University POLITEHNICA of Bucharest

Faculty of Automatic Control and Computers,
Computer Science and Engineering Department

BACHELOR THESIS

Clustering Conversations in Social Networks

Scientific Adviser:
Prof.dr.ing Costin Chiru

Author:
Victor Andrei Oprea

Bucharest, 2015

Maecenas elementum venenatis dui, sit amet vehicula ipsum molestie vitae. Sed porttitor urna vel ipsum tincidunt venenatis. Aenean adipiscing porttitor nibh a ultricies. Curabitur vehicula semper lacus a rutrum.

Quisque ac feugiat libero. Fusce dui tortor, luctus a convallis sed, lacinia sed ligula. Integer arcu metus, lacinia vitae posuere ut, tempor ut ante.

Abstract

Here goes the abstract about Streamer. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean aliquam lectus vel orci malesuada accumsan. Sed lacinia egestas tortor, eget tristique dolor congue sit amet. Curabitur ut nisl a nisi consequat mollis sit amet quis nisl. Vestibulum hendrerit velit at odio sodales pretium. Nam quis tortor sed ante varius sodales. Etiam lacus arcu, placerat sed laoreet a, facilisis sed nunc. Nam gravida fringilla ligula, eu congue lorem feugiat eu.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This is just a demo file. It should not be used as a sample for a thesis.

TODO:
Remove this line (this is a TODO)

1.1 Project Description

1.1.1 Project Scope

Ce nu ar trebui sa lipseasca: - cum ai gandit rezolvarea problemei - care este arhitectura aplicatiei - ce ai facut tu ca implementare - ce metoda de testare ai folosit - care sunt rezultatele obtinute

Chestiile astea ar trebui sa fie in jur de 30 de pagini. In plus, inainte ar trebui sa ai o introducere in care sa pui: - descrierea problemei 1-2 pagini - state-of-the-art (ce s-a mai facut similar) -3-4 pagini - scurta descriere Twitter (jumatate de pagina cred ca ar fi suficient) + - ce instrumente externe ai folosit (maxim 3-4 pagini)

The scope of the project **Streamer** is to provide close to realtime clustering of conversations that take place in the online medium. My choice for a social network is Twitter. Twitter has around 302 million active users (May 2015) [1] who send 500 million tweets each day mostly from their mobile phones. A tweet is a 140 character long message and because of this conversations are hard to keep track of and provide little to no context on their subject. This makes them an excellent candidate for a clustering application like **Streamer** which aims to provide an overview for conversations spanning over all the topics the user of the application provided.

This thesis presents the **Streamer**.

This is an example of a footnote [2]. You can see here a reference to Section 1.1.2.

Here we have defined the CS abbreviation. and the UPB abbreviation.

The main scope of this project is to qualify xLuna for use in critical systems.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean aliquam lectus vel orci malesuada accumsan. Sed lacinia egestas tortor, eget tristiqu dolor congue sit amet. Curabitur ut nisl a nisi consequat mollis sit amet quis nisl. Vestibulum hendrerit velit at odio sodales

[1] https://about.twitter.com/company
[2] www.google.com

pretium. Nam quis tortor sed ante varius sodales. Etiam lacus arcu, placerat sed laoreet a, facilisis sed nunc. Nam gravida fringilla ligula, eu congue lorem feugiat eu.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean aliquam lectus vel orci malesuada accumsan. Sed lacinia egestas tortor, eget tristiqu dolor congue sit amet. Curabitur ut nisl a nisi consequat mollis sit amet quis nisl. Vestibulum hendrerit velit at odio sodales pretium. Nam quis tortor sed ante varius sodales. Etiam lacus arcu, placerat sed laoreet a, facilisis sed nunc. Nam gravida fringilla ligula, eu congue lorem feugiat eu.

1.1.2 Project Objectives

We have now included Figure 1.1.

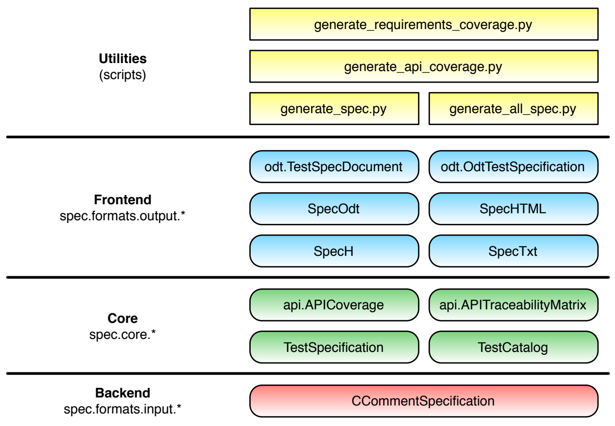


Figure 1.1: Reporting Framework

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean aliquam lectus vel orci malesuada accumsan. Sed lacinia egestas tortor, eget tristiqu dolor congue sit amet. Curabitur ut nisl a nisi consequat mollis sit amet quis nisl. Vestibulum hendrerit velit at odio sodales pretium. Nam quis tortor sed ante varius sodales. Etiam lacus arcu, placerat sed laoreet a, facilisis sed nunc. Nam gravida fringilla ligula, eu congue lorem feugiat eu.

We can also have citations like [**?**].

1.1.3 Related Work

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean aliquam lectus vel orci malesuada accumsan. Sed lacinia egestas tortor, eget tristiqu dolor congue sit amet. Curabitur ut nisl a nisi consequat mollis sit amet quis nisl. Vestibulum hendrerit velit at odio sodales pretium. Nam quis tortor sed ante varius sodales. Etiam lacus arcu, placerat sed laoreet a, facilisis sed nunc. Nam gravida fringilla ligula, eu congue lorem feugiat eu.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean aliquam lectus vel orci malesuada accumsan. Sed lacinia egestas tortor, eget tristiqu dolor congue sit amet. Curabitur ut nisl a nisi consequat mollis sit amet quis nisl. Vestibulum hendrerit velit at odio sodales pretium. Nam quis tortor sed ante varius sodales. Etiam lacus arcu, placerat sed laoreet a, facilisis sed nunc. Nam gravida fringilla ligula, eu congue lorem feugiat eu.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean aliquam lectus vel orci malesuada accumsan. Sed lacinia egestas tortor, eget tristiqu dolor congue sit amet. Curabitur ut nisl a nisi consequat mollis sit amet quis nisl. Vestibulum hendrerit velit at odio sodales pretium. Nam quis tortor sed ante varius sodales. Etiam lacus arcu, placerat sed laoreet a, facilisis sed nunc. Nam gravida fringilla ligula, eu congue lorem feugiat eu.

We are now discussing the **Ultimate answer to all knowledge**. This line is particularly important it also adds an index entry for *Ultimate answer to all knowledge*.

1.1.4 Demo listings

We can also include listings like the following:

```
CSRCS = app.c
SRC_DIR =..
include $(SRC_DIR)/config/application.cfg
```

Listing 1.1: Application Makefile

Listings can also be referenced. References don't have to include chapter/table/figure numbers... so we can have hyperlinks like this.

1.1.5 Tables

We can also have tables... like Table 1.1.

Table 1.1: Generated reports - associated Makefile targets and scripts

Generated report	Makefile target	Script
Full Test Specification	full_spec	generate_all_spec.py
Test Report	test_report	generate_report.py
Requirements Coverage	requirements_coverage	generate_requirements_coverage.py
API Coverage	api_coverage	generate_api_coverage.py

Chapter 2

State of the Art

2.1 Background

The internet is offering a voice to millions of people that have access to it. Creating content which is accessible by everyone in the network is possible with little or no barriers. Anyone can report on events, share their thoughts and ideas and because there are no limits to this process the results are massive amounts of information.
Facebook[1] and Twitter[2] are two examples of Internet platforms that have made it increasingly easier for people to generate content in a multitude of different formats: from text and pictures to rich media such as audio and video.
People send on average 50 million tweets with a peek record of 6939 TPS record. Facebook has even bigger numbers, in an average of 20 minutes, 1 million links are shared, 10 million comments are posted and 1,6 million wall posts are made[3].

TODO:
add number of users for each platform

. These are just two examples of popular social media websites and the rate at which content is being generated.
At the same time Facebook offers no way for a user to search through those comments and posts.
People communicate in short gists with the help of special annotations made possible on the platform. Mentions are a way of including another Twitter user into the conversation, they are formed by prepending the character "@" to the string that represents the user. Hashtags are a method of creating channels of communication, they are a way to distill the idea or felling of your tweet to a single word, and by doing so you ensure the inclusion of your message to a certain ongoing conversation. Hashtags are created by prepending the character "#" in front of words. Users can also reply to tweets, their message is grouped with the one they are replying to and context is preserved this way. Due to its short message format of 140 characters per message, Twitter has become a popular micro blogging platform for reporting on news and events as they occur. As a user you can always view the 10 most popular hashtags in different regions around the world or worldwide and participate in the conversation. You can also search for a certain query and messages that match gets returned either if it contains the string as a hashtag or in the message body.
The massive amount of information being generated especially on popular topics make it difficult to keep track of conversations as they happen. Unlike Facebook where the people you

[1] www.facebook.com
[2] www.twitter.com
[3] http://highscalability.com/blog/2010/12/31/facebook-in-20-minutes-27m-photos-102m-comments-46m-messages.html

interact with are mostly people you know and that number can be within reasonable limits, on Twitter there are no barriers in communication and you have access to all the messages produced by every user of the platform. Even though hashtags help filter conversations they are too inclusive, there are no constraints over how to use them or how many to use so messages are included to any conversation as long as they have the hashtag.
There are a number of services that use Twitter data and attempt to solve some of these problems. We will be presenting some of them in the following section.

2.2 Existing Solutions

2.2.1 Analytics

There are a number of analytics services that provide information regarding Twitter data. Most of them are businesses which offer information about the engagement of followers with the content created. Their goal is to help increase the visibility of tweets for businesses and therefor the metrics are related to the followers and focus less on exploring content.

This is also the solution offered by the Twitter analytics **TODO:** add link some of the information it provides is the number of user that views your messages, how many new users are now following your account.

One such example is SproutSocial **TODO:** add link which allow you to publish content from their application to Twitter, monitor your content for engagement and offer analytics on the users which interacted with your content.

Another example that tries to solve a similar problem to Streamer is TweetArchivist **TODO:** add link . You are able to query specific time frames and see top users and words related to certain search terms, as well as the most shared URLs and the most influential **TODO:** explain how they are influential users that have send messages.

2.2.2 3rd Party Clients

There are a number of 3rd party clients. They allow for filtering of content based on a particular hashtag and popularity (this is rated by number of retweets and favorites). This is a good alternative for finding popular opinions, you can judge it by how popular that certain tweet is but it conveys either the voice of popular users which get lots of favorites and retweets or some tweets which happen to gain popularity by accident.

2.2.3 Trends

Twitter website offers access to world-wide trends and also custom trends. First off world trends represents a list of key words present in tweets in a certain region. This allows you to browse all the tweets with those key words in real time. You do not have any other type of control over the data. The data is not grouped by any other means so exploring it means going through each tweet and reading it and taking into account the volume of tweets some trends may produce (as presented in the introduction of this chapter) this task may be impossible.

2.3 Related Work

Politics, Twitter, and information discovery by **Moritz Sudhof**. [1].
The aim of the paper is to cluster Twitter users into groups based on the opinions they expressed regarding a political controversy. The corpus is fixed and contains tweets from the time the events occurred, they have been selected due to using the same hashtag specific to the event. Several different attempts are made at clustering the users using different methods. *Tf-idf* and *odds weighting* are used to extract relevant key words from messages. Multiple keywords shared between tweets are an indication of how similar they are and thus link the users together. *Mentions*, referencing one or more users in your tweet, are also used. Mentioning someone in your tweet means that they are relevant to your opinion or somehow involved. Finally *hashtags* are taken into consideration the idea behind it being that users who send out messages using the same hashtags share similar opinions, again the more hashtags users share the similar they must be.

[1] http://web.stanford.edu/group/journal/cgi-bin/wordpress/wp-content/uploads/2012/09/Sudhof_Eng_-2012.pdf

Chapter 3

Design

Streamer is composed of two parts:

1. A backend that communicates with the Twitter API, fetches the data and performs the parsing the clustering. This exposes the information via a HTTP server that serves it in JSON format.

2. A frontend that takes the JSON and renders clusters of tweets as well as provides an interface for the user to explore the conversations

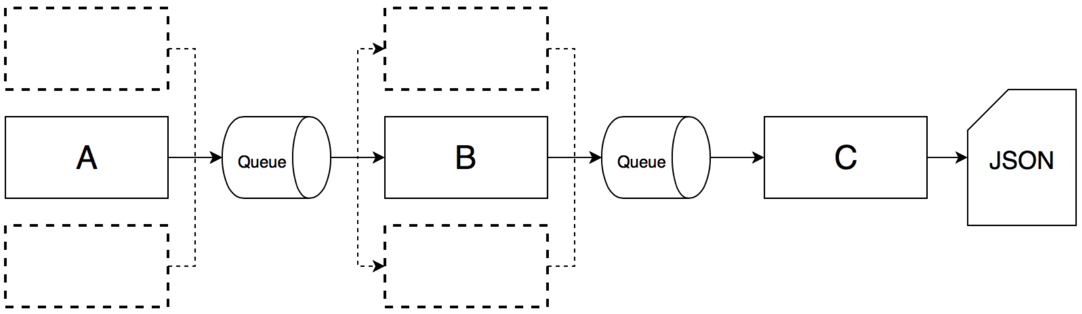


Figure 3.1: Design for the data processing pipeline

The data processing pipeline is composed of three main parts. Data acquisition (marked with A in the design) is responsible for fetching data that will later be processed, the component is agnostic of the data that passes through but in our case it is Twitter messages (tweets). It uses a library [1] to communicate with Twitter API, it retrieves the tweets and stores them in a queue. The queue library is provided by Apache Kafka [2]. As shown in the design, the reason for using queues between intermediary steps is that they allow for the producer and consumer to operate and different frequencies. The data acquisition segment could launch several clients regardless of what is happening further down the pipeline.

The second section (B), data processing, parses the raw tweets and converts them into shorter messages with key terms. Messages are read from the queue that is being filled by the data acquisition layer (A). The processed messages are written to a new queue, thus allowing this layer to scale just as the previous one. Tweets are parsed and using StanfordNLP library [3] each word is categorized with its own part-of-speech tag. Tags such as personal pronouns, possessive

[1] http://twitter4j.org/en/index.html
[2] http://kafka.apache.org
[3] http://nlp.stanford.edu/software/index.shtml

pronouns, prepositions, conjunctions are removed because they are too common. We could easily spin up several clients that consume messages because reading and writing is performed using two queues and thus the layer is decoupled from the other components of the pipeline.

The last part that processes data is responsible for clustering the messages based on the keywords generated in the previous step. The clustering algorithm uses K-Means with cosine similarity as a distance function, which I will go into more detail in the following section.

The visualization (Streamer-Frontend) is rendered in the browser. This offers the advantage of being able to explore the profile of Twitter users and see the messages and their context. It works by polling the webserver that is serving a JSON file through it API. When new data becomes available it renders the clusters and the corresponding adjacent nodes. The polling process will continue in the background. From the user interface you are able to see the clusters and quickly identify large clusters. You are able to see all the clusters that belong to it either by hovering over the nodes or clicking the cluster and getting an expanded view with all messages.

3.1 Implementation

In the following paragraphs I will go into further details on how the system is implemented.
The implementation is done in Scala[1]. The reason behind this choice is the fact that Scala enables us to use a functional programming paradigm and at the same time provides a type system that makes implementation easier. Many of the operations in the application include transformations of lists of messages, something that functional programming is very good at. At the same time the Scala source code is intended to be compiled into Java bytecode and run on the JVM. This allows us to include any Java library into the project as Scala was designed with Java interoperability in mind.
Another benefit of Scala is their implementation of parallel processing into the standard library. The aim of the language designers was achieving a high level abstraction that is easy to use thus achieving efficient parallel computations over collections in a transparent manner.

```
// Example of using parallel collections in Scala

list.map(_ + 42) // regular, sequential map over a collection
list.par.map(_ + 42) // parallel processing of the collection
```

Listing 3.1: Example of parallel collection usage in Scala

Using a similar approach we can speed up message parsing and also the clustering step. This decision has had beneficial results for the overall processing time of the Twitter messages. We will go into further details about the running time and speed benefits of parallel processing in the pipeline in the Testing and Evaluation section.

Accessing the Twitter API requires a developer account and an application created on their website dev.twitter.com [2]. The application gives you access to public, user or site streams. We will be using the public streams which returns data flowing through Twitter in real time given a certain array of keywords. This is the most useful endpoint for our data mining usecase.Using the provided API authentication tokens you can configure twitter4j library to retrieve tweets that match a specific keyword (or multiple keywords). The library comes with OAuth support and handles authentication with the endpoint. All messages received are passed to queue.

The messages retrieved are stored in a queue provided by Apache Kafka[3]. The queue, has a

[1]http://www.scala-lang.org
[2]https://dev.twitter.com/streaming/overview
[3]http://kafka.apache.org/documentation.html#introduction

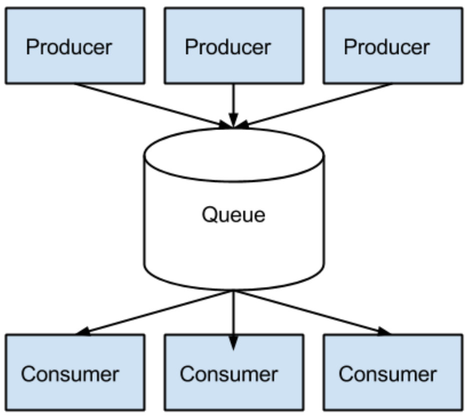


Figure 3.2: Design for queue processing in the pipeline

configurable storage period for the messages which allows us to use it for temporary storage. Communication between producers, consumers and the queue is done via a binary protocol over TCP so message delivery is always assured. The protocol does not require a handshake step in order to get or put messages in the queue, after a socket connection is established the client simply writes the messages it requires as request and then reads them. Kafka also guarantees message order and load balancing over a group of consumers but these features are not of interest for our project. Using a queue provides an advantage over the fact that messages arrive and are consumed at different frequencies. Depending on the popularity of the keywords specified tweets can come in at different rates. Based on a series of tests the rate of messages has been up to 150 per minute. At the same time the StanfordNLP has a parsing speed of one tweet per second. Using a queue also means that it is possible to start several consumers and producers at the same time. A consumer is concerned with getting the messages out of the Apache Kafka queue and placing them in a a new queue where they will eventually be parsed. The Apache Kafka documentation lists "Stream Processing" as an ideal use case for the library, and the processing of the Twitter API feed is exactly this sort of use case, thus confirming our design decision.

Before applying the clustering algorithm the tweets are first parsed. Parsing the tweets means removing all non alphanumerical characters or punctuation: such as unicode characters. One of the reasons for removing non-alphanumerical characters is that the StanfordNLP library cannot parse emojis [1].

The new messages are annotated using StanfordNLP library. This library reads the text and

[1] An emoji is a pictogram, similar to ASCII emoticons, which have been incorporated into Unicode meaning a wide adoption in online communication. Emoji symbols are two-byte sequences and support for them in browsers and mobile devices varies.

assigns a part of speech or other token to each word. The set of part of speech tags follows the Penn Treebank Project [1]. The resulting output is a list of tuples containing of the word, its POS tag and its level in the dependency graph. The dependency tree is constructed by drawing an edge between a token and the all others that it determines. The next step is to filter out words based on the part-of-speech tagging. Tags such as personal pronouns, possessive pronouns, prepositions, conjunctions are removed because they are too common and might introduce false positives for the clustering algorithm.
These parsed messages are pushed to a new queue. The reason for this is to completely decouple the 3 different stages of the pipeline:

1. Retrieving messages in realtime from Twitter using its API.
2. Tagging the messages with their part-of-speech tag and using this to filter out common terms.
3. Running the clustering algorithm using the parsed messages as input.

The final stage of the pipeline takes all the parsed messages from the queue and applies the clustering algorithm. The algorithm used for this step is K-Means clustering and the distance function is the cosine similarity. Because the design of the K-Means clustering algorithm is to start the iterations with a random seed consisting of randomly selected points (in our case messages), we run the clustering using different starting seeds and choose the highest value out of all the runs as our best clustering option. This in turn will affect the overall running time of the application as we will see. Depending on our goals either speed or precision we can choose to just rely on the first random seed provided.

3.1.1 Clustering Algorithm

The first step of the algorithm is to compute the TF (term frequency) and IDF (inverse document frequency) for each of the tweets. TF-IDF is a numerical statistic intended to provide information about the importance of a word in a document. TF is a direct measure for the number of times a word appears in a document while IDF helps scale down that measure for words that tend to appear often and scale up for words that are rare.

$$\text{TF(D, t)} = \frac{\sum \text{occurrences of t in D}}{\sum \text{number of terms in D}} \tag{3.1}$$

The TF of a term t in a document D is the total number of occurrences of that term divided by the total number of terms in the document. A document in this case would constitute a tweet. And the terms we are computing TF for are all the terms that remain after the filtering in the previous stage.

$$\text{IDF(D, t)} = \log\left(\frac{\sum \text{total number of terms in D}}{\sum \text{total number of occurrences of t}}\right) \tag{3.2}$$

The IDF of a term t in a document D is the total number of terms in D divided by the total number of occurrences of t in D. The document in this case constitutes all of the available tweets that we are attempting to cluster. The number of occurrences is also computed taking into account all the available messages.
Using TF and IDF we transform each tweet into a vector with weights for each of the terms. This way we can use the cosine similarity as a distance function in the clustering algorithm. Cosine similarity is a way of computing the similarity of two vectors by measuring the cosine

[1]https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

of the angle between them. The cosine function output is in the range [-1, 1] where -1 means that the vectors have opposite directions and 1 means that they have the same direction.

$$\text{cosine similarity(A, B)} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \times \sqrt{\sum_{i=1}^{n} B_i^2}} \tag{3.3}$$

The A and B vectors in our equation constitute two vectors i.e. two tweets that have been turned into vectors after computing their TF and IDF. And the result of the *cosine similarity* function tells us how similar the two vectors are, in return how similar two tweets are.

The clustering algorithm uses the K-Means clustering algorithm. The algorithm starts with an initial set of clusters chosen at random from all the tweets. It then uses the distance function (the cosine similarity described above) to assign the rest of the tweets to those clusters. For each cluster now formed it computes a weighted average. These steps are repeated only now we use the weighted average centroids instead of the random points until the centroids remain the same between 2 consecutive steps or until an upper bound of steps is reached. This upper bound is added in to ensure the algorithm halts if it fails to reach convergence in a fixed number of steps.

Different assignments yield different clusters which in turn have different weights. A higher weight means better similarity between the node and the cluster they belong to and therefore a better result. This similarity is computed with the help of the *cosine similarity* function after the K-Means algorithm has finished its assignment. For better results several iterations of the algorithm are used and the best score is chosen, at the cost of time spent clustering.

The result of the clustering algorithm as well as the tweet message and tweet author are combined and converted to a JSON data structure. This is in turn written to disk to be consumed by the endpoint accessing the server.. Using *Finagle*[1] an open source library from Twitter that provides an HTTP server implementation we can create the API endpoint. By having the data accessible via HTTP it can easily be consumed by any application regardless of the programming language used, also the reason we have chosen JSON as a format is because it is lightweight and most languages have an implementation of a JSON parser. The connection the the API endpoint is not kept alive, all clients are expected to poll the endpoint and react to changes. As an optimization the server could reply with status code 304 Not Modified [2] this would not include a message body and would let the client know that no new data is currently available. This would reduce the payload that has to be transmitted. On the client side this can be improved by using an exponential backoff approach for long polling. The polling period would always double when a 304 status code is presented and it would reset to a lower default value when the endpoint has new data available.

3.1.2 Data Visualization

Visualizing the data is made possible in the browser using JavaScript. We chose the web as a platform because it made more sense for a number of reasons:

1. It is easily accessible from a number of different devices such as laptops, phones and tablets much like the content we are clustering.
2. There are a multitude of libraries that implement visualizations, graphs, plots for JavaScript. Transforming the JSON file exposed by the API in a graphical representation is faster this way.

[1]https://twitter.github.io/finagle/
[2]http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

3. Using URLs the tweets can easily be traced back, the user profile and be viewed and content can be explored without having the implement it.

The visualization for Streamer named **Streamer-Frontend**[1] was build using two libraries.

1. React[2] an Open Source JavaScript library from Facebook that handles updating the DOM (Document Object Model) every time new data comes in.
2. D3.js[3] which stands for Data Driven Documents, this is an Open Source library used for plotting, graphs and other types of data transformations.

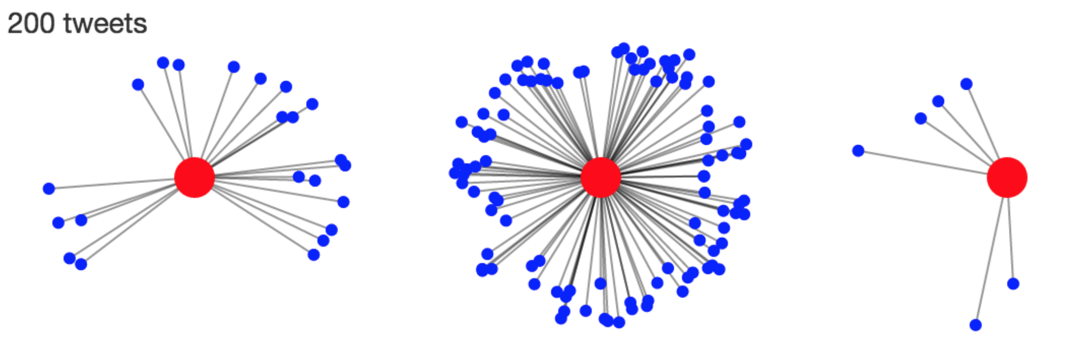


Figure 3.3: Example of cluster rendering

Streamer-Frontend polls the server API at a fixed time intervals and gets the latest version of the tweets as well as their respective cluster. Using an appropriate data structure we group the tweets together based on cluster id and draw the clusters to the webpage. Drawing is done using an SVG element with the help of the D3 library.
The interface presents the user a more meaningful representation of the data: tweets that belong to the same cluster are shown drawn together, also it is possible to explore the cluster and see all the messages that compose it.
The clusters will automatically update when new data comes in without the need to refresh the webpage. The application polls the API and when new messages become available it will redraw the clusters.

3.1.3 Deployment

For automated deployment of the project I used Docker[4], it is an Open Source project that runs applications inside of software containers. A container is similar to a virtual machine but avoids the overhead of it by sharing the same kernel as the host. Dockerfiles are configuration files for Docker containers, they are scripts that describe the steps necessary to create the environment for the application. A Dockerfile for Streameris also available as a gist[5]. Some of the requirements of the project that the container automatically configures:

1. Scala 2.10

[1] add link to Github
[2] https://facebook.github.io/react/
[3] http://d3js.org
[4] https://www.docker.com
[5] https://gist.github.com/piatra/53c0b6d185d10eeeaf8b

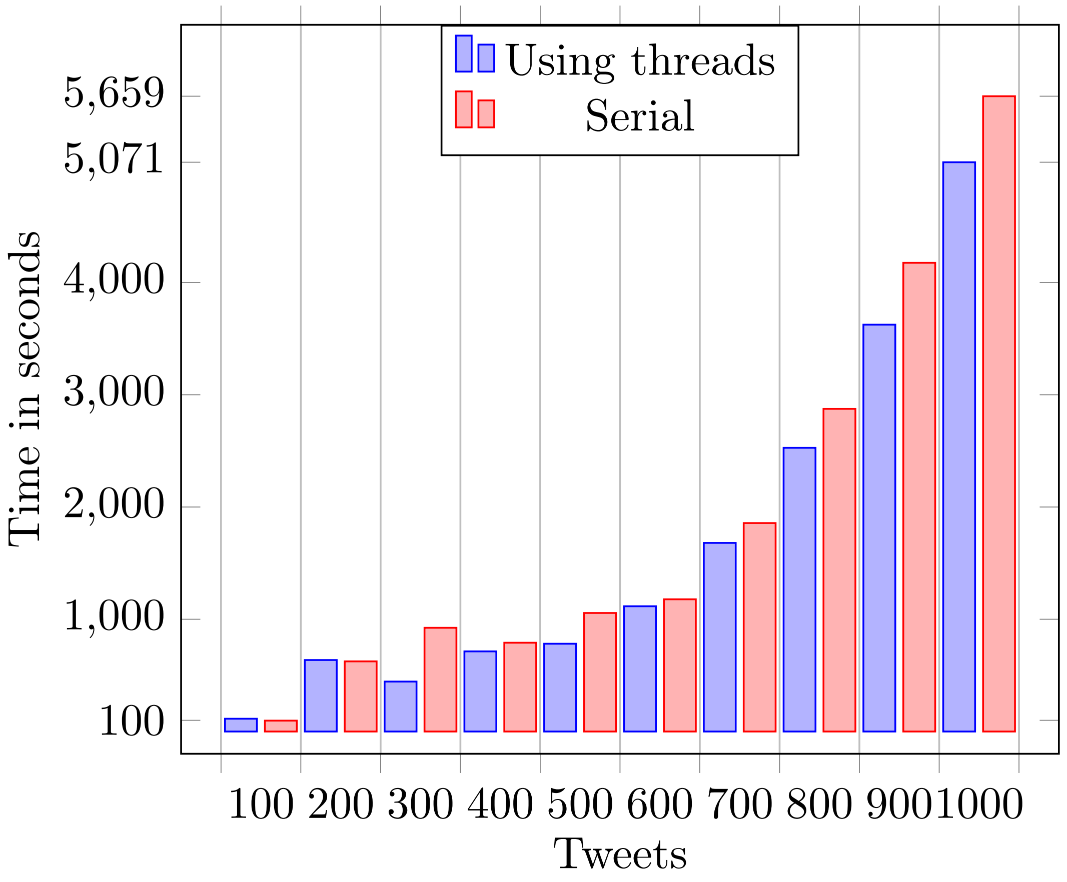
2. sbt 0.13.1 (Scala built tool) - Allows for task automation, running tasks and access to the Maven Central Repository for installing and updating packages
3. Kafka 2.9.1
4. Java 8
5. These are on top of Ubuntu 12.04

The script installs the required software with appropriate versions, updates all packages and it also created all the necessary configurations. Using this setup benchmarking was greatly simplified and we were able to try out different parameters for the clustering algorithm at the same time and observe how that affected the clustering performance and quality. Much of the testing involved in the development of the application was done on DigitalOcean. They are a PaaS[6] that offer virtual private servers, and one of the most useful feature they provide is the ability to start up several machines at the same time capable of running the project, each with custom performance capabilities. The servers used were equipped with 4 CPUs Intel(R) Xeon(R) CPUs and 8GB of RAM. Using virtual machines was an ideal setup for benchmarking because even though we used several different instances we were able to use the same snapshot across all machines thus ensuring a uniform testing environment. Once a container is started it will automatically begin fetching new data from Twitter and after an initial waiting period (in which the queue fills up with messages) the processing pipeline starts as well.

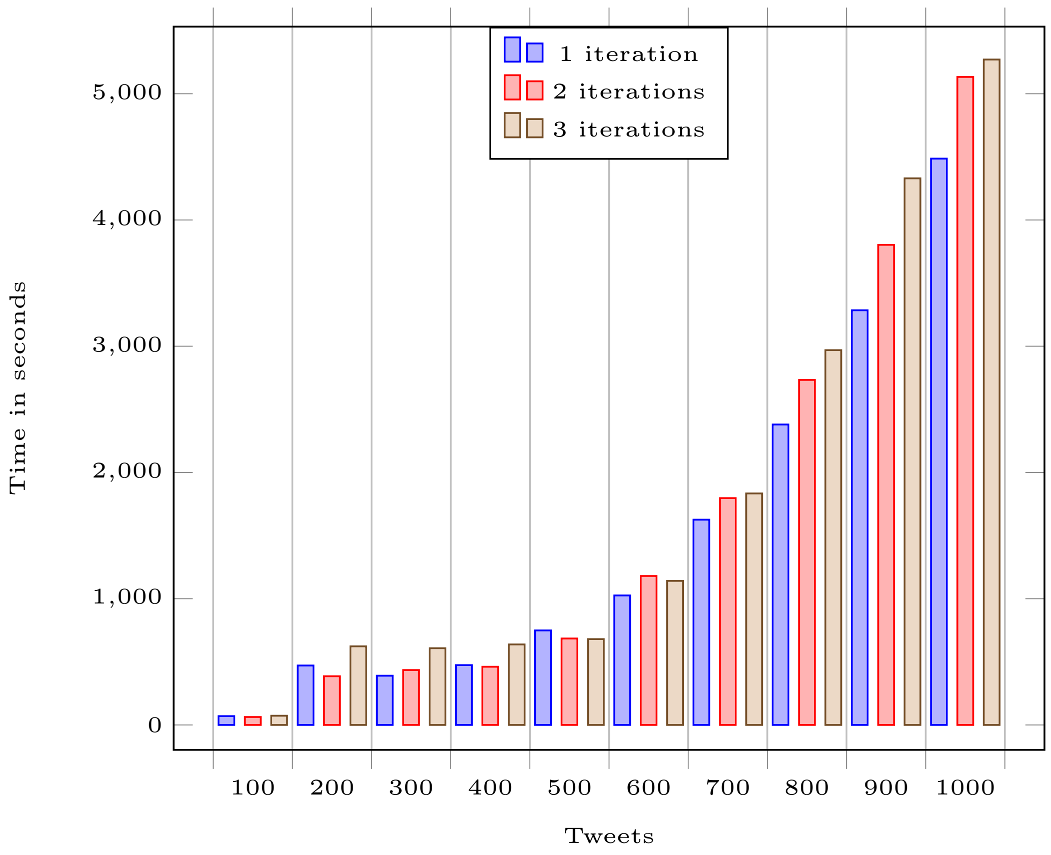
3.2 Testing and Evaluation

One of the advantages provided by the Scala programming language is its ease of manipulating collections using functions familiar to most functional programming languages such as map, reduce, filter, fold. This was an advantage that made it a perfect choice for manipulating large collections of messages. At the same time Scala provides *parallel collections*[1]. These are special collections included in the standard library that offer a high level API that facilitates parallelization. Calling the **.par** method on a regular collection such as **List**s or **Vector**s transforms it into a parallel collection and one can continue using that as a regular collection but now the transformations are done in parallel.
Below is a chart that outlines the performance improvements of using parallel collections. For the final set of data containing 1000 messages the performance improvement is of 588s (9,8 minutes).

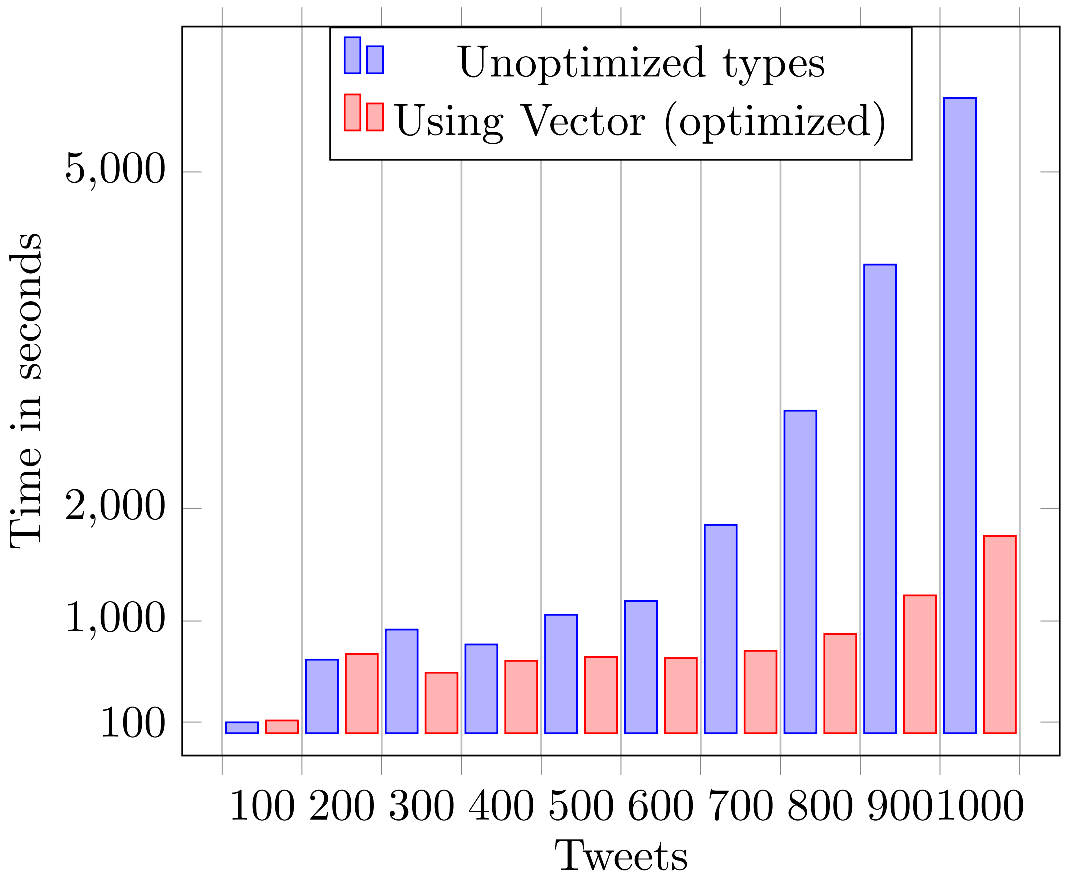
[6]Platform as a Service
[1]http://docs.scala-lang.org/overviews/parallel-collections/overview.html



As mentioned in the **Implementation** section, each cluster generated by the K-Means algorithm has a different weight and running the algorithm multiple times can yield better results, at the cost of running time.



Above we can see how running 1, 2 or 3 iterations impacts the running time.
An advantage of using queues means that the consumer and producer processes that interact with the queue are completely decoupled. Therefore we are able to spin up several producers and consumers at the same time. Producers could either be Twitter API producers that provide the tweets, this would speed up the cold start of the application, currently it waits for 6 minutes for the queue to fill up with messages. Another producer in the pipeline is the parser that handles part-of-speech tagging and filtering of messages, this consumes raw tweets and produces tokenized messages that are stored in another queue.
Types in Scala are another optimization that yielded improved performance. Parallel collections offer improved running time by delegating the work to several threads but using incorrect types can yield unfavorable results. **List**s in Scala are collections optimized for LIFO (last-in-first-out) type of access. Although random access is possible it comes with a performance loss. This disadvantage is also present when converting to a parallel collection *ParVector*, the overhead comes from the fact that all sequential collections such as lists have to be copied over to each thread. Using a different data structure improves performance especially when the quantity of messages increases.



From the chart above we can notice a 3x speed improvement when choosing the appropriate data structures.

Chapter 4

Conclusion

This is the conclusion. ce ați obținut, ce obiective ați avut, cum este relevant proiectul, ce rezultate ați obținut, cum ați evaluat.

The Internet allows anyone to create content, publish and spread information. Social platforms are lowering the barrier of entry and are making it especially easy for everyone to have a voice on the Internet and as a result millions of messages and content of all forms is generated daily. Making sense of everything, and keeping track is becoming difficult and therefore it is time for tools that help understand the content to evolve and adapt to these mediums and make content exploration as easy as it is to post a message.
Streamer attempts to solve this problem of content discovery and exploration. Streamer endeavors to understand the content being published and presents it to the user in a way that is accessible and easy to use. It creates clusters of messages by interpreting content and presents it to the user in a web interface that allows for him to browse through a large number of messages efficiently.
The feature that makes Streamer relevant for the fast passed rate of tweets is its ability to parse the messages in real time. The data is not based on an archived corpus of documents but on streaming tweets as they happen. This way popular events, news and messages get reported in the interface and the user is able to keep in touch with what is happening right now.

- Getting real time data from Twitter using its API based on user queries.
- Parsing tweets as they arrive. Using part-of-speech tagging to make annotations that help filter messages and extract important information.
- Using a clustering algorithm that is able to group messages, that has the ability to configure precision and that can run in parallel for a choice between speed and precision.
- Building a decoupled system that can easily scale through the use of queues which allow for different rates of consumption and multiple consumers that can process the workload in parallel.
- Presenting the information through an accessible medium: the web browser, with an easy to use interface that allows the information to be explored.

Chapter 5

Further work

The system is perfectly usable and but requires that each user deploys its own version of Streamer. Although the use of containers for automatic deployment via Docker reduces this task to running a script, the overhead involved with managing your own machine or environment where this can be done is a drawback. Wanting to improve on this issue but also increase the overall performance of the project, we have identified some areas where it can be improved:

- Having a centralized solution that offers immediate access to Streamer: it would offer Streamer as a service, allowing users to insert keywords or preferred sources of data and provide in return the clustered messages. This would remove the overhead of taking care of deployment and would make it easier to experiment with the project.
- Further improving the clustering algorithm both in terms of speed and precision. Currently a bottleneck of the project is the part-of-speech library which has an average parsing speed of 1 message per second. This could be replaced with other methods for determining the relevant keywords in a message. One example is using G-test log likelihood to remove statistically insignificant terms and improve clustering precision. This solution could run on multiple threads and improve the overall performance, but tests are required to ensure the project does not suffer a loss of precision.
- Using a distributed solution over multiple containers or even multiple machines. The container used in deployment contains both Streamer and the queues that hold the data. Extracting the queues would mean they can be shared between multiple instances and would be oblivious to any restarts to the server and would not lose data.
- Expanding the pipeline to add multiple consumers for the data, using multiple part-of-speech-taggers, and having more than one producer that retrieves data from the Twitter API. This would allow the system to retrieve more data as well as improve the time it takes to process it.
- Use websockets to notify Streamer-Frontend of new data that has been clustered. It would improve the user experience: right now when the API has new data the whole interface is redrawn including the clusters meaning that clusters may change position on the page making it harder to identify them. With a websocket implementation Streamer-Frontend can use event listeners and simply append new information to the page.
- Adding a storage layer or a caching layer in order to speed up the results. The storage could be shared between all running instances of Streamer and provided that the same query is used clustered results could be returned instantly. The storage layer could also hold raw or parsed tweets but basic check to ensure that the data is relatively new are

required. Caching would make more sense due to the fact that real time messages are expected.

Appendix A

Project Build System Makefiles

A.1 Makefile.test

```
# Makefile containing targets specific to testing

TEST_CASE_SPEC_FILE=full_test_spec.odt
API_COVERAGE_FILE=api_coverage.csv
REQUIREMENTS_COVERAGE_FILE=requirements_coverage.csv
TEST_REPORT_FILE=test_report.odt


# Test Case Specification targets

.PHONY: full_spec
full_spec: $(TEST_CASE_SPEC_FILE)
	@echo
	@echo "Generated full Test Case Specification into \"$^\""
	@echo "Please remove manually the generated file."

.PHONY: $(TEST_CASE_SPEC_FILE)
$(TEST_CASE_SPEC_FILE):
	$(TEST_ROOT)/common/tools/generate_all_spec.py --format=odt -o $@
	    $(TEST_ROOT)/functional-tests $(TEST_ROOT)/performance-tests
	    $(TEST_ROOT)/robustness-tests
#

# ...
```

Listing A.1: Testing Targets Makefile (Makefile.test)