from scratch, compatible with the new policies of the Twitter API. We use the list of the users in .json format as the input and return the list of the followers for each user in the list in .txt format. To avoid the rate-limit we modified our code based on the existing Python script available in the Github repository [17].

To enhance the relationship of the users, we also collected the list of the friends for each user. As with the followers, fetching the following list of a user has a rate-limit in the Twitter API. Thus, we had to change the module in the Twitter library to fetch the following in the intervals in which the rate limit does not stop the code. We created around five authentication keys using a Twitter developer account and used these five keys in our Python code to fetch user followers concurrently. So, at the same time it can fetch followers of five users. We plan to fetch the top 1000 users of each event. To get the top N users, we weight each user with different weights based on their follower, friends, and statuses count. Then we multiply each of the counts with a weight and get the average weight. So, for the first 1000 users who have the highest weights we fetch followers of those users using the Twitter API. For example, if a user has 1200 followers, 1500 friends, and 100 statuses, we set up a score of 0.4 to followers, 0.3 to friends and 0.3 to statuses. We assume that users with more followers are given much importance in a tweet. So, the weight of the user is $1200 * 0.4 + 1500 * 0.3 + 100 * 0.3 = 960$

5.4.2 Creating the RDF

For the social network creation, we decided to take a different approach than previous teams. Instead of building a simple network, we employed the relational network which can specify the semantic relationship between entities. For this, we have to classify the entities as to the different types of nodes. For the social graph at first we planned to create two different types of nodes: users and tweets. However after interim report 2 the Front End team asks for the user relationships only. Thus, we create nodes only for the users.
Users: Users are the most important nodes in the social network. They form a relationship with other users as their followers or following entities. They are also involved in different types of relationship with tweets via re-tweets, mentions, replies, etc.
Based on users we identified, we create three types of relationship as the edges in our social network. These edges will then be used as the predicate feature in the RDF database and connect two userIDs.
user-mentions-user edges: A user can mention another user in a tweet. So, the predicate in the RDF would be *mentions*.
user-replyingTo-user edges: A user can reply to another user's tweet. So, the predicate in the RDF would be *ReplyingTo*
User-retweeting-user edges: A user may re-tweet another user's tweet; the predicate would be *retweeted*.
user-follower-user: These edges simply relate two users who are followers or following another user, thus

the predicate would be *followedBy* or *friendOf*.

To build the RDF database we used Apache Jena as one of the most common and popular software packages for creating RDF databases. Jena provides a programmatic environment for RDF, SPARQL, and OWL. We build a RDF model from the data we collected from JSON using the Apache Jena library in Java and output that model in N-Triple format file which is compatible for the social network server. Figure 11 shows a sample of our social network model of solar eclipse 2017 data in N-triple format file. Each line is a triple representing a user-user relation. Here the first part of each line is the subject of the triple, second part is the predicate or relation of the triple, and the third part is the object of a triple.

```
<http://example.org/893259045947056128> <http://xmlns.com/SNR/0.1/mentions> "148517828" .
<http://example.org/893259045947056128> <http://xmlns.com/SNR/0.1/mentions> "14159148" .
<http://example.org/893259045947056128> <http://xmlns.com/SNR/0.1/in_reply_to> "893259045947056128" .
<http://example.org/70041882> <http://xmlns.com/SNR/0.1/mentions> "127041492" .
<http://example.org/70041882> <http://xmlns.com/SNR/0.1/in_retweet_to> "127041492" .
<http://example.org/29710286> <http://xmlns.com/SNR/0.1/mentions> "277964435" .
<http://example.org/29710286> <http://xmlns.com/SNR/0.1/in_retweet_to> "277964435" .
<http://example.org/250636544> <http://xmlns.com/SNR/0.1/mentions> "846540223127392256" .
<http://example.org/250636544> <http://xmlns.com/SNR/0.1/in_reply_to> "846540223127392256" .
<http://example.org/100308479> <http://xmlns.com/SNR/0.1/mentions> "748725392433770498" .
<http://example.org/100308479> <http://xmlns.com/SNR/0.1/in_retweet_to> "748725392433770498" .
<http://example.org/83171397> <http://xmlns.com/SNR/0.1/mentions> "275686563" .
<http://example.org/83664568> <http://xmlns.com/SNR/0.1/mentions> "18438867" .
<http://example.org/83664568> <http://xmlns.com/SNR/0.1/in_retweet_to> "18438867" .
<http://example.org/1540863097> <http://xmlns.com/SNR/0.1/mentions> "381024654" .
<http://example.org/1540863097> <http://xmlns.com/SNR/0.1/mentions> "2896841402" .
<http://example.org/1540863097> <http://xmlns.com/SNR/0.1/mentions> "2330543227" .
<http://example.org/1540863097> <http://xmlns.com/SNR/0.1/mentions> "3094514747" .
<http://example.org/3556802183> <http://xmlns.com/SNR/0.1/mentions> "132385468" .
```

Figure 11: Sample of N-Triple file format

5.4.3 Transferring RDF data to Social network server

For storing all RDF triple store data or N-Triple files we use the Fuseki server. In Figure 12, we provided a sample output of our RDF database that we generated. Our sample data has been created based on the users of Solar Eclipse 2017 data. We provide a URL to the Front End team so they can get the results for each of their queries. How we provided that interface and use Fuseki server to query has been explained in Section 6 and Section 7

5.4.4 Performance Evaluation

Table 8 shows the number of triples and file size of each of the Solar Eclipse 2017 and Las Vegas shooting core data files. Table 9 shows the time taken to convert each of the CSV data files to N-Triple (nt) format using a Java parser and approximate time taken to upload each file onto the server. We observe that the largest file in the Solar Eclipse data, which is around 373MB, takes almost 8 minutes to be uploaded and almost 3 hours to convert to NT format. However, the sparql query is faster than any other database query and it takes only 86 ms to fetch results for the query shown in Figure 42.

Table 8: File size of each NT file in solar eclipse and las vegas shooting events

File Name	Number of triples	File size
SolarEclipse1	1182	101KB
SolarEclipse2	18911	1.5MB
SolarEclipse3	30062	2.49MB
SolarEclipse4	17201	1.40MB
SolarEclipse5	262601	21.2MB
SolarEclipse6	651274	57.2MB
SolarEclipse7	1841718	150.2MB
SolarEclipse8	2090039	171MB
SolarEclipse9	4574528	373MB
VegasShooting1	33290	2.73MB
VegasShooting2	249508	20.8MB

Table 9: Approximate time to convert to NT file and upload in server

File Name	Time to convert to triple	Time to upload to server
SolarEclipse1	5s	2s
SolarEclipse2	30s	2s
SolarEclipse3	2min	1s
SolarEclipse4	50s	1s
SolarEclipse5	15min	4s
SolarEclipse6	45min	9s
SolarEclipse7	2hr	40s
SolarEclipse8	2hr	32s
SolarEclipse9	3hr	8min
VegasShooting1	3min	662ms
VegasShooting2	15min	2.721s

```
{
  "head": {
    "vars": [ "s" , "p" , "o" ]
  } ,
  "results": {
    "bindings": [
      {
        "s": { "type": "uri" , "value": "http://example.org/90379747:" } ,
        "p": { "type": "uri" , "value": "http://xmlns.com/SNR/0.1/followedBy" } ,
        "o": { "type": "literal" , "value": "627124461" }
      } ,
      {
        "s": { "type": "uri" , "value": "http://example.org/90379747:" } ,
        "p": { "type": "uri" , "value": "http://xmlns.com/SNR/0.1/followedBy" } ,
        "o": { "type": "literal" , "value": "128220551" }
      } ,
      {
        "s": { "type": "uri" , "value": "http://example.org/12868312:" } ,
        "p": { "type": "uri" , "value": "http://xmlns.com/SNR/0.1/followedBy" } ,
        "o": { "type": "literal" , "value": "201433608" }
      } ,
      {
        "s": { "type": "uri" , "value": "http://example.org/395810455:" } ,
        "p": { "type": "uri" , "value": "http://xmlns.com/SNR/0.1/followedBy" } ,
        "o": { "type": "literal" , "value": "162499430" }
      } ,
      {
        "s": { "type": "uri" , "value": "http://example.org/1593504410:" } ,
        "p": { "type": "uri" , "value": "http://xmlns.com/SNR/0.1/followedBy" } ,
        "o": { "type": "literal" , "value": "199339580" }
      } ,
      {
        "s": { "type": "uri" , "value": "http://example.org/908564819548053000:" } ,
        "p": { "type": "uri" , "value": "http://xmlns.com/SNR/0.1/followedBy" } ,
        "o": { "type": "literal" , "value": "3313983353" }
      } ,
      {
        "s": { "type": "uri" , "value": "http://example.org/703754312485367000:" } ,
        "p": { "type": "uri" , "value": "http://xmlns.com/SNR/0.1/followedBy" } ,
        "o": { "type": "literal" , "value": "386522370" }
      } ,
```


Figure 12: Sample JSON query in Fuseki based on the data we uploaded

6 User Manual

6.1 Transferring Data from JSON to HBase

For the purpose of transferring data from JSON to HBase we used Scala. For that purpose several dependencies had to be met. Here are the dependencies:

- JDK
- JSON4S

6.1.1 JDK

For running Scala code one of the first dependencies is the JDK version. JDK stands for Java Development Kit. Cloudera VM comes with JDK version 1.7. But the requirement for Scala is 1.8. This can be installed from here. First downlaod the file locally. We can't use the "wget" command to do that because it requires user agreement to the terms of condition.

6.1.2 JSON4S

JSON4S is a Spark library for parsing JSON data. There are approximately six libraries available in Scala. JSON4S wraps those and provides a uniform format. We used the jackson format from there. It has

a object like architecture to parse data. Each field can be modeled as a Scala object. A sample code of using this library is shown in Figure 13. Here first we check if a field exists in the data. Then we extract it. Then data is put into a Put object to store in HBase.

```
var user_location = ""
var user_statuses_count = "0"
if(((json_object \ "place") \ "country_code") != JNothing){
    place_country_code = ((json_object \ "place") \ "country_code").extract[String];
}
if(((json_object \ "user") \ "favourites_count") != JNothing){
    user_favourites_count = ((json_object \ "user") \ "favourites_count").extract[String];
}
if(((json_object \ "user") \ "followers_count") != JNothing){
    user_followers_count = ((json_object \ "user") \ "followers_count").extract[String];
}
if(((json_object \ "user") \ "friends_count") != JNothing){
    user_friends_count = ((json_object \ "user") \ "friends_count").extract[String];
}
if(((json_object \ "user") \ "lang") != JNothing){
    user_lang = ((json_object \ "user") \ "lang").extract[String];
}
if(((json_object \ "user") \ "location") != JNothing){
    user_location = ((json_object \ "user") \ "location").extract[String];
}
if(((json_object \ "user") \ "statuses_count") != JNothing){
    user_statuses_count = ((json_object \ "user") \ "statuses_count").extract[String];
}


val put = new Put(Bytes.toBytes(id))
put.add(Bytes.toBytes("metadata"),Bytes.toBytes("doc-type"), Bytes.toBytes("tweet"))
put.add(Bytes.toBytes("metadata"),Bytes.toBytes("collection-id"), Bytes.toBytes(collection_Id))
put.add(Bytes.toBytes("metadata"),Bytes.toBytes("collection-name"), Bytes.toBytes(collection_name))
put.add(Bytes.toBytes("metadata"),Bytes.toBytes("dummy-data"), Bytes.toBytes(false_))
put.add(Bytes.toBytes("tweet"),Bytes.toBytes("tweet-id"), Bytes.toBytes(id))
put.add(Bytes.toBytes("tweet"),Bytes.toBytes("like-count"), Bytes.toBytes(favorite_count))
put.add(Bytes.toBytes("tweet"),Bytes.toBytes("comment-count"), Bytes.toBytes(negative_))
put.add(Bytes.toBytes("tweet"),Bytes.toBytes("retweet-count"), Bytes.toBytes(retweet_count))
```

Figure 13: Sample code for transferring JSON value in HBase

6.2 Cleaning Data

For the purpose of cleaning data several dependencies have to be met in the system. These dependencies are discussed below:

6.2.1 Stanford CoreNLP

Stanford CoreNLP provides several APIs to extract named entities, parts of speech, and other features like from the sentence tokenizer and lemmatizer. The list of features is in Fig. 14. The CoreNLP package that we used can be found here.

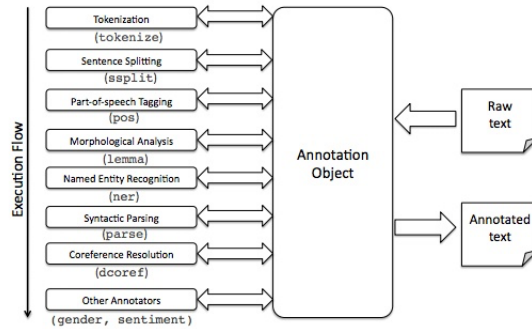


Figure 14: Stanford CoreNLP features

Following is a list of types of named entities that are returned by the Stanford CoreNLP package.

- Location
- Person
- Organization
- Money
- Percent
- Date
- Time

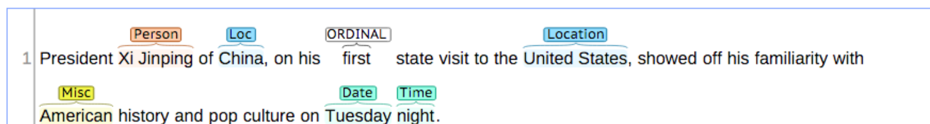
For the POS tags CoreNLP provides the Penn Treebank tags which can be found here. Figure 15 shows sample NER and POS tags found by the CoreNLP.

After downloading the CoreNLP the Spark driver for it has to be downloaded from here.

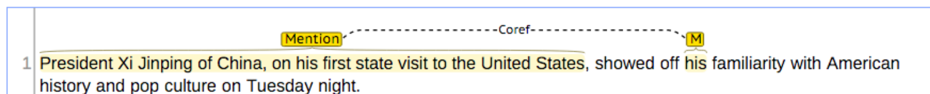
6.2.2 Building Matthew's framework

Several changes were made to Matthew's framework for the purpose of extracting mentions and retweets. So it has to be built. To build first Simple Build Tool (sbt) has to be downloaded from here. Then using the command "sbt package" it has to be built. Finally we can run the code for cleaning the HBase data using the scala script. A sample portion of the code is shown in Figure 16. Here first data is collected from HBase. Then it is sanitized. Then the collection is transformed into DataFrame. Finally CoreNLP annotators are used to extract tags.

Named Entity Recognition:



Coreference:



Basic Dependencies:

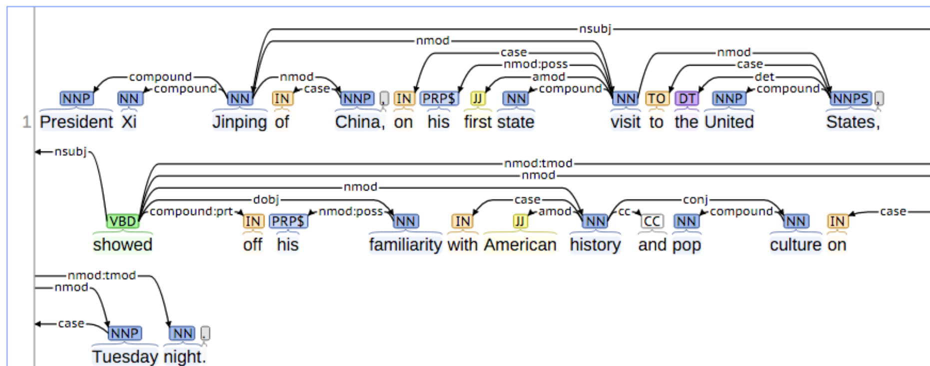


Figure 15: Sample annotations by CoreNLP

```
val tweetcollectionfactory = new TweetCollectionFactory(sc,sqlContext)
val tweetcollection = tweetcollectionfactory.createFromHBase(collectionId,hbaseconfig)
tweetcollection.applyFunction(sanitize2)
val input = tweetcollection.getCollection().map(tweet => (tweet.id,tweet.text,tweet.tokens.mkString(" ")+".",tweet.
    mentions.mkString(","),tweet.hashtags.mkString(","),tweet.urls.mkString(","),tweet.isRetweet)).toDF("id","text","
    cleaned","mentions","hashtags","urls","rt")

val output = input.select('text.as('sen),'cleaned,'id,'hashtags,'mentions,'urls,'rt).select('rt,'hashtags,'mentions,'urls,
    'id,'sen,lemma('cleaned).as('cleandata), tokenize('sen).as('words), ner('sen).as('nerTags), pos('sen).as('posTags))

output.show(numRows = 5, truncate = false)
```

Figure 16: Sample code for cleaning data in HBase

6.3 Running Matthew's Framework

Matthew's framework was used by our group to fetch the user mentions results from the dataset. The framework was very useful as it could process tons of large datasets and fetch the results in a very reduced runtime. However, one of the challenge that we faced was running Matthew's framework. Due to unavailability of proper documentation, it took us till the middle of the semester to run Matthew's framework. Below are the steps that we followed to get the framework running:

- Connect to Hadoop cluster: ssh cs5604f17_cmt@hadoop.dlib.vt.edu then provide your password.
- Connect to node00.

- Transfer all the source code file of Matthew's framework to the Hadoop cluster using the following command in the Hadoop cluster:
  scp -r /local_machine_username@ip_address:/path/to/sourcecode/matthewframework ~/.
- Check if the dataset is already loaded in the Hadoop cluster: hadoop fs -ls
- If not, then load the dataset into the Hadoop cluster: hadoop fs -put *.csv
- Change the file path in your desired example Scala file: vi examples/MassExtractionExample.scala
- Start the Spark shell using: 'spark-shell –master local –jars ./SOURCE-CODE/dlrl-lib-latest.jar -i ./SOURCE-CODE/examples/MassExtractionExample.scala'
- All the results will be stored in the "results" directory of the source code.

While doing the transfer we need to keep the remote login turned on in our computer as shown in Figure 17.

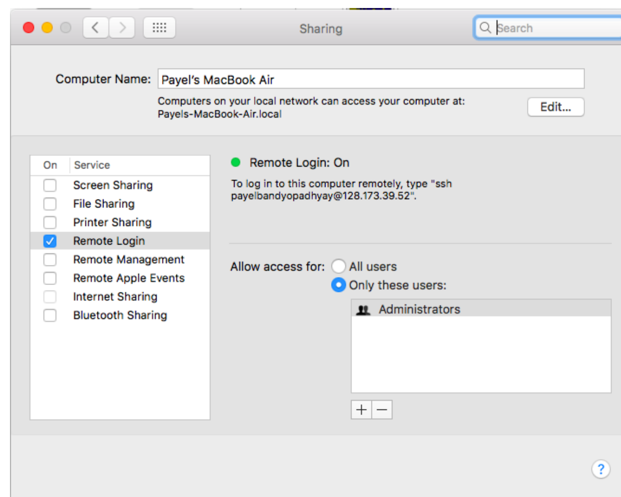


Figure 17: Sharing turned on in local server.

The outputs are stored in the *results* directory of Matthew's framework folder. Figure 18 shows an example result of *mentions*.

6.4 Social Network

6.4.1 Fetching list of followers and friends

In the Twitter API, each tweet will be stored in the form of an object with a set of attributes. These attributes can be collected by sending the request to the API. Fortunately, Python developers created several useful libraries that enable us to fetch the information from the API. The most general library is the Twitter library, which is a Python wrapper around the Twitter API. We benefit from this powerful wrapper to fetch the useful information related to the users and tweets to create our social network. The first step to use the

```
829009583985324034      @cbmcanada @
829007550540558337      @anthropolitiks
829022751667130369      @nchousingbuilds
829012624599875584      @iandfox28 @wtgsfox28
829011611117637632      @iandfox28
829011610069061633      @robintlewis @worldcares
827017861721911298      @ciara
829100955773894656      @movetheworld
829007357254434816
828057444353859584      @kbriwashdc
827542311601106947      @wvhaiti @aquatabs
828108287719649280      @ordinarycasss
827180644669210624      @raisinghaiti
827665087720734725      @fema
829134715047313408      @maritaunicef
827179050317799424      @ciara
828427233144557568      @sccer4ever
827003947692929024      @insurancepost
```

Figure 18: Sample output from Matthew's framework.

Python Twitter is to creat the OAuth keys. Therefore, we registered an app with Twitter in order to use the library. Full details of the registration can be found in the Twitter developer page (https: dev.twitter.com="" oauth="" overview="").

6.4.2 Creation of RDF data

We convert each of the data we collect from a CSV file and Twitter API into a triple or RDF data using a simple script file written in Java. In our RDF data, we convert each subject and predicate to a URI, and each object to a string literal. Every URI of a subject starts with "http://example.org/" and every URI of a predicate starts with "http://xmlns.com/SNR/0.1/". In the triple format each subject and object is a user ID and each predicate is a relationship. For example, if we want to create a triple of User ID 2351245436 mentions a user 90379747 in a tweet, the converted RDF data is "http://example.org/2351245436" "http://xmlns.com/SNR/0.1/mentions" "90379747". Table 10 shows the URI we used for each relationship of our social network.

Table 10: URI maintained for creating each relationship

Relationship	**Description**	**URI**
mentions	A user mentions another user in a tweet	http://xmlns.com/SNR/0.1/mentions
replying to	A user replied to another user in a tweet	http://xmlns.com/SNR/0.1/in_reply_to
retweeted	A user retweeted another user's tweet	http://xmlns.com/SNR/0.1/retweeted_to
follower	A user followed by another user	http://xmlns.com/SNR/0.1/followedBy
friend	A user is friend of another user	http://xmlns.com/SNR/0.1/friendOf

6.4.3 SPARQL Query

SPARQL is the query language for RDF data. The Front End team needs to use this query language from the Fuseki server to visualize the relationships among users. For assisting the Front End team we show examples of some SPARQL queries in Table 11.

Table 11: SPARQL Query

Query Description	SPARQL
to fetch all types of relationships among two users	prefix sub: http://example.org/ prefix pred: http://xmlns.com/SNR/0.1/ SELECT ?p WHERE sub:userID1 ?p “userID2”
to select all data	prefix sub: http://example.org/ prefix pred: http://xmlns.com/SNR/0.1/ SELECT ?s ?p ?o WHERE ?s ?p ?o

6.4.4 Transferring RDF data to server

For transferring RDF N-triple data to the Fuseki server we used the account provided by GTA. Once we installed the Fuseki server as a standalone server in the VM using the instructions given in the Jena website[1]. Then we start the Fuseki server in port 3030 using the VM account. We create a dataset path named /Follower where we added all the RDF N-triple format files. For the Front End team the interface they need to use should be http://cs.mule.dlib.vt.edu:3030/Follower/query?query=<some_query>&wt=json&json.wrf=my_callback. “some_query” is the query that the Front End team needs to query from the server.

[1] https://jena.apache.org/documentation/serving_data/#running-a-fuseki-server

7 Developer Manual

7.1 Setting up Virtual Machine

One of the first steps for this project is to set up a virtual machine to work on. There in a Cloudera kvm image available for that purpose. Here is the configuration of the machine we requested from Dr. Fox:

- No. of CPU Cores: 16
- HDD: 1024GB
- RAM: 32GB
- OS: Ubuntu 16.04LTS

For future purposes our recommendation is to ask for at least 128 Core PC with 2TB HDD and 128GB RAM. 128 Core PC is recommended because of the processing time it takes to find NER and POS tags. The benchmark for that can be seen in section 5.3.3. Although the default Spark setting for RAM is 512MB, it is not enough for loading the CoreNLP libraries. That is why the amount of RAM required for each CPU is large.

After getting access to the server we first need to install utilities for checking if the server does support virtualization. For this purpose see the commands in Fig. 19. We are installing the utilities for checking that capability as well as networking utility necessary for the KVM virtual machine. After that we need to install a virtual machine manager. For that we need to set up a different user group. Then we have to add permission to the user for installing virtual machine. This is shown in Figure 20.

```
fox@cmtweet:~$ sudo apt-get install qemu-kvm libvirt-bin virtinst bridge-utils cpu-checker
Reading package lists... Done
Building dependency tree
Reading state information... Done
bridge-utils is already the newest version (1.5-9ubuntu1).
cpu-checker is already the newest version (0.7-0ubuntu7).
libvirt-bin is already the newest version (1.3.1-1ubuntu10.14).
qemu-kvm is already the newest version (1:2.5+dfsg-5ubuntu10.16).
virtinst is already the newest version (1:1.3.2-3ubuntu1.16.04.4).
The following packages were automatically installed and are no longer required:
  libgsoap8 libvncserver1 linux-headers-4.4.0-21 linux-headers-4.4.0-21-generic linux-image-4.4.0-21-generic
  linux-image-extra-4.4.0-21-generic virtualbox-dkms
Use 'sudo apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 27 not upgraded.
fox@cmtweet:~$ kvm-ok
INFO: /dev/kvm exists
KVM acceleration can be used
```

Figure 19: Check Virtualization Support in Server

```
fox@cmtweet:~$ sudo adduser `id -un` kvm
The user `fox' is already a member of `kvm'.
fox@cmtweet:~$ sudo adduser `id -un` libvirtd
The user `fox' is already a member of `libvirtd'.
fox@cmtweet:~$ groups
fox adm cdrom sudo dip plugdev lxd lpadmin sambashare kvm libvirtd
fox@cmtweet:~$ modprobe -a kvm
fox@cmtweet:~$ sudo apt-get install virt-manager
Reading package lists... Done
Building dependency tree
Reading state information... Done
virt-manager is already the newest version (1:1.3.2-3ubuntu1.16.04.4).
The following packages were automatically installed and are no longer required:
  libgsoap8 libvncserver1 linux-headers-4.4.0-21 linux-headers-4.4.0-21-generic linux-image-4.4.0-21-generic
  linux-image-extra-4.4.0-21-generic virtualbox-dkms
Use 'sudo apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 27 not upgraded.
```

Figure 20: Add user and install virtual machine manager

Now we are ready to set up the the virtual machine. First we need to start networking for KVM. The commands to do that are shown in Figure 21.

```
fox@cmtweet:~$ sudo virsh net-list --all
 Name                 State      Autostart     Persistent
----------------------------------------------------------
 default              active     no            yes

fox@cmtweet:~$ sudo virsh net-start default
```

Figure 21: Start Networking for KVM

Then we need to download the KVM image from the server. Use the command shown in Figure 22 to do that. It will download the file in the current directory. If a second argument is given it will download the file and rename to that argument.

```
fox@cmtweet:~$ wget http://hadoop.dlib.vt.edu/cloudera_vm/cloudera-quickstart-vm-5.12.0-0-kvm.zip
```

Figure 22: Download KVM image

As the file is zipped it needs to be unzipped somewhere using the command shown in Figure 23. The command below unzips a file in the current directory. As a second argument, the location of unzipping can be given.

```
fox@cmtweet:~$ unzip cloudera-quickstart-vm-5.12.0-0-kvm.zip
```

Figure 23: Unzip KVM image

Now we can install the KVM image using the downloaded file shown in Figure 24. The arguments provided here are the image file to use to create a new virtual machine along with number of CPUs and RAM size. We have set the number of cores to 12 from the 16 available and 24GB RAM from 32GB available. We have also set the vnc argument for remote graphical access to the VM.

```
fox@cmtweet:~$ sudo virt-install --connect qemu:///system -n cloudera_new --vcpus=12 --disk path=/home/fox/cloudera_image/cloudera-quickstart-vm-5.12.0-0-kvm/cloudera-quickstart-vm-5.12.0-0-kvm.qcow2,device=disk,bus=virtio,format=qcow2 --ram 24576 --vnc --noautoconsole --import
```

Figure 24: Install KVM image

7.2 Adding HDD space to VM

The KVM image that is provided has 64GB physical and logical space available. We were given a file of size 50GB for testing which requires more space in VM. So now we need to add space to the virtual machine. First we need to expand available space to the VM using the command shown in Figure 25.

```
fox@cmtweet:~$ sudo qemu-img resize cloudera-quickstart-vm-5.12.0-0-kvm.qcow2 +256G
```

Figure 25: Add available space to VM

The following commands have to be run inside the virtual machine. Now that we have added the available space to the virtual machine it is still not available to the operating system. To do that we have to first run the "fdisk" command as shown in Figure 26.

```
sudo fdisk /dev/vda
```

Figure 26: Run fdisk to change partition

Inside the fdisk terminal we can print the current partition using the p command as shown in Figure 27. As shown here we have three partitions. The first one is the primary partition for boot. The 2nd and the 3rd one's are the partition used by the user. Note that sometimes the disk space is shown as cylinders instead of sectors which creates problem if multiple partitions start at the same cylinder. Try using "c" command to change the unit to sectors.

```
Command (m for help): p

Disk /dev/vda: 48.3 GB, 48318382080 bytes
16 heads, 63 sectors/track, 93622 cylinders, total 94371840 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00079e72

   Device Boot      Start         End      Blocks   Id  System
/dev/vda1   *        2048      391167      194560   83  Linux
/dev/vda2          393214    41940991    20773889    5  Extended
/dev/vda5          393216    41940991    20773888   8e  Linux LVM
```

Figure 27: Print current partition in VM

Now we are going to delete the last partition using the command shown in Figure 28.

```
Command (m for help): d
Partition number (1-4): 3
Partition 3 is deleted
```

Figure 28: Delete the last current partition in VM

After deleting the partition we can expand the last one. To do that, use the commands shown in Figure 29. Currently these changes hasn't been updated inside the partition table. To do that we have to write that using "w" command here.

```
Command (m for help): n
Partition type:
   p   primary (1 primary, 0 extended, 3 free)
   e   extended
Select (default p): e
Partition number (1-4, default 2): 2
First sector (391168-94371839, default 391168): <ENTER>
Using default value 391168
Last sector, +sectors or +size{K,M,G} (391168-94371839, default 94371839): <ENTER>
Using default value 94371839
```

Figure 29: Create a new partition in VM

Now we have to reboot the machine. After reboot we can add the space to the physical and logical volume. To add the available space to the physical volume use the command in Figure 30. Then try the command in Figure 31 to add the space to the logical volume. Finally we have to resize the file system to add the additional space. Use the "resize2fs" command for that purpose. This whole process is described very well in here.

```
sudo pvresize /dev/vda5
```

Figure 30: Physical volume resize

```
sudo lvextend -L +15G /dev/vmit/home
```

Figure 31: Logical volume resize

7.3 GETAR Server Architecture

7.3.1 Hardware architecture

- Number of Nodes: 20 + 1(SOLR)
- CPU Cores : 88 (1 Intel Xeon + 20 Intel Core i5 (Haswell))
- RAM : 704 GB
- HDD : 154.3 TB

7.3.2 Software Architecture

In the server Cloudera Hadoop 5.12 is installed. For the sake of necessity we are going to describe the specific part about CDH 5.12 instead of everything. For a detailed review see this.

7.3.3 Tweet Collection Module

The server uses three software routines for Twitter data collection. Here is the list:

- yTK: yourTwapperKeeper is a tool for collecting data from the Twitter streaming API and search API. Initially it was used a lot. Due to changes in Twitter's terms of services this is used much less.
- dmi-tcat: Digital Methods Initiative Twitter Capture and Analysis Toolset is another tool for collecting twiiter data. It has a nice way to format the output data.
- SFM: Social Feed Manager is the main tool currently being used for collecting Twitter data. It uses the Twitter streaming API.

Currently the Twitter search API is being used by all the modules. They are mainly using hashtags to collect data about different events.

7.3.4 Apache Hadoop

Apache Hadoop is a map-reduce based framework used for parallelizing large tasks using multiple nodes. The distributed computing style used by Hadoop makes it easier to use for tasks that can be divided into sub-tasks and run by multiple machines. Apache Hadoop provides a system along with four main components:

- Hadoop Commons: Common utility library used by all other sub-systems
- HDFS: Hadoop Distributed File System provides a high throughput file system along with system recovery feature
- YARN: Hadoop provides a manager for distributing resources and scheduling tasks named YARN.
- Map-Reduce: This is the system used by YARN to parallelize tasks.

Here we describe the Map-Reduce framework by Hadoop. Figure 32 shows a detailed view about how Hadoop distributes tasks. When a job is submitted it is first split into two tasks named map and reduce. Then those are distributed to the available nodes. For example: a simple task of counting distinct strings in a text file could be divided into two tasks like:

- Map: Read and split string
- Reduce: Count number of distinct string

Then half of the nodes could be given the map task and the other half the reduce task. Hadoop's architecture provides several benefit:

- Scalability: Hadoop provides a scalable framework where new nodes could be added very easily.
- Parallelization: All tasks performed in Hadoop are parallelized which provides ample speed for the large tasks.
- Resilience: Hadoop provides a resilient system where if one of the nodes becomes unavailable due to a crash its tasks are divided to other nodes quite easily without any real loss of performance.
- Simple Model: Hadoop's architecture of programming is very simple which makes it easier to be adapted for any type of tasks.

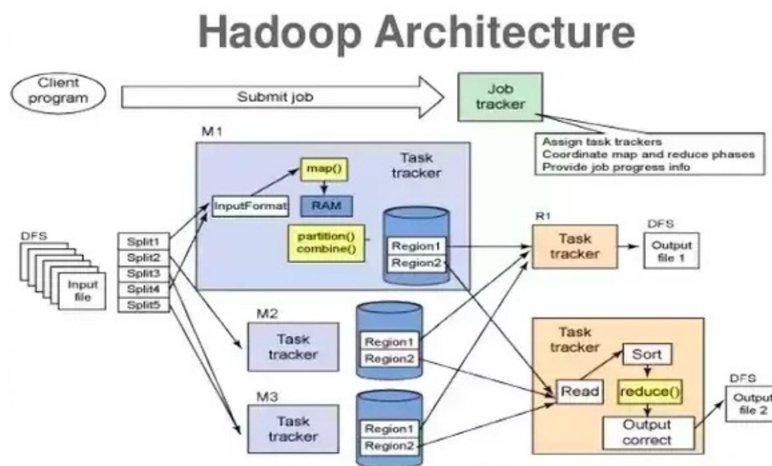


Figure 32: Map-Reduce task flow in Hadoop

Installation: For detailed instruction on installing Hadoop see this.

7.3.5 HDFS

HDFS is the distributed file system provided by Hadoop with several capabilities. The architecture of HDFS can be simplified as in Figure 33. HDFS has mainly two types of node. Namenode contains the metadata as well as the information about the distribution of all the data in different nodes. Datanode contains data in Hadoop. To provide crash recovery, data is replicated across nodes. As this is used mainly for large data sets, the block size in HDFS is by default 64MB. For the Cloudera version it is 128MB by default. Here are several benefits of HDFS:

- Reliability: HDFS provides reliability from data loss by replication across nodes.
- Low Network Bandwidth: Hadoop manages data in a way that uses the lowest amount of network bandwidth.

- Integration to Hadoop Map-Reduce: As HDFS is integrated in the Hadoop system it works best with any map-reduce task. The data is prefetched to every node so that there is minimal wait time between processing.

For our project we used HDFS for initially storing the JSON files. Then those were processed using Spark. That way we utilized as much parallelism as possible.

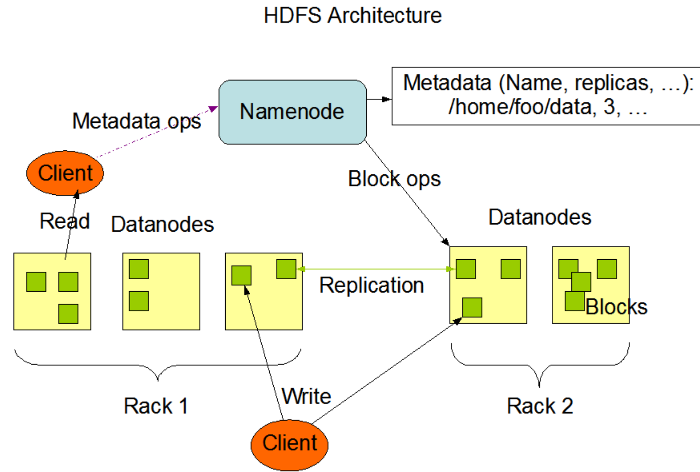


Figure 33: HDFS architecture

7.3.6 HBase

HBase is Apache's NoSQL database that uses the HDFS. HBase tables are defined by column families. Here a column family is a group of columns. HBase tables are stored sequentially by row keys. So partitions happen by range of row key. These partitions are called "Region". Region servers are the nodes who store those regions. Regions are stored in datanode in HDFS. There is another type of node named HMaster. It contains all the metadata and assignment of regions in regions and provides an interface for all types of commands. Figure 34 provides a detailed view of the HBase architecture. For our project we stored tweets in HBase. HBase also has similar benefits like HDFS. Moreover it uses a block cache and bloom filter for query optimization.

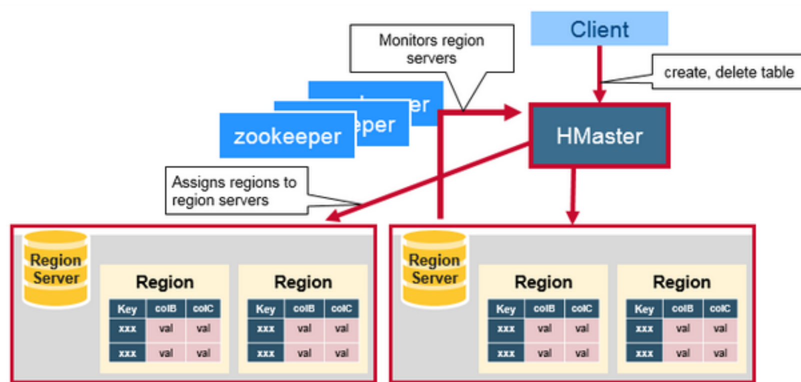


Figure 34: HBase architecture

7.3.7 Spark

Spark is another Map-Reduce framework by the Apache Foundation that is faster than Hadoop Map-Reduce. It uses in-memory computation to make it faster. If the data can be stored in memory for processing then Spark is 100 times faster than Hadoop. Spark is easily integrated with the Hadoop system so it can use HDFS, YARN, and HBase for processing. Spark uses similar architecture like Hadoop Map-Reduce as we can see in the figure 35 below. It has a master node that manages tasks. The slave nodes have some executors that complete the task. Spark as a standalone application can't do much. But when integrated with the Hadoop system it works very rapidly. Beside the Hadoop context Spark provides some machine learning library routines that are very optimized. Spark also provides some data structures like DataFrame to make it easier for programming.

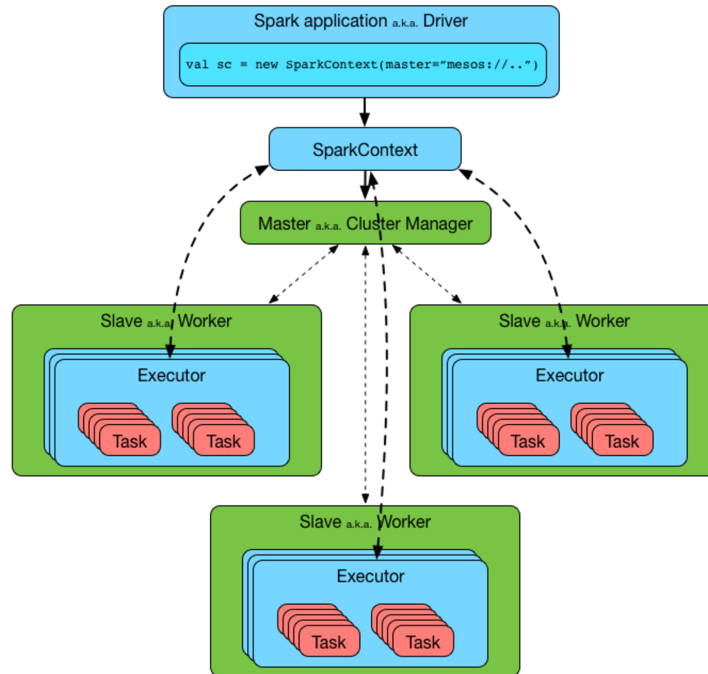


Figure 35: Spark architecture

7.4 Social Network

As a triple-store database, we have used th Apache Jena Fuseki server. The overall process to create a social network has been performed in three steps:

- Conversion of JSON data file to CSV format for necessary schemas [We used user mentions, in_reply_to, in_retweet_to, userId, user statuses count, user followers count, user friends count]
- Fetching Twitter followers and friends using the Twitter API
- Conversion of CSV file format to RDF format [i.e., N-Triple(.nt) format], to put those data into the triple store on the Fuseki server
- Installation of Fuseki server in Virtual Machine, upload data in server, and make server a standalone server so the Front End team can access that anytime

7.4.1 List of Softwares for social network

In Table 12 we provide a list of software routines along with the version we used to build the social network.

Table 12: List of software routines and their version

Software	Version	Task
CentOS	6.8	To install Fuseki server
Java OpenJDK	1.8.0	To build RDF N-Triple model
Python	2.7	To fetch followers and friends using Twitter API
Jq	NA	To convert JSON to CSV file
MacOS	10.12.1	Above json to csv file conversion was done in MacOS Sierra

7.4.2 Conversion of JSON to CSV file

One of the main task was to convert the given JSON datafile to CSV file format. Now, it was not so simple as all the fields were not similar in the JSON data and the fields contained array elements. So retrieving all the values accordingly was difficult. Figure 36 shows an example of the given JSON dataset.

```
{
  "contributors":null,
  "truncated":false,
  "is_quote_status":true,
  "in_reply_to_status_id":null,
  "id":898284179707777024,
  "favorite_count":8,
  "full_text":"Wow! This is insane! #OREclipse @Ktvz
  https://t.co/KSDod7WPsr",
  "entities":{
    "symbols":[
    ],
    "user_mentions":[
      {
        "id":25451167,
        "indices":[
          32,
          37
        ],
        "id_str":"25451167",
        "screen_name":"KTVZ",
        "name":"KTVZ NewsChannel 21"
      }
    ],
    "hashtags":[
      {
        "indices":[
          21,
          31
        ],
        "text":"OREclipse"
      }
    ],
    "urls":[
      {
        "url":"https://t.co/KSDod7WPsr",
        "indices":[
          38,
          61
        ],
        "expanded_url":"https://twitter.com/orstatepolic
        e/status/898283672536719360",
```


Figure 36: Part of JSON data for file oreclipse.json

So, to convert the JSON files we followed the following steps:

- Install jq

- cat Eclipse.json | jq -r . | [.user.id_str, .retweeted_status.id_str, .in_reply_to_user_id, .entities.user_mentions[].id] | @csv > ./Eclipse/Eclipse.csv

One of the main challenges arose when there were more than one array item. For those cases, the files were processed separately and then re–joined later. Figure 37 shows an example of the sample CSV file:

id	favourite_ count	full_text	user_id	retweeted_status_id	in_reply_to_ user_id	entities_user_mentions
888201064817860613	5	There's going to be a	103167711	889882842242707456	15102849	713741422000807937
19199743	2	I gotta buy some solar eclipse	264792278	889941327202455553	125485258	2470058834
2762027475	0	Cellphone service could be spotty	466665274	889874789611048960	15102849	124197346
224233529	0	Anyone else notice how	101144034	889898800411688960	11348282	11348282

Figure 37: Part of equivalent CSV data for file oreclipse.json

We repeated the same process for all the given files. The time for conversion took a while for various files, depending on the size of each file.

7.4.3 Fetching Twitter followers and friends with Twitter API

To fetch the followers and friends list of the user, there are several libraries for the Twitter API. We chose the Python Twitter library for our work. We modified an existing script using the Twitter library in Python, hosted in Github [17], to access the Twitter API for fetching the list of followers and friends in the form of screen name or user ID. One limitation of Twitter API for Get endpoints is the rate limit. For "GET followers/list" and "GET friends/list" the limit is 15 "Requests / window (app auth)". In our code we have included a block to avoid this rate limit. Thus, running the code for the large dataset can be time-intensive (almost 45 seconds to fetch the followers list for each user). The highlighted part in Figure 38 is the script for the rate limit issue. The script can be found in our files under *Social Network/Fetch-followers.py* .

```python
import twitter, time, json


# Load config from JSON
with open('config.json') as data_file:
    json = json.load(data_file)

# Connect to Twitter API
api = twitter.Api(consumer_key=json['twitter']['consumer_key'],
                      consumer_secret=json['twitter']['consumer_secret'],
                      access_token_key=json['twitter']['access_token_key'],
                      access_token_secret=json['twitter']['access_token_secret'])
# Print payload to verify Twitter access is working
print (api.VerifyCredentials())

# Write list of followers to a followers.txt file
def writeFollowersToFile(followers, username):
    # Write name of selected user
    with open("followers.txt", "a") as myfile:
        myfile.write('Followers for {0}:\n\n'.format(username))
    # Write list of followers
    for friend in followers:
        with open("followers.txt", "a") as myfile:
            myfile.write('{0}\n'.format(friend))
    print ('Done writing to file.')

# Get all followers from users in array
friends = []
for name in json['accounts']:
    #sec = api.GetSleepTime('/followers/list') + 2 # Wait 2 seconds more to make sure rate-limit is avoided
    #print ('Will wait {0} sec to avoid rate-limit'.format(sec))
    time.sleep(45)
    print ('Fetching followers for {0}...'.format(name))
    # Uncomment the next line to instead fetch from the /followers/list API (takes much longer)
    #friends = api.GetFollowers(screen_name=name, skip_status=True, count=200, include_user_entities=False)
    try:
        friends = api.GetFollowerIDs(user_id=name)
    except Exception,ex:
        print ex
        continue
    print ('Writing the {0} followers of {1} to file...'.format(len(friends), name))
    writeFollowersToFile(friends, name)
```

Figure 38: Twitter library code for fetching the followers list

7.4.4 Conversion of CSV to N-Triple file

```java
import org.apache.jena.rdf.model.Model;
import org.apache.jena.rdf.model.ModelFactory;
import org.apache.jena.rdf.model.Property;
import org.apache.jena.rdf.model.RDFNode;
import org.apache.jena.rdf.model.Resource;
import org.apache.jena.rdf.model.ResourceFactory;
```

Figure 39: Packages to import in JAVA to create N-Triple data files

To convert CSV data files to N-Triple (.nt) file format we write our own source code in Java. We download Apache Jena jar files from the Internet and add the jar file to build the path of our project in Eclipse. To build the model in (.nt) format we need to import some packages given in Figure 39. Here, we import first two packages to build a model in N-Triple (.nt). The other packages are to store subject, predicate, and object in different data format. We store each subject as a Resource, each predicate as a Property, and each object as an RDFNode literal. Resource, Property, RDFNode – they are all different data formats of N-Triple. To create Resource and Property objects they need a URI. So, we create a dummy URI for each subject and predicate. The URI of each subject is "http://example.org/20445571", where 204455671 represents a userID. Similarly, since our predicate or relations are constant, i.e., we are storing only predefined relationships [mentions, replies, retweets] we create each predicate as Property. Each Property also needs to be a URI. Thus, we created dummy URIs "http://xmlns.com/SNR/0.1/relations", where relations ={mentions, in_reply_to, in_retweet_to}. Next, we create a CSV parser in Java to fetch each subject (userID), each predicate (relation), and object (userID) of corresponding subject and predicate. The first column of the CSV file represents a userID. The second column represents in_retweet_to, which is userID of a user whose tweet has been re-tweeted in this tweet. The third column represents in_reply_to_userId which means the tweet is a reply and replied to the user in the respective column. The fourth column represents userID of entities_user_mentions. So, for each userID, there are a maximum of three triples produced. Not all the columns contain userID; many of them contain null values. We used StringTokenizer to parse each of the columns of the CSV file. For each predicate in a row, if a column is not null, we create a triple object at first for the corresponding subject and object and add that triple to the model using model.add(subject, predicate, object). Here the subject is a URI in Resource format, the predicate is a URI in Property format, and object is a String in RDFNode format. Finally, we dump the whole model in a file using model.write(out,"N-Triple"). Here the first parameter is an output buffer file opened in Java and the second parameter is the file format of the RDF, i.e., N-Triple for us. The file has been saved as (.nt) file. Our code to convert CSV to NT file can be found in Social Network/triple-store

Triple-store Fuseki server We store all triples of N-Triple file format in the Fuseki server provided by Apache Jena Fuseki. Our main goal was to put all the triple data in the server and provide an interface to the Front End team so they can fetch using a query for their visualization. To fulfill our goal we had to install the Fuseki server in the Virtual Machine (VM) provided by the GTA. The operating system of the VM is based on Centos and the memory limit of VM was 8GB at first. We download the Apache-Jena-Fuseki server from the Apache Jena official site.

Installation and running Fuseki server The installation process was not very hard. After sending the file we downloaded to the VM through scp , we unzip the file in the home folder and go inside the Fuseki folder. To install Fuseki in Centos we at first need to install Tomcat and have Java installed in the VM. Then we installed Fuseki following the steps from the installation site [2]. To start the server we need to run fuseki-server.bat file by using command ./fuseki-server. The server is going to start at host 8080. However, to create an empty in-memory dataset we use script ./fuseki-server –update –mem /eclipse, since we created a dataset named eclipse. To upload each eclipse N-triple files in the server we used the script command ./s-put http://mule.dlib.vt.edu:3030/eclipse/data path/data.nt.

Increase memory size of server After installation, while uploading NT files, we faced some problems. Since we had to upload triples from 6M solar eclipse tweets and .18M vegas shooting tweets, it was show-

[2]https://confluence.si.edu/display/SIDKB/Install+and+Configure+Jena-Fuseki+with+Fedora+Repository

ing a Java heap space error and crashed every time we tried to upload a huge data file and issue a query. So, we opened the fuseki-server.bat file in edit mode and edited the memory size to 4GB to serve our purpose. To do this, we opened the fuseki-server file and edit JVM_ARGS=${JVM_ARGS:–xmx4096M} inside the fuseki-sever. [Shown in Figure 40]

```
if [ -z "$JAVA" ]
then
    (
    echo "Cannot find a Java JDK."
    echo "Please set either set JAVA or JAVA_HOME and put java (>=1.8) in your PATH."
    ) 1>&2
  exit 1
fi

JVM_ARGS=${JVM_ARGS:--Xmx4096M}

exec $JAVA $JVM_ARGS -jar "$JAR" "$@"
```

Figure 40: Changing memory size to avoid java heap space error

Changing memory type of dataset If we keep the memory data set as in-memory, with every restart of the server it loses all data except the dataset name. So, we make the dataset in-persistent instead of keeping it as an in-memory dataset. In-persistent memory means the data will not be lost when the server restarts. Figure 41 shows how to change memory type when creating a new dataset.

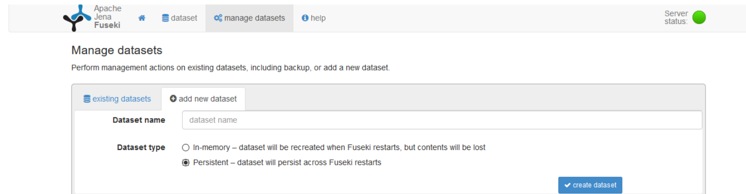


Figure 41: Changing memory type of data set on Fuseki server

7.4.5 SPARQL Query

SPARQL is the query language of the RDF triple-store database. Figure 42 shows a sample query of some triples [see Figure 11] we uploaded on the server. Here, prefix sub contains the common URI of the subject and prefix pred contains the common URI of the predicate. In our sample query, we want to fetch all users or objects who are connected to subject or userId '2351245436' by relationship mentions, in_reply_to, and in_retweet_to relations. ?o is the variable of the object and every variable we want to fetch in the query should start with ?.

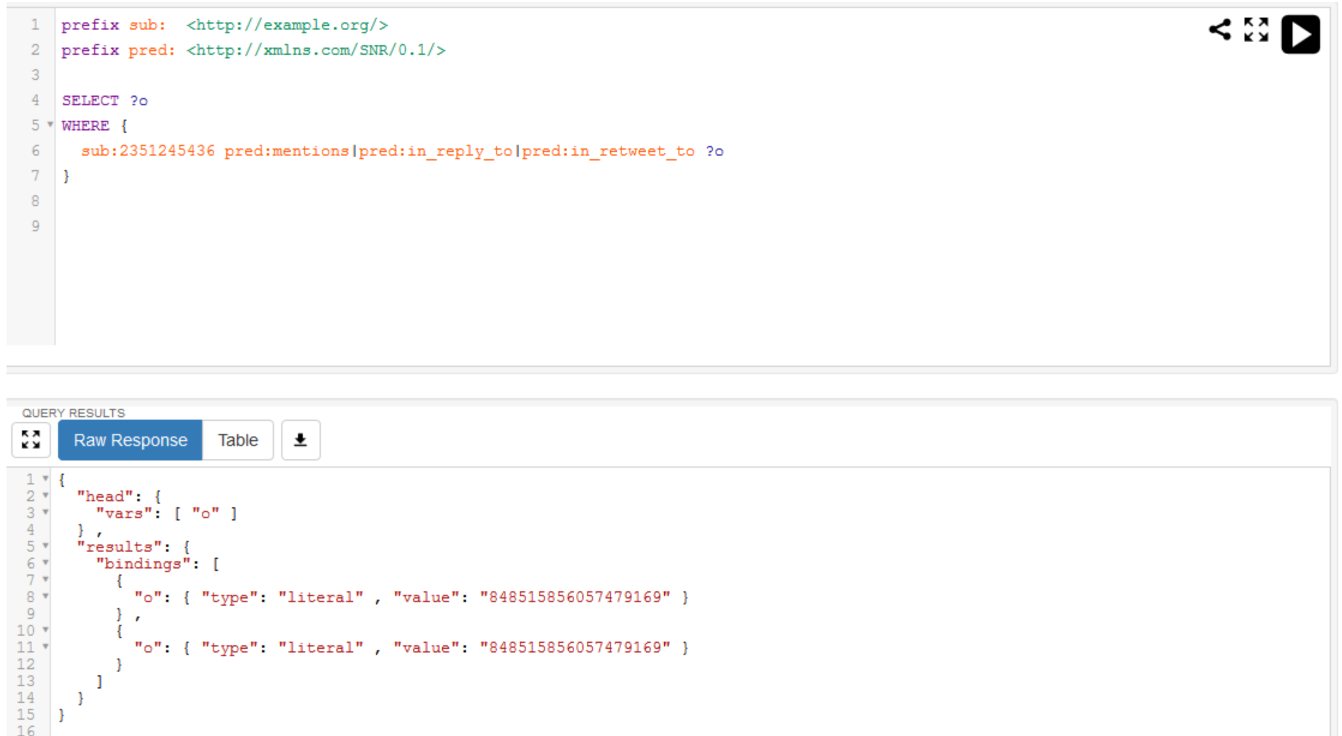


Figure 42: Example of a SPARQL query based on our dataset

7.4.6 Interface to Front End Team

We already mentioned in the previous section how the Front End team can access the data set or get the query results from the Fuseki server. Figure 43 shows the different data set the Front End team can use. We provided three different datasets for different events.
eclipse contains all triples and relationships of solar eclipse 2017 tweets.
shooting contains all triples or relationships of Las Vegas Shooting events. As per requested by the Front End Team we also create a
getar dataset which contains relationships of both events.

To make the server access anytime by the Front End team we needed to run Fuseki in the background, so the server does not need to restart every time they want to fetch for a query. To do this, before starting the fuseki-server we type screen and then run fuseki-server by .
fuseki-server and exit safely. Every time we need to start the server for uploading or creating new data we use screen list and scree -r sessionID, to start the server from before. sessionID is the ID of the session when we started the server and we can retrieve it from the screen list.

Before providing the interface to Front End Team, we had to change the access control of the Fuseki server so that anonymous users cannot change data we created. For this, we had to change shiro.ini controlled by configuration file which is located at '$FUSEKI_BASE/shiro.ini. We add user name and password inside

the shiro.ini file as shown in Figure 44[3].

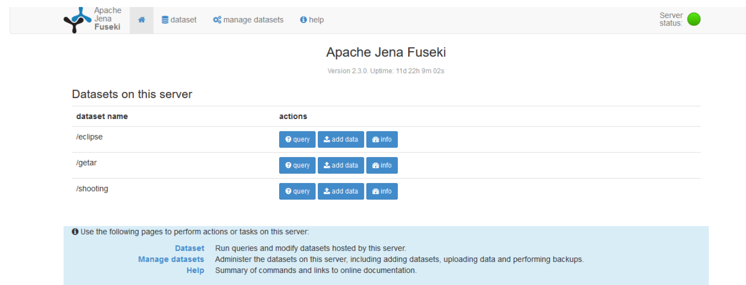


Figure 43: A screen shot of dataset we created in Fuseki

```
[users]
admin=pw

[urls]
## Control functions open to anyone
/$/status = anon
/$/ping   = anon
/$/** = authcBasic,user[admin]
# Everything else
/**=anon
```

Figure 44: A screen shot of dataset we created in Fuseki

8 Future Work

For the tweet cleaning task, several avenue of improvement is possible. For the named entities, addition tags can be added for twitter specific elements like emojis. Another improvement is necessary for the memory allocation of each executor in Spark. For the NER and POS tag processing, the library that is loaded into memory is too large. The profanity list requires further memory. So optimizing the number of executors per node is necessary which could speed up the processing. Another optimization can be done by merging data transferring and data cleaning steps into a single step task.

For the social network, our plan is to parallelize the conversion of RDF N-triple format using Spark and instead of using two intermediary step conversions to CSV from JSON and then converting to N-Triple, we plan to parse the JSON file directly to N-Triple and store it on the server. We also aim to store more tweet relationships and along with user ID, we plan to store user names, user's top N friends, followers and top N tweets posted by the user based on highest favorite counts.

[3]https://jena.apache.org/documentation/fuseki2/fuseki-security.html

9 Acknowledgements

This material is based upon work on the Collaborative Research: Global Event and Trend Archive Research (GETAR) project, supported by the National Science Foundation under Grant No. IIS-1619028.

10 References

[1] Edward A. Fox, Kristine Hanna, Andrea L. Kavanaugh, Steven D. Sheetz, Donald J. Shoemaker, et al. Integrated Digital Event Archiving and Library (IDEAL). http://grantome.com/grant/NSF/IIS-1319578, 2014. (accessed).

[2] Global Event and Trend Archive Research (GETAR). http://www.eventsarchive.org/sites/default/files/GETARsummaryWeb.pdf, 2016.

[3] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schutze. *An Introduction to Information Retrieval*, chapter 2, pages 177–194. Cambridge University Press, 2009. Online.

[4] Faiz Abidi, Shuangfei Fan, and Mitchell Wagner. CS 5604: Information and Storage Retrieval, Collection Management Tweets, Final Report.

[5] Matthew Bock. A Framework for Hadoop Based Digital Libraries of Tweets. Master's thesis, Virginia Tech, Virginia USA, 2017.

[6] John Scott. *Social network analysis*. Sage, 2017.

[7] Wouter De Nooy, Andrej Mrvar, and Vladimir Batagelj. *Exploratory social network analysis with Pajek*, volume 27. Cambridge University Press, 2011.

[8] Stephen P. Borgatti, Martin G. Everett, and Linton C. Freeman. Ucinet for Windows: Software for social network analysis. 2002.

[9] Carter T Butts. SNA: Tools for social network analysis. R package version 2.2-0, 2010.

[10] Pure Python wrapper for the Twitter API. https://github.com/ryanmcgrath/twython, 2013.

[11] Twitter for Python. https://github.com/tweepy/tweepy/, 2014.

[12] How to use RDF and the framework. http://docs.rdf4j.org/rdf-tutorial/, 2017.

[13] Jehiah Czebotar. json2csv. https://github.com/jehiah/json2csv, 2016. (accessed).

[14] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The Stanford Corenlp natural language processing toolkit. In *ACL (System Demonstrations)*, pages 55–60, 2014.

[15] Resource Description Framework (RDF): Concepts and abstract syntax. https://www.w3.org/TR/rdf-concepts/#section-Concepts, 2017.

[16] Apache. https://jena.apache.org/index.html, 2017.

[17] Python script to fetch a large number of Twitter followers. https://github.com/ChunaraLab/Fetching-Twitter-Followers, 2015.