Design and Implementation of a Component-based Distributed System for Text Mining in Social Networks

by

Yu Huang

A Project Report Submitted in Partial Fulfillment
of the Requirements for the Degree of

Master of Engineering

in

Electrical and Computer Engineering
University of Ontario Institute of Technology

Oshawa, Ontario, Canada

Abstract

Design and Implementation of a Component-based Distributed System for Text Mining in Social Networks

Yu Huang
University of Ontario Institute of Technology, 2016

Advisor:
Professor Qusay H. Mahmoud

This report presents the design and implementation of a component-based distributed system for text mining in social networks. The system consists of three main types of components, data collection, data processing and data visualization. Three possible frameworks explore simple linear architecture, message feedback architecture, Kafka centric architecture and provide implementations of them. The final system adopts Kafka-centric architecture in which all components are connected through Kafka brokers. In terms of functionality, data collection components are responsible for collecting data from Twitter and producing messages to Kafka brokers. Data processing components contain a series of basic text mining topologies. Based on JavaScript libraries, data visualization is presented on web pages and allows users to interact with graphs and charts. In order to improve the scalability and performance of text mining, the project selects Apache Storm framework to implement data processing components. In this report, we evaluate the availability of Kafka and Storm, the rates of data collection components and the performance of data processing components. The experimental results demonstrate our system is available and scalable, and the component-based structure of this system enables it to be extended easily.

Dedication

To my parents and girlfriend.
They mean a lot to me!

Acknowledgements

I would like to thank my parents and girlfriend first. Without their efforts and encouragement, I wouldn’t have the opportunity to study here.

I would like to thank my supervisor, Dr. Qusay H. Mahmoud as well, who gave me instructions and offered advice through the project.

I would also like to express my gratitude to Mr. Mark Neville, who is the writing specialist at the UOIT student learning centre. He gave me a lot of advice on English writing.

Last but not least, I would like to thank all the people I met in my life.

Table of Contents

Abstract ii

Acknowledgements iv

Acronyms and Abbreviations ix

List of Figures x

List of Tables xii

Chapter 1 Introduction 1

1.1. Motivation 2

1.2. Problem Statement 2

1.3. Contributions 3

1.4. Report Outline 4

Chapter 2 Background and Related Work 6

2.1. Social Networks 6

2.1.1. Social Network Data 7

2.2. Text Mining 8

2.2.1. Normal steps 8

2.2.2. TF-IDF 9

2.3. Distributed Stream Data Processing 9

2.3.1. Apache Storm 9

2.3.2. Twitter Heron 11

2.3.3. Spark Streaming 11

2.3.4. ZooKeeper 12

2.4. Data Visualization 13

2.5. Messaging Techniques 13

2.5.1. Kafka 15
2.6. Related Work 15
2.6.1. Social Mining 15
2.6.2. Data Visualization 17
2.6.3. Related Frameworks 18
2.6.4. Related Products 20
2.7. Summary 20

Chapter 3 Proposed Solution 21
3.1. Overview 21
3.1.1. System Architectures 22
3.2. Details of Architecture 25
3.3. Scalability, Expandability, and Availability 26
3.4. Trend Analysis 27
3.5. Summary 30

Chapter 4 Prototype Implementation 31
4.1. Overview 31
4.2. Selection of Technologies 32
4.3. System Implementation 34
4.4. Component Implementation 39
4.4.1. Data Collection Components 39
4.4.2. Data Processing Components 41
4.4.3. Data Visualization Components 47
4.4.4. Data Persistence Components 49
4.5. Component Management 49
4.6. Data Format 51
4.7. Challenges and Solutions 51
4.8. Summary 52

Chapter 5 Evaluation and Results 54

5.1. Experiment Environments 54

5.1.1. Vagrant and VirtualBox 54
5.1.2. AWS EC2 55

5.2. Deployment 56

5.2.1. Single Machine 56
5.2.2. Server Configuration 57
5.2.3. Deployment of ZooKeeper Cluster 58
5.2.4. Building Storm Cluster 59
5.2.5. Building Kafka Cluster 59

5.3. Twitter Stream Throughput Evaluation 60

5.4. Scalability of Data Processing Components 62

5.4.1. Evaluation of Sentiment Component 62
5.4.2. Evaluation of Word Count Component 64
5.4.3. Evaluation of Trend Analysis Component 66
5.4.4. Evaluation of TF-IDF 67

5.5. Availability 68

5.5.1. Availability of Kafka 68
5.5.2. Availability of Storm 71

5.6. Use Cases 73

5.6.1. Sentiment Analysis of Real-time Twitter Messages 74
5.6.2. Word Cloud of Real-time Twitter Messages 75

5.7. Summary 76

Chapter 6 Conclusion and Future Work 77

6.1. Conclusion 77

6.2. Future Work 78

Appendix A: Dependencies of Project 80

Appendix B: Configuration Files of ZooKeeper 81

Appendix C: Configuration Files of Apache Storm Cluster 83

Appendix D: Configuration Files of Apache Kafka..84

Bibliography 85

Acronyms and Abbreviations

TF-IDF	Term Frequency-Inverse Document Frequency
AWS EC2	Amazon Web Services Elastic Compute Cloud
NLP	Natural Language Processing
REST	Representational State Transfer
CAP	Consistency, Availability, Partition tolerance
URL	Uniform Resource Locator
CSS	Cascading Style Sheets
JSON	JavaScript Object Notation
JDK	Java Development Kit
DOM	Document Object Model
ML	Machine Learning
API	Application Program Interface
UI	User Interface
DRPC	Distributed Remote Procedure Call
JDBC	Java Database Connectivity
HDFS	Hadoop Distributed File System
CSV	Comma Separated Values
TSV	Tab Separated Values
vCPU	Virtual CPU

List of Figures

Figure 2.1: Storm Topology 11

Figure 2.2: Spark Streaming General Processing Flow 12

Figure 2.3: Kafka Connect Framework 19

Figure 2.4: Lambda Architecture 20

Figure 3.1: High Level Architecture 22

Figure 3.2: Simple Linear Architecture of Proposed System 24

Figure 3.3: Message Feedback Architecture of Proposed System 24

Figure 3.4: Messaging Technique Centric Architecture of Proposed System 24

Figure 3.5: Flow Chart of Trend Analysis 29

Figure 4.1: File Structure for Proposed System 31

Figure 4.2: Implementation of Simple Linear Architecture 37

Figure 4.3: Implementation of Message Feedback Architecture 37

Figure 4.4: Kafka Centric Architecture 38

Figure 4.5: Implementation of Lambda Architecture 39

Figure 4.6: Storm Sentiment Analysis Topology 43

Figure 4.7: Storm TF-IDF Topology 44

Figure 4.8: Storm Top N Topology 45

Figure 4.9: Storm Word Count Topology 46

Figure 4.10: Trend Analysis 47

Figure 4.11: Trend Analysis with Components 47

Figure 4.12: Web Page Flow Chart 50

Figure 5.1: Number of Messages Received By Twitter4J and Tweepy 61

Figure 5.2: Connection Time of Twitter4J and Tweepy 62

Figure 5.3: Evaluation Results of Sentiment Component 63

Figure 5.4: Evaluation Results of Word Count Component (1) 65

Figure 5.5: Evaluation of Trend Analysis Component 66

Figure 5.6: Evaluation of TF-IDF Component 67

Figure 5.7: Stormnode1 Screen Shot 69

Figure 5.8: Stormnode2 Screen Shot 70

Figure 5.9: Stormnode3 Screen Shot 70

Figure 5.10: Checking Connection of Three ZooKeeper servers 70

Figure 5.11: Kafka Consumer and Producer, When Closing one ZooKeeper Server 71

Figure 5.12: Kafka Consumer and Producer, When Closing Two ZooKeeper Server..... 71

Figure 5.13: The Screen Shot of Storm Nimbus Machine 72

Figure 5.14: The Screen Shot of Storm supervisor Machine 72

Figure 5.15: Storm UI with Two ZooKeeper Servers 73

Figure 5.16: Storm UI with One ZooKeeper Server 73

Figure 5.17: Web Index Page 73

Figure 5.18: Sentiment Map Visualization 74

Figure 5.19: Word Cloud 75

List of Tables

Table 4.1: Type 1 Message String 51

Table 4.2: Type 2 Message String 51

Table 5.1: Configuration File of Vagrant with Comments 55

Table 5.2: Details of Cluster 56

Table 5.3: Details of t2.micro 56

Table 5.4: System Information of Single Machine 57

Table 5.5: Evaluation Results of Word Count Component (2) 66

Chapter 1

Introduction

Social networks are defined as web-based services which allow users to compose their public profiles, connect with others and interact with their connected users [1]. The large amount of data on social networks provide opportunities for developers and companies to mine concealed relationships and values, while it presents challenges related to the capability of dealing with its large volume. The data from social networks can be divided into online and offline data. Commonly, online data are more valuable than offline data, as offline data can only provide analysis of history and habits. To address online data, Apache Storm, a real-time distributed framework is used universally. For example, Twitter, Yahoo!, Flipboard, and Taobao use Apache Storm to analyze real-time data [2].

Text data are more frequently observed on social network websites and express information more directly [3]. Other forms of data, such as audio, videos and images, are more complex and may incur ambiguity. On the contrary, the processing of videos and images are time-consuming. Thus these data are not suitable for real-time analysis.

With the increasing complexity of text mining problems, a single system becomes incapable of providing entire solutions [4]. Thus most systems require integration with other systems to a greater or lesser extent. To avoid reconstruction in the future integration, the proposed solution requires expandability.

Scalability is the other feature which system should be taken into consideration. With the increasing scale of problems, the developers should be able to add resources effortlessly.

This chapter describes the general ideas of this project. Section 1.1 discusses the motivation of this project, and Section 1.2 states the problems which are solved by the proposed project. Section 1.3 lists several contributions of this project, and finally Section 1.4 presents the organization of whole report.

1.1. Motivation

The project targets to provide a component-based distributed system for solving texting mining tasks in social networks. The system provides data collection, data processing, and data visualization components, which can be assembled into a social network text mining solution. In order to decrease the difficulties of grasping this system, the system allows users to manipulate all components with the index page. In the real world, the different text mining tasks have different data volume, thus the system is designed to be able to run both locally and in the cloud [5].

In terms of specific cases, based on this system, the report also develops a novel word cloud application of Twitter accounts, which provides some basic data processing components. By combining these components, users can customize their data processing tools. For a Twitter account, the system can generate the top trends by analyzing its timeline. Additionally, the project provides an application of Twitter sentiment analysis.

1.2. Problem Statement

In the real world, most people do not possess skills to analyze social networks, whereas their work may involve tasks of social analysis. Most existing social analysis tools have high pre-required knowledge which increases the difficulty of analyzing social networks.

To solve this predicament, the project aims to develop a series of social mining components which allows users to customize social analysis tools.

Due to the fact that social networks contain several data types such as audio files, images, videos, text is the most popular form within these types [3], the project only covers text mining techniques.

In order to implement a component-based social networks text mining system, all components in the system should be reusable. To increase reusability, every function is implemented as component. Users can customize data processing function by combining provided data processing functions.

1.3. Contributions

The contributions of this project are:

- **Implementation of distributed social mining components:** The project implements some social mining components, including sentiment analysis, TF-IDF algorithm, word count, and ranking algorithm.
- **Integration of different Apache Storm topologies:** According to the problem statement of Section 1.2, which requires data processing functions to be customized by users, the system has to implement dynamic topologies which allow users to change topologies while it is running. However, once Storm topology is submitted to the nimbus node, it is impossible to change Storm topology anymore. According to the single responsibility principle, Apache Storm uses bolt as the smallest data processing unit, which means each bolt only implements one function. In order to solve this problem, the project treats Storm topology as the basic data processing

unit, and each topology implements a set of natural language processing functions, instead of only one function per topology.

- **Implement an entire social mining system:** Instead of directly implementing natural language processing with Storm, the project contains four series of components: data collection, data processing, data visualization, and data persistence.
- **Novel features of architecture:** The authors of [6] simply apply NLP algorithm on the Storm, however this project provides an entire solution of text mining on social networks including collecting data from Twitter, NLP operators, data visualization components. This project treats every Storm topology as a component, in addition to the combination of Storm topologies that can be customized by users.
- **Novel features of trend analysis:** There are some existing tools for trend analysis, whereas only a few of them filter out stop words and almost none of them fetch the content of URL. Furthermore, most of the tools do not allow users to customize their solutions. Targeting these drawbacks, the project applies TF-IDF algorithm on contents of URLs which are contained in tweets and allow users to customize their trend analysis tools.

1.4. Report Outline

This report is organized as follows: **Chapter 2** introduces the background information of social mining and cloud computing. **Chapter 3** provides the general solution of component-based distributed social mining system and evolution phases of the final architecture. The implementation details of simple linear architecture, message feedback architecture, and Kafka centric architecture are described in **Chapter 4**. **Chapter 5** describes the details of experimental environments, deployment of dependencies and evaluates the scalability and availability of the project. Finally, **Chapter 6** concludes the report and introduces ideas of future work.

Chapter 2

Background and Related Work

This chapter presents all related fields of the project. Section 2.1 describes the most popular social networks and methods of acquiring data from them. Section 2.2 presents background of text mining techniques including tokenization, TF-IDF. The most popular distributed stream frameworks are listed in the Section 2.3. Section 2.4 presents related libraries of data visualization and Section 2.5 introduces popular messaging techniques which are available to integrate different components. Finally all related work is discussed in Section 2.6.

2.1. Social Networks

There are various social networking websites and applications including Twitter, LinkedIn, Facebook and Instagram as some of the most popular social networks in the world [7, 8].

Twitter is a real-time microblog which contains up to 140 characters. The tweets can contain images or videos as well as the most significant feature is its instantaneity. Additionally, Twitter contents are limited by the number of characters restriction, most tweets contain around 14 words and uses URL or hashtags to express the main idea. The hashtags indicate the keywords of tweets, however most hashtags are abbreviations [9].

Facebook data are more complex and contain all kinds of data. Compared to Twitter, messages on Facebook are much longer and contain more media contents. The most significant feature of Facebook is its relationship network. The data from Facebook is suitable for analyzing relationships between users [10]. LinkedIn focuses on job interview information which contains educational background, work experience and professional skills of users. Both Facebook and LinkedIn require users to register accounts with their real name, while Twitter and Instagram only allow account names to be unique and not contain any special symbols [10]. Aside from these social network sites, Reddit and Stackoverflow are popular as well.

2.1.1. Social Network Data

To collect data from social networks, there are several methods, official public APIs are implemented but the request rate is limited [11], web spiders which analyze the structure of web page and extract useful information [12], and static open data which are provided by third parties. In terms of public APIs, most social network tools, such as Twitter, Facebook and Instagram, provide APIs interface which allow developers to acquire their data, despite the fact that all these APIs have rate limits. Using public APIs is the most reliable way to get data. For example the authors of [13] built its data crawler based on Twitter stream APIs.

If users want to collect more data beyond rate limits, a web spider is an alternative. However not all social network websites permit the usage of web spiders. For example, LinkedIn announced that they do not allow developers to scrawl their information with web spiders. Additionally, almost all websites set traps or restrictions to prohibit web spiders. Compared to public APIs, the implementation of web spiders is more complicated than invoking APIs. The Social Media Lab builds their system to collect Twitter messages and stores them by different data fields [14].

Among all methods, acquiring static open data is the simplest way that only need few clicks, however the volume is limited and its instantaneity is the worst. Some websites provide open source data to users, such as Kaggle [15], Ontario Open Data [16], whose offline data sources are filtered and processed. The advantage of this method is that developers can download data directly without writing a line of code. The drawbacks are limited size and data lag. These offline data are suitable for examining accuracy and performance of algorithms though it is not advisable for analyzing recent information.

2.2. Text Mining

Text mining is "the process of extracting interesting and non-trivial patterns or knowledge from unstructured text documents" [3]. This Section introduces the basic concepts of text mining which are involved in further Sections, including tokenization, n-gram, part-of-speech, stop word, and TF-IDF.

2.2.1. Normal steps

The normal steps of text analyzing are [17]:

- **Tokenization:** Tokenization transmits sentence to words
- **N-gram:** "an n-gram is a contiguous sequence of n items from a given sequence of text or speech" [18].
- **Part-of-speech:** Part-of-speech tags the speech of word in the sentence. In most cases, noun is more important than other speeches.
- **Stop words:** Stop words are a list of meaningless words. Normally, NLP projects will remove stop words from the text to increase the accuracy of classification or clustering. Paper [19] contains dataset of stop words from multiple languages.

2.2.2. TF-IDF

Term Frequency Inverse Document Frequency (TF-IDF) is “a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus” [20]. There are several weighing schemes to calculate the IDF score. We employ document frequency smooth scheme: $tf(t,d) * \log\frac{\mathrm{D}}{1+df(t)}$. Applied in the calculation, tf(w,d) means the times word w occurs in document d, idf = log(n / df(w,D)), and df means the number of document which contains word w [21, 22].

2.3. Distributed Stream Data Processing

In distributed computing, Hadoop [23] is the most famous framework which only addresses batch data. However the data processed in this system are stream data which means data flow continuously and unboundedly. Additionally, the latency requirements of stream data are stricter than batch data. This Section lists several existing distributed stream frameworks, namely Apache Storm, Twitter Heron, and Spark Streaming. The authors of [24] list more stream data processing frameworks and categorize them into three types.

2.3.1. Apache Storm

“Apache Storm is a free and open source distributed real-time computation system” [25]. Apache Storm is first open sourced by Twitter and becomes one of the top projects in Apache organization. The structure of Apache Storm is shown in Figure 2.1 (adapted from [25]). The programming model of Apache Storm is called as spout and bolt where spout is the data source of topology and bolt is the data processing unit [26]. Along with the main processing structure, this is known as topology.

The followings are basic concepts of Apache Storm [27]:

- **Topology:** A topology defines the structure of data flow and consists of spout and bolt.
- **Spout:** A spout is the data source of topology, a topology can consist of multiple spouts.
- **Bolt:** A bolt is the basic data processing unit of Storm.
- **Tuple:** A tuple is the basic data unit of Storm, which can contain stacks of fields. The data structure can be string, list, or any types of objects.
- **Stream:** A stream contains a collection of tuples.
- **Grouping methods:** "A stream grouping defines how that stream should be partitioned among the bolt's tasks" [28]. Shuffle grouping which distributes tuples randomly to next bolt, field grouping which transmits tuples with same field to same bolt, and global grouping which concentrates all tuples to one bolt are the most frequently used grouping methods.

Besides the basic programming model, Storm also provides high level functions. "Trident is a high-level abstraction for doing real-time computing on top of Storm" [29]. Trident dramatically decreases the lines of code. It encapsulates commonly used functions, such as word count and filter, and simplifies bolt interfaces to operators.

In terms of programming language, Storm can almost support all programming languages as it integrates Apache Thrift.

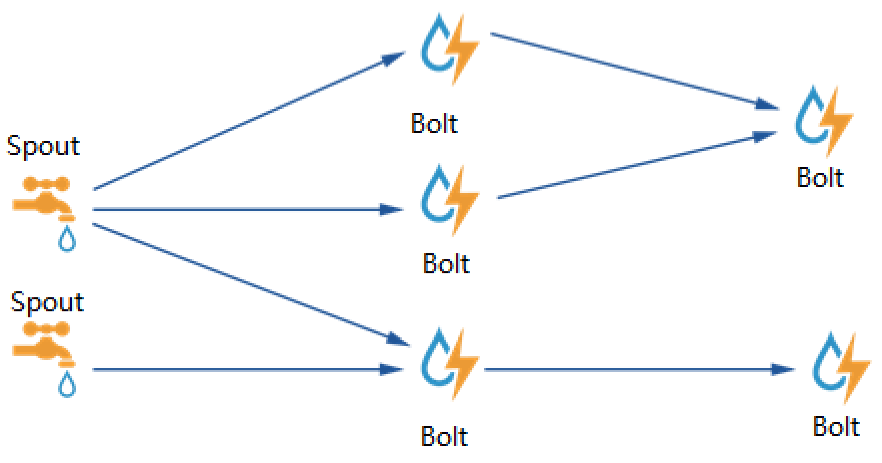


Figure 2.1: Storm Topology

2.3.2. Twitter Heron

Twitter Heron [30] was open sourced by Twitter recently, which is the successor of Storm. Since the inner systems of Twitter rely on Storm heavily, the Heron is designed to be applied with Storm code. Its programming model is the same as Apache Storm, thus it can apply Storm topology code directly. Heron only keeps basic programming models of Storm while it truncates high level functions of Storm, such as DRPC, and Trident. As the successor of Storm, Heron reduces the difficulties of debugging Storm topology in production environment. Compared to Storm, Heron also dramatically reduces the usage of CPU resources. In the evaluation section, CPU usage is the bottleneck of Storm topology.

2.3.3. Spark Streaming

"Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams" [31]. The architecture of Spark is shown in Figure 2.2 (adapted from [31]). Unlike other Stream frameworks, Spark has built-in machine learning and graph analysis algorithms.

Spark Streaming utilizes sliding window on stream data and transforms it to several batches of input data. Then the batches of data will be processed by Spark Engine. The core of Spark Streaming is still batch processing. The difference between Spark streaming and Hadoop is that the batches in Spark are much smaller [32].

The programming model of Spark is Map and Reduce. The map function applies methods on every data element, notwithstanding the reduce function combining the processed data together.

In terms of programming language, Spark only supports Scala, Java, and Python, where its Python version only supports a part of Spark APIs.

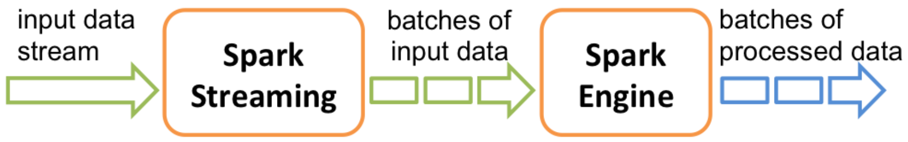


Figure 2.2: Spark Streaming General Processing Flow

2.3.4. ZooKeeper

“Apache ZooKeeper is an effort to develop and maintain an open-source server which enables highly reliable distributed coordination” [33]. Most distributed systems employ ZooKeeper to manage their clusters including Storm, Kafka, and Spark. The management details are packaged by these systems, thus developers only need to run zkServer.sh which runs ZooKeeper threads in the background. Besides this function, ZooKeeper also provides APIs which are suitable for building distributed lockers, distributed queues, and naming services.

Due to the fact that a ZooKeeper cluster is runnable when at least half of its ZooKeeper threads are running, a ZooKeeper cluster commonly consists of an odd number of machines. This feature authorizes high availability to ZooKeeper clusters, such that all distributed systems based on ZooKeeper clusters inherit availability.

2.4. Data Visualization

Data visualization is the simplest way to present data analysis results, as data can be visualized according to attributes, position, size, value, texture, color, orientation, and shape [34]. The following are visualization libraries:

- **D3.js:** D3.js (Data-Driven Documents) [35] is a powerful DOM selector library which allows developers to manipulate elements freely in the front-end.
- **Echarts:** Echarts [36] is a library developed by Baidu Inc. To operate it, developers only need set data set of x and y axises.
- **Google charts:** The usage of Google charts [37] is similar to Echarts, however it is more powerful and provides functions of transformation matrix.
- **Matplotlib:** Matplotlib [38] is a visualization library of Python, however it cannot be applied to browsers directly.

2.5. Messaging Techniques

As the architecture may involve several different sub-systems, it is necessary to include connection components to integrate these systems where messaging techniques are suitable selections. There are several advantages of using messaging techniques [39]:

- **Heterogeneous interoperability:** Different programming languages or different systems can be connected by messaging queues.
- **Component decoupling:** Each component can run separately.

- **High scalability through load balancing:** Messaging queues can be extended to several producers and consumers.
- **Asynchronous capabilities:** Since messages are stored in messaging brokers, the speed of producers does not influence the speed of consumers.
- **Guaranteed delivery of messages:** Using Kafka as an example, Kafka guarantees delivery of messages using log files. By configuring Kafka properties, the developer can decide the maximum size and resident time of logs.

The following are common used message queues:

- **LinkedBlockingQueue:** LinkedBlockingQueue is a Java built-in data structure and can be regarded as the simplest message queue.
- **Redis:** Redis is a key-value NoSQL database which stores data in memory. It owns a publisher/subscriber pattern which can be used to connect different components [40].
- **RabbitMQ/ZeroMQ/ActiveMQ:** These three message queues are very similar, they both implement point-to-point and broker architecture and are applicable for production environment.
- **Kafka:** "Apache Kafka is publish-subscribe messaging rethought as a distributed commit log" [41].Compared to RabbitMQ, ZeroMQ, and ActiveMQ, Kafka has lower runtime overhead, it can persist messages on the file system with complexity of O(1). According to the benchmark [42], the throughput of a single Kafka producer or consumer can reach around 1 million messages per second.

2.5.1. Kafka

Before applying Kafka in the system, the concepts of broker, topic, partition, producer, and consumer should be introduced [43].

- **Broker:** A cluster of Kafka containing several servers, every server is called a broker.
- **Topic:** Every message in Kafka belongs to one topic. By designating topic name, producers or consumers can push or pull messages from Kafka broker.
- **Partition:** One topic can include multiple partitions, which correspond to folders on the disk.
- **Producer:** Producers are responsible for pushing messages to Kafka brokers.
- **Consumer:** Consumers are responsible for pulling messages from Kafka brokers.

2.6. Related Work

The following introduces related work on this system, including social mining, data visualization, related framework, and relevant products.

2.6.1. Social Mining

Twitter messages are noisy and informal, thus there are multiple papers which propose researches mining information from Twitter data. The authors of [44] proposed a framework which rebuilds the pipeline of NLP with sequences of part-of-speech tagging, chunking, and named entity recognition to detect name entity in Twitter text. Compared to normal NLP pipelines, the performance of this framework increases F1 score by 25%.

The authors of [45] proposed an accurate open domain event extraction framework with 14% increase of F1 score. This project categorizes events and presents results on a calendar. The processing flow of this project is tagging the part-of-speech first, and determining whether nouns represent events, and categorizing them.

The authors of [46] both infer the estimated geographic information by analyzing tweet contents and metadata. The second project is implemented with Apache Storm. Furthermore, Google Cloud Prediction [47] and Monkey Learn [48] provide RESTful machine learning APIs of classification and clustering. The following are existing distributed machine learning libraries:

Apache SAMOA [49, 50]: It is a data mining platform applicable to S4 [51], Storm and Samza, which is designed for distributed stream data frameworks. Developers can implement a machine learning algorithm once and transcribe the code to multiple stream platforms which decreases the learning costs of building distributed machine learning applications [52].

Trident-ML [53]: Is a distributed machine learning library based on Trident which is “a high-level abstraction for doing real-time computing on top of Storm” [22]. Trident decreases the volume of codes to build Storm application by encapsulating commonly used methods but requires greater training and education on the part of developers. In terms of natural language processing, the Trident-ML provides TF-IDF, classifier, feature extractor, and a pre-trained Twitter sentiment classifier. However, its TF-IDF algorithm computes TF and IDF scores in the same bolt, which cannot modify the parallelism of TF and IDF separately [54].

StormCV [55]: StormCV is a combination of Apache Storm and OpenCV library, which processes images and videos in distributed systems. Unlike Trident-ML, StormCV implements all algorithms with spout and bolt programming model.

Likewise, there are still many other related projects. The authors of [56] generated network graphs of Twitter followers with python and networkx. The authors of [57] visualized a number of tweets on maps. The authors of [58] generated panorama information of specific topics from multiple platforms.

2.6.2. Data Visualization

Stacked graph [59] is a visualization method which presents stream data based on time series. The graph looks like a river and is stacked with a different theme. The authors of [60] and [61] both implemented themeriver with D3.js.

The authors of [62] proposed a visualization method called roseriver which extract topics from Twitter text and presents them in a ranking manner. The main idea of this paper is building a hierarchical evolution tree for each topic in real-time. The difficulty of implementing this system is that the amount of topics on Twitter is tremendous and most of them are not logically related.

The authors of [63] proposed a framework which visualizes the friendship between users with network graphs and allows users to search nodes and relationships with keywords. However once data corpus becomes enormous, the lines which represent the relationships of each node will become untidy. To solve this problem the authors of [64] simplified the number of lines by clustering them in groups based on the geometric information of nodes. The authors of [65] implemented the visualization graph of "Geometry-Based Edge Clustering" paper with d3.js.

2.6.3. Related Frameworks

There are two related frameworks, Kafka Connect which is applied to connect different systems to Kafka brokers and Lambda architecture which combines distributed batch data processing and distributed stream data processing.

2.6.3.1. Kafka Connect

Kafka Connect "is a tool for scalable and reliably streaming data between Apache Kafka and other data systems" [66]. The structure of Kafka Connect is shown in Figure 2.3. It officially supports the connection of JDBC and HDFS and the connectors of some framework communities.

Although both Kafka Connect and Kafka centric architecture use Kafka systems to connect other systems, Kafka Connect is only responsible for transmitting data between systems, a mere enhancement version of Kafka while complex data transformation operations are still required to be implemented by developers. Kafka centric architecture provides an entire solution of collecting, analyzing, visualizing social data, and more complex algorithms such as sentiment and trend analysis. Furthermore, Kafka Connect does not provide integration packages for Apache Storm. Kafka Connect focuses on general data transportation, however our system focuses on natural language processing of social network data. Besides, Kafka centric architecture is easily extended by implementing a Kafka producer or consumer, which is easier than extending Kafka Connect.

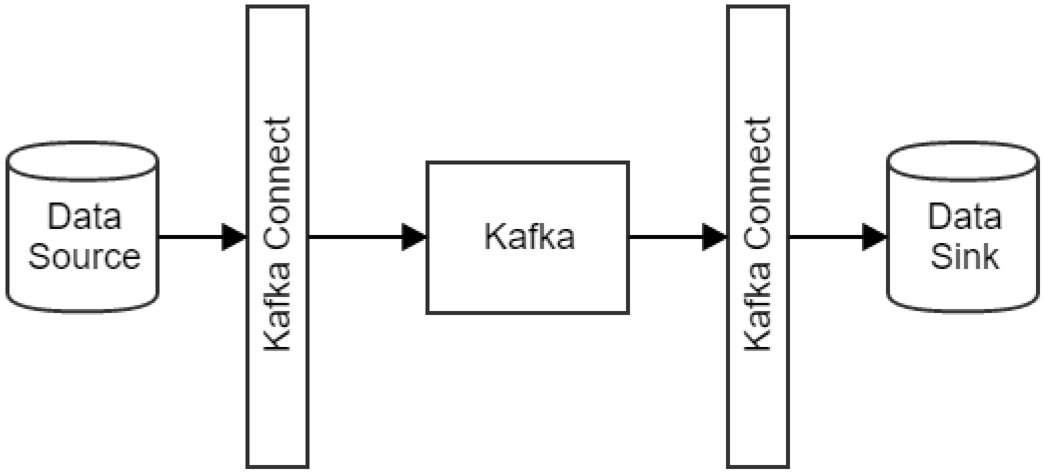


Figure 2.3: Kafka Connect Framework

2.6.3.2. Lambda Architecture

Lambda architecture is proposed by Nathan Marz [67]. The purpose of this architecture is to solve the CAP theorem which indicates that consistency, availability and partition tolerance cannot be satisfied at the same time [68].

Lambda architecture divides system into three layers, namely serving layer, speed layer and batch layer as Figure 2.4 shows. The speed layer is responsible for handling real-time data and storing message in the database. The batch layer is responsible for processing history data in batches. All results from speed and batch layer are sent to serving layer which interacts with clients. As a result, Lambda architecture has the capacity to address both stream and batch data. However lambda architecture requires developers to implement processing algorithm twice, once for spout and bolt model, another for map and reduce model. This drawback increases the costs of developing large systems dramatically [69].

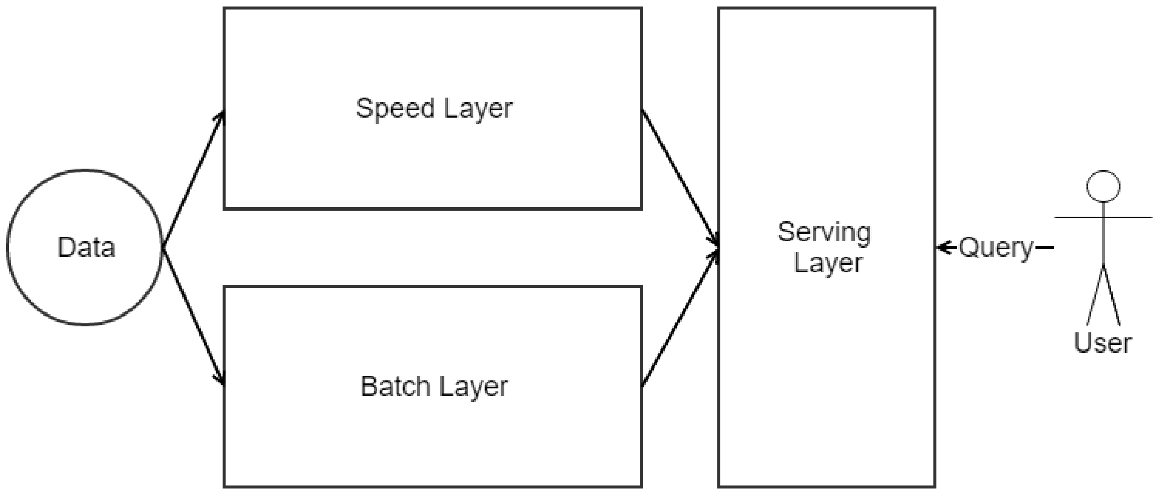


Figure 2.4: Lambda Architecture

2.6.4. Related Products

The following lists some existing social mining tools:

1. **Tweetstats:** Tweetstat [70] graphically shows the daily or monthly number of tweets from designated accounts.
2. **Xefer:** Xefer [71] graphically shows the behaviours of users when users post tweets every day.
3. **Twittercounter.com:** Twittercounter [72] graphically shows the by line chart
4. **Word cloud bot:** Word cloud bot [73] generates word cloud of Twitter account based on its tweets.

2.7. Summary

This chapter introduced the background of social networks, text mining, distributed stream framework, data visualization, messaging technique, deployment, and Lambda architecture. All these techniques are taken into consideration when we formulate the proposed solutions presented in Chapter 3.

Chapter 3

Proposed Solution

This chapter describes the high level architecture of whole system and chooses suitable tools from various options. Section 3.1 presents an overview description of high-level architecture and three proposed frameworks. The evolutionary procedures of these frameworks are presented in Section 3.2. Section 3.3 lists the libraries and systems which the project involves and explains the reasoning for choosing these components. Furthermore, the scalability, expandability, and availability are presented in Section 3.4. Section 3.5 shows the steps of trend analysis. Finally Section 3.6 presents the solution of deployment.

3.1. Overview

This section presents a high-level architecture of component-based distributed social network text mining system. As Figure 3.1 shows, the main purpose of this framework is to provide a general component-based architecture which can extract data from social networks, and allow users to observe visualization results with web browsers. The system has four types of components: data collection components which collect data from social networks, data processing components which process data and provide results to the next component, data visualization components which present data with visualization methods and provide the interfaces of websites to users, and data persistence components which store data in databases and allow users to query or reuse stored data.

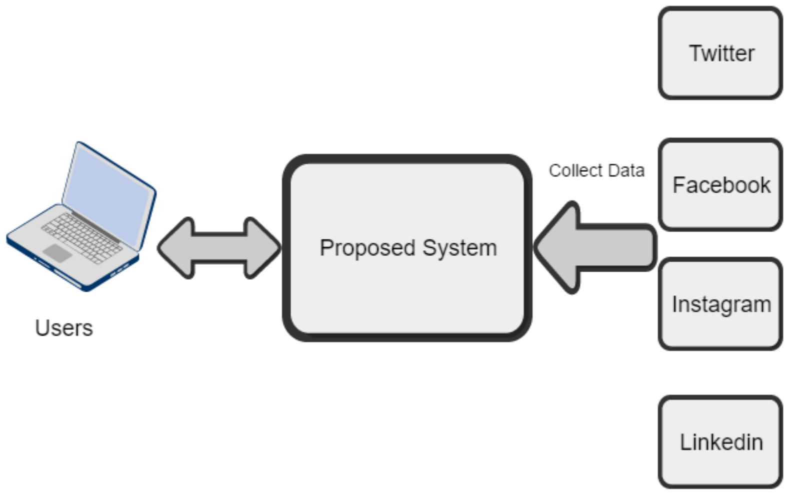


Figure 3.1: High Level Architecture

3.1.1. System Architectures

This section discusses the framework of the system. From the first simple linear architecture to the current Kafka centric architecture, the core architecture experiences the evolution of three architectures. Simple linear architecture and message feedback architecture are transitional products of Kafka centric architecture. They can only be applied to implement systems with single function. If users have more requirements, these architectures cannot be extended to satisfy their requirements. To solve the issue of expandability, we designed Kafka centric architecture.

As Figure 3.2 shows, the first generation architecture is called simple linear architecture. For the first proposed solution, the architecture does not involve the concept of component. In order to increase the diversification of options of techniques, we separate websites which are used for visualization along with other functions and connect them with a database, however the back-end and front-end are separated naturally. The data collection and processing part collects and processes data, then stores them into a database, furthermore the website queries data from the database then sends them to the front-end. In general, the main feature of this architecture is that the data flow between each component is uni-directional, which means the interactions with website do not influence data collection and processing. In this case, users can only view but not change the results.

As Figure 3.3 shows, compared to simple linear architecture the next generation architecture becomes more complex. On the basis of first generation, messaging techniques are involved to decouple data collection and data processing functions. As a result, the data collection components can be implemented with more techniques, the combinations of data collection and data processing become varied as well. The other feature of this architecture is that the front-end of websites are equipped with the ability to send feedback to messaging techniques to select data sources.

As Figure 3.4 shows, the current proposed architecture is messaging technique centric architecture. This framework involves the concept of component formally and treats every function as a component. The architecture is based on a cluster of messaging systems which are the centre of the architecture. Instead of integrating both databases and messaging techniques to connect different components, the architecture only applies messaging techniques. However, the database functions are packed as independent components. Specifically every component acts as both a producer and a consumer of the messaging cluster, which means they can push data to the cluster or pull results from it. Therefore all components can communicate with other components freely. Compared to former models, this architecture can add components easily by integrating producers and consumers of messaging system.

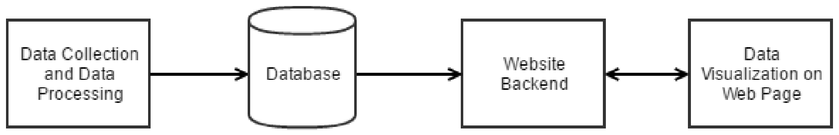

Data Collection and Data Processing
Database
Website Backend
Data Visualization on Web Page

Figure 3.2: Simple Linear Architecture of Proposed System

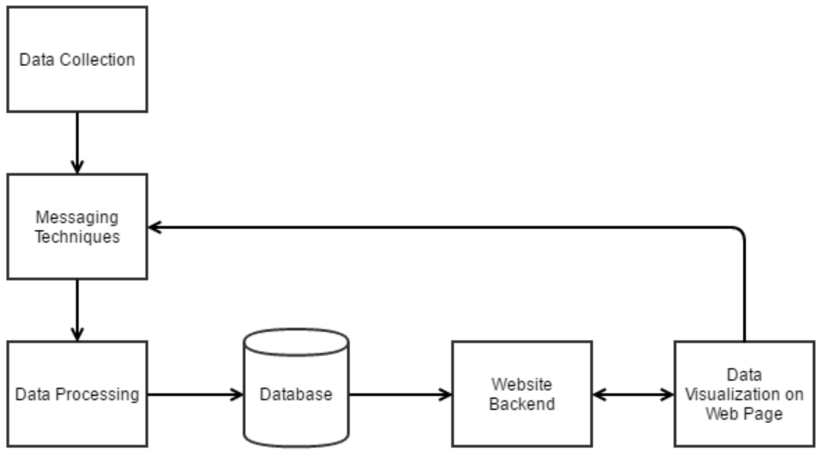


Figure 3.3: Message Feedback Architecture of Proposed System

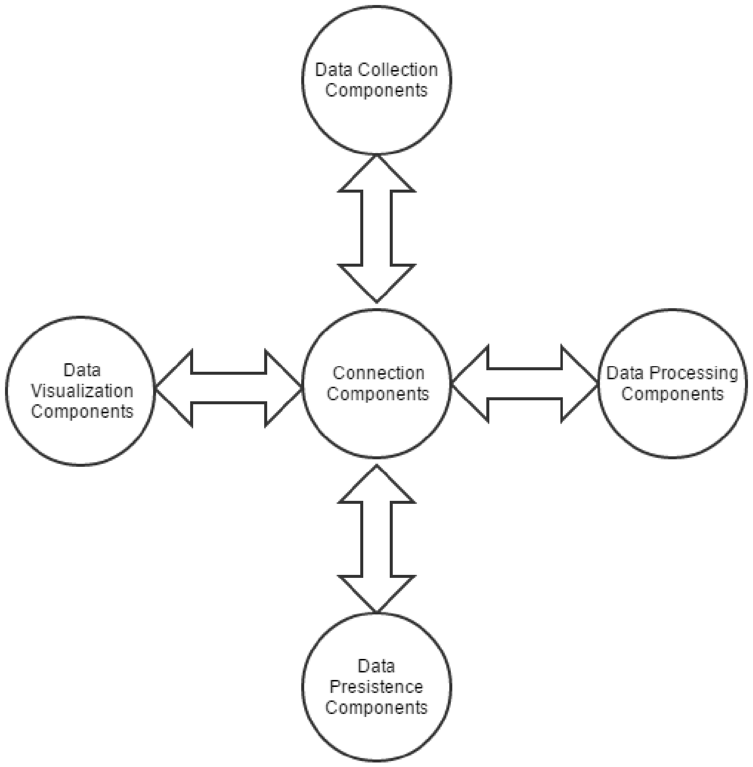

Data Collection Components
Data Visualization Components
Connection Components
Data Processing Components
Data Presistence Components

Figure 3.4: Messaging Technique Centric Architecture of Proposed System

3.2. Details of Architecture

As the above figures present, all architectures contain functions of data collection, processing, visualization and persistence. In the simple linear architecture, data collection and processing functions are encapsulated in one sub-system of the architecture. Data persistence functions are contained in the databases which serve to connect website, while data visualization is implemented in the form of web page. In the message feedback architecture, aside from the data collection and data processing functions being dissociated, other functions are similar to the first architecture. The messaging technique centric architecture divided four functions into different components completely. The following are details of the four different functions:

- **Data collection components:** The data collection components are responsible for collecting data from social networks. From the perspective of the single responsibility principle, data collection components should only collect data without any procession. However, since the transmission of data between different machines and systems is expensive the architecture requires data collection components to implement several simple operations. As the amount of original data from social networks is vast, the data collection components provide operations of filtering useless data and assembling different data fields to desired data formats.
- **Data processing components:** The data processing components operate to process data and deliver it to following components. Although there is no limitation of techniques to implement data processing, considering the scalability and the amount of data, distributed frameworks are recommended to be applied to implement these components. For the sake of the tasks of this architecture, data

processing components should implement NLP algorithms and integrate messaging techniques.

- **Data visualization components:** The data visualization components act as the platforms of presenting data results to users. In addition, these components can even supply interactions with users. Website is a suitable form to host visualization functions because techniques on the front-end are various and implementation of user interaction is accessible.
- **Data persistence components:** The data persistence components are alternative and not the requisite components in the system. It is responsible for storing data in the database to accelerate the process of presenting history data. In addition, data persistence functions can be integrated into connection or data processing components.
- **Connection components:** Beside the above functional components, connection components are important as well. When selecting techniques, the chosen should equip features that multiple languages and clients have implementation for. The connection components can be implemented with messaging techniques or databases, albeit the selected techniques should have capacity of extending to clusters to guarantee scalability.

3.3. Scalability, Expandability, and Availability

One target of this project is to deploy a scalable, extendable, and available framework in the Cloud. The scalability of this project can be represented in two fields, web and data processing. With more components involved, the project can deploy more Kafka brokers to address more messages.

In terms of web scalability, every client who wants to visit this website has to act as one consumer of Kafka. Thus the scalability of website directly relies on the number of consumers Kafka broker can host. Adjusting the number of parallelism can influence the performance of Storm.

The expandability is guaranteed by Kafka broker as well. Every component which involves Kafka producers or consumers can connect to the Kafka broker directly.

The availability of data on Kafka is adjusted by changing the numbers of replication and partition. In any distributed systems, machines may disconnect during runtime. To ensure high availability requirements, the simplest way is adding replications of messages and storing them in distributed machines. Thus the disconnection of one machine will not influence the whole system.

Normally, one Storm cluster only has one nimbus, however HDInsight supports the configuration of two nimbuses in one cluster. Although using a single nimbus may cause one point of failure, the nimbus is designed to be stateless and fail-fast. Once the nimbus fails, other machines still function properly and the nimbus restarts soon. Supervisors in Storm cluster are stateless as well, once supervisors fail, the workers tied to this supervisor will be assigned to other supervisors.

3.4. Trend Analysis

We introduce a solution of trend analysis and the flow chart of this solution is shown in Figure 3.5. The following are phases of analyzing the trends of tweets:

1. Analyzing the trends of Twitter account by fetching its timeline through the Twitter API.

2. Fetching the content of URLs which are shared by users: Twitter’s contents are limited to 140 characters, most tweets can hardly describe entire viewpoints. Instead of describing ideas, users post tweets with URLs and hashtags. URLs content describe what users want to share in detail and hashtags are keywords of the tweet which are recognized by users.
3. Tokenizing the contents of URLs.
4. Filtering out stop-words. Besides the normal stop words, Twitter text may contain some special stop words, such as “http”, “RT”, usernames.
5. Tagging the words left with their part of speech. In trend analysis, the noun is more important than the adjective and verb [60]. In order to improve accuracy, the algorithm also involves the concepts of unigrams, bigrams and trigrams. The combinations of speech tag are listed below.
   i) Unigrams: noun whose length is more than three.
   ii) Bigrams: noun + noun.
   iii) Trigrams: noun + noun + noun, noun + conjunction + noun.
6. Stemming, lemmatization and normalization: remove words affix and transform all different variations of words to the canonical form. More preprocessing steps can be found in paper [75].
7. Calculating the TF-IDF score of every gram.
8. Returning the top N grams to the data visualization component.

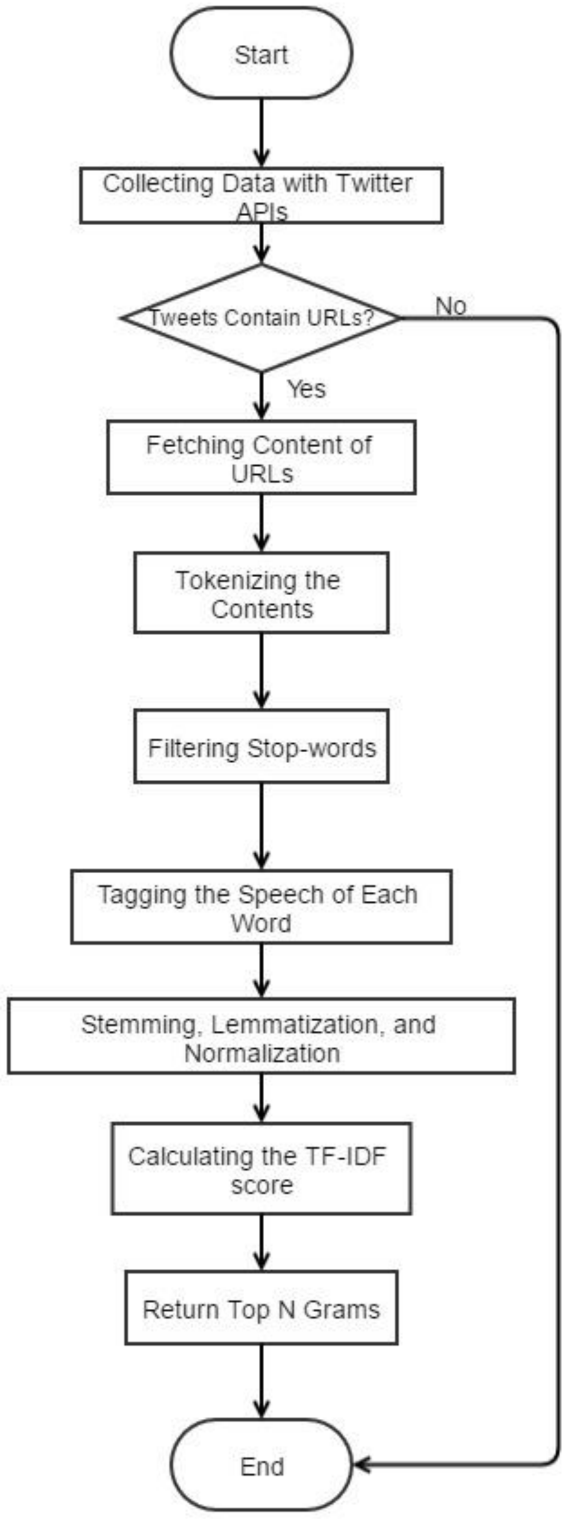

Start
Collecting Data with Twitter APIs
Tweets Contain URLs?
No
Yes
Fetching Content of URLs
Tokenizing the Contents
Filtering Stop-words
Tagging the Speech of Each Word
Stemming, Lemmatization, and Normalization
Calculating the TF-IDF score
Return Top N Grams
End

Figure 3.5: Flow Chart of Trend Analysis

3.5. Summary

This chapter describes the general solution of distributed text mining system and trend analysis in high level. The architecture consists of data collection, processing and visualization components and equips features of both scalability, expandability, and availability. In Chapter 4, the implementation details of proposed solution architectures are presented.

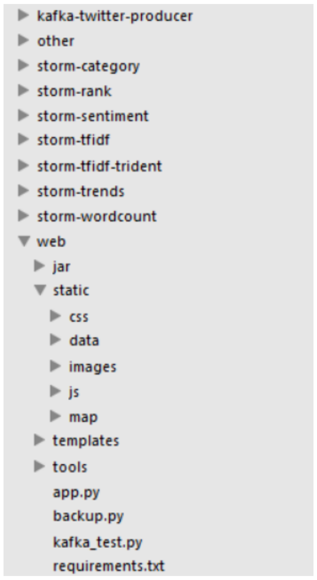
Chapter 4

Prototype Implementation

This chapter presents the details of implementation of this system. Section 4.1 presents an overview of system file structure. Section 4.2 lists implementations of all components, however component management details are described in Section 4.3. Furthermore, Section 4.4 defines a unified data format for data transmission. Finally, Section 4.5 presents the scalability and availability of the framework.

4.1. Overview

The system file structure is shown in Figure 4.1. The folder whose name begins with storm is the code of Storm component. Kafka-twitter-producer contains the code of Java implementation of Kafka producer. Since all Java code is packaged in the web/jar folder, web folder contains all necessary files of operating the system.



```
▶ kafka-twitter-producer
▶ other
▶ storm-category
▶ storm-rank
▶ storm-sentiment
▶ storm-tfidf
▶ storm-tfidf-trident
▶ storm-trends
▶ storm-wordcount
▼ web
  ▶ jar
  ▼ static
    ▶ css
    ▶ data
    ▶ images
    ▶ js
    ▶ map
  ▶ templates
  ▶ tools
    app.py
    backup.py
    kafka_test.py
    requirements.txt
```

Figure 4.1: File Structure for Proposed System

4.1.1. Web Files

The full files are shown above, this sub-section describes the function of each file. The *web* folder stores all web page files and packed jar files. The *jar* folder stores packed jar files of data processing components. The *static* folder stores resources of this website that the CSS folder stores CSS files, the *js* folder stores JavaScript files, the *data* folder stores test data sets in CSV or TSV formats, the *images* folder stores images of websites, the *map* folder stores map data in JSON or GeoJSON formats. Simultaneously, the *template* folder contains all HTML files of web pages. The app.py is the controller of this website which binds the HTML file with python functions, manages jar files and redirects webpages based on the submitted forms. All dependencies of python libraries are recorded in the requirements.txt and can be installed with python-pip tool.

4.2. Selection of Technologies

All related techniques are listed in the background section, this section introduces the techniques which are selected to implement the prototype and explains the reason behind choosing these techniques. We apply Twitter APIs to collect data from Twitter website and use Apache Kafka to connect different components. Apache Storm is utilized to implement data processing components. We choose a light weight framework Flask to implement the server side and integrate D3.js and Google Charts to the front-end.

The following are components which are used in the framework:

Twitter APIs: To simplify the problem, the system only collects data from Twitter through its public APIs. Although the public APIs of Twitter limit the data accessing rate, the volume of fetched data can still meet the requirement.

Apache Kafka: Apache Kafka acts as a message queue to connect different components. Apache Kafka is based on Zookeeper clusters, thus the availability and scalability of Kafka are guaranteed. All data which flow through Kafka will be stored as logs in the system, therefore it is unnecessary to involve persistent components in the system. Kafka is formatted into Kafka spout and grouped in Apache Storm as Storm topology's data source. Kafka equips the ability of storing messages in replications and partitions. With replication capacity, the transmission of messages in the project is fault-tolerant. By taking advantages of partitions capacity, Kafka can store messages separately and increase the parallelism of Storm spout. Compared to Redis, Kafka has a better scalability. The structure of Redis is different than Kafka. The framework of Kafka is based on topics which allow multiple consumers to receive messages at the same time. However the framework of Redis is based on queues, which means the messages in queues can only be consumed by a single client.

Apache Storm: "Apache Storm is the leading real time processing tool" [76]. In the system, Storm is adopted to implement data processing components. Apache Storm exploits the same programming model with Twitter Heron, which means one package can be submitted to both Storm and Heron clusters. Storm is called real-time Hadoop, the result can be processed and reflected to the website quickly.

Front-end is implemented with D3.js and Google Charts: As websites are scalable and interactive, visualizing data with websites has an edge over local platforms. Furthermore, the pre-requirement to view the visualization result is only opening browsers. In terms of JavaScript libraries, the Google Charts provide more powerful built-in functions; D3.js is more flexibly used to customize visualization methods.

Flask: The system utilizes Flask to achieve the server side of web site. Compared to other web frameworks, Flask hosts a few web pages with fewer costs. The other reason to use Flask is that Python provides libraries of invoking operating system commands which are used to orchestrate components in the system. Compared to other programming languages, the libraries of Python cover almost all fields such as system management, data mining and machine learning, which is suitable to attach different fields together.

4.3. System Implementation

There are implementation details of these proposed architectures and Lambda architecture. The justifications of implementing all these architectures are explained in the challenges and solutions Section.

Figure 4.2 shows the implementation of simple linear architecture. Data collection and data processing functions are implemented in the Storm topology. Redis database acts as the connection component which connects Storm topology and Flask server. Instead of storing data and querying data with database, the publisher and subscriber pattern of Redis is applied, where the report bolt of Storm acts as publisher and Flask server acts as subscriber. In this case the processed data are only stored in the memory, once the system is interrupted, all data will be lost. Side by side, the data visualization webpages are hosted on the Flask server. This implementation is straightforward, however it is not suitable for expansion. Data collection methods have to be embedded in Storm topology. In order to incorporate novel functions, the whole structure of Storm topology needs to be modified.

As Figure 4.3 shows, the message feedback architecture is implemented with Kafka, Storm, Redis, Cassandra, Flask, and frontend techniques. Data collection functions are decoupled from Storm topology with Kafka, which integrates Kafka producer and

Twitter API. Similar to the former implementation, Redis database acts as the connection component as well. The main difference is that website frontend can send feedback to Kafka broker to change the behaviour of data collection. Further, Apache Cassandra is selected to persist data.

Kafka centric architecture is the implementation of messaging technique centric architecture. As shown in Figure 4.4, the cluster of Kafka brokers is the centre of the architecture. For the reason that the message feedback architecture applies Redis and Kafka, both of them are messaging techniques. To simplify the structure, the architecture removes Redis and only employs Kafka. Besides performing data communication, Kafka also stores data as log file on the hard disk, while Redis only stores data in the memory. The phases of data transmission in this architecture are shorter than the message feedback architecture, which means once machines in the second architecture suffer from power off or failure, the data can be lost. In the current system, once data flows through Kafka brokers, they will persist on disks. In terms of scalability, to increase the data volume the former needs to increase the number of every connection component, while the latter only needs to scale the Kafka clusters. The architecture is loose coupled, which means the extended function can be easily added. The system aims to provide a dynamic social mining solution which grants users the ability to change the entire topology when project is running. However Storm cannot dynamically change its topology once topology is updated to clusters, the topology cannot be altered. To solve this problem, the system treats every Storm topology as a basic processing unit and connects them with Kafka brokers. In order to connect different topologies and other components, the Storm topology has to specify unified data formats as well. The functions of this system are listed in as following:

1. Generating analysis of trends of specific Twitter accounts.
2. Mining information of real-time Twitter data.
3. Visualizing data with different graph formats, including line graph, pie chart, bar chart, map, word cloud.
4. Integrating any components which implement Kafka producer or consumer.
5. Increasing scale by increasing the number of Kafka brokers
6. Storing data in Kafka brokers as logs.
7. Customizing Storm topologies.

Every function is encapsulated as one component and every component is connected to the Kafka broker. There are 3 types of components in the system, data collection components which collect real-time data or query results from social networks, data processing components which process NLP tasks concurrently, data visualization components which present analysis results in pie charts, bar charts, word clouds and maps.

Figure 4.5 shows the implementation of Lambda architecture. The speed layer is implemented with Apache Storm which provides real-time analysis results and stores processed results into Cassandra databases, however the batch layer is implemented with Hadoop and Hive. The HDFS loads history data from Cassandra to distributed file system while Hive provides data query interfaces. The server layer provides user interfaces which are similar to the data visualization components in above architectures. The main drawback of Lambda architecture is that developers need to implement the same algorithm twice, both in Storm and Hadoop frameworks.

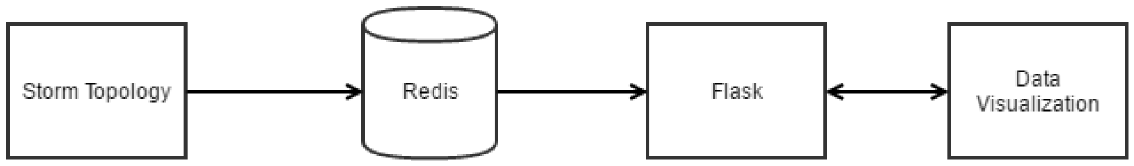


Figure 4.2: Implementation of Simple Linear Architecture

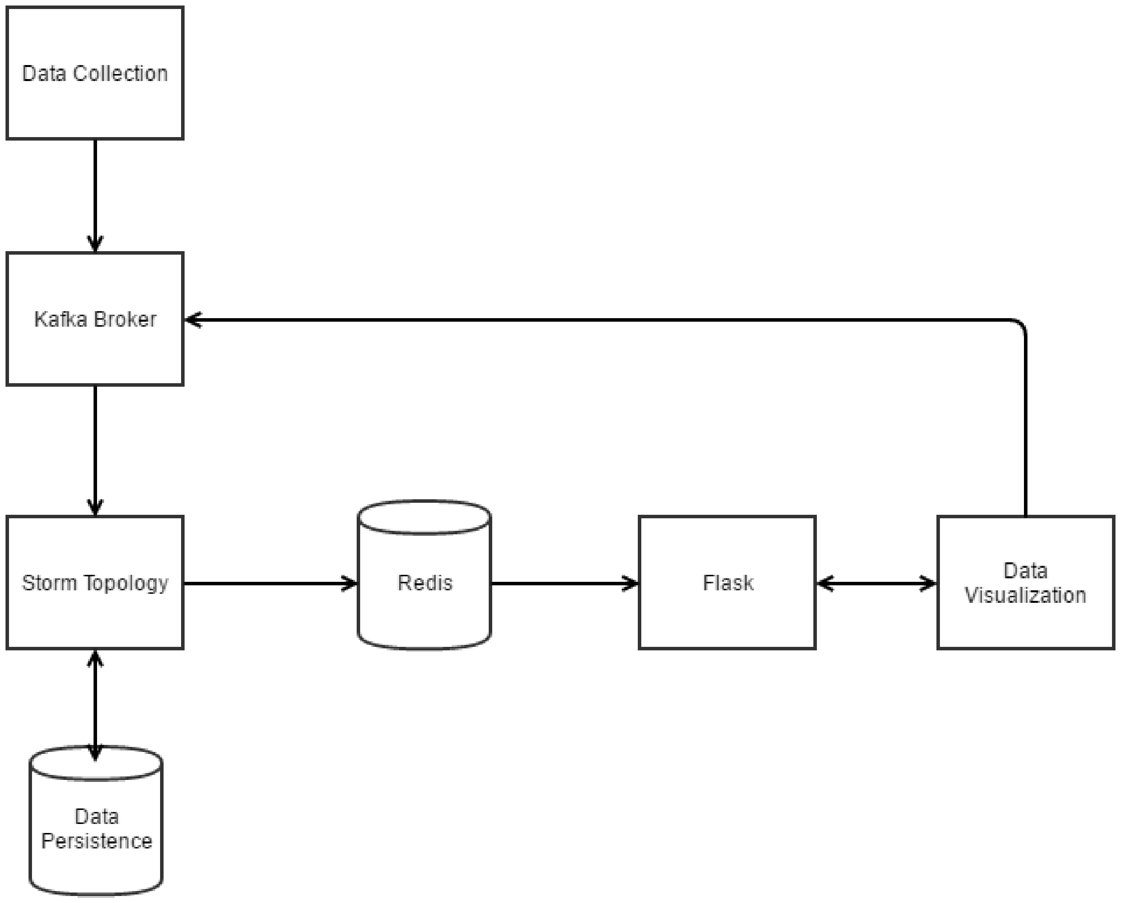


Figure 4.3: Implementation of Message Feedback Architecture

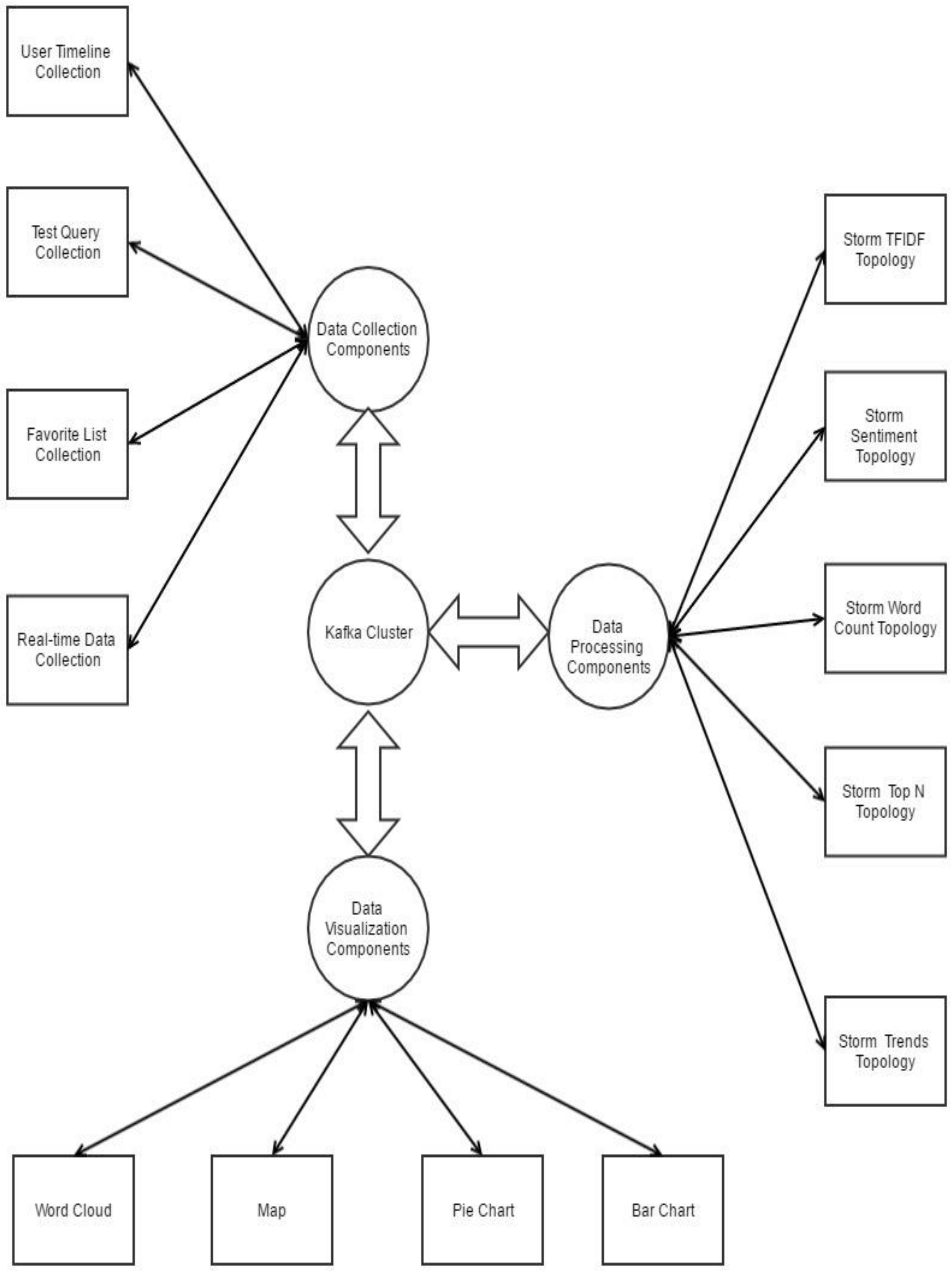


Figure 4.4: Kafka Centric Architecture

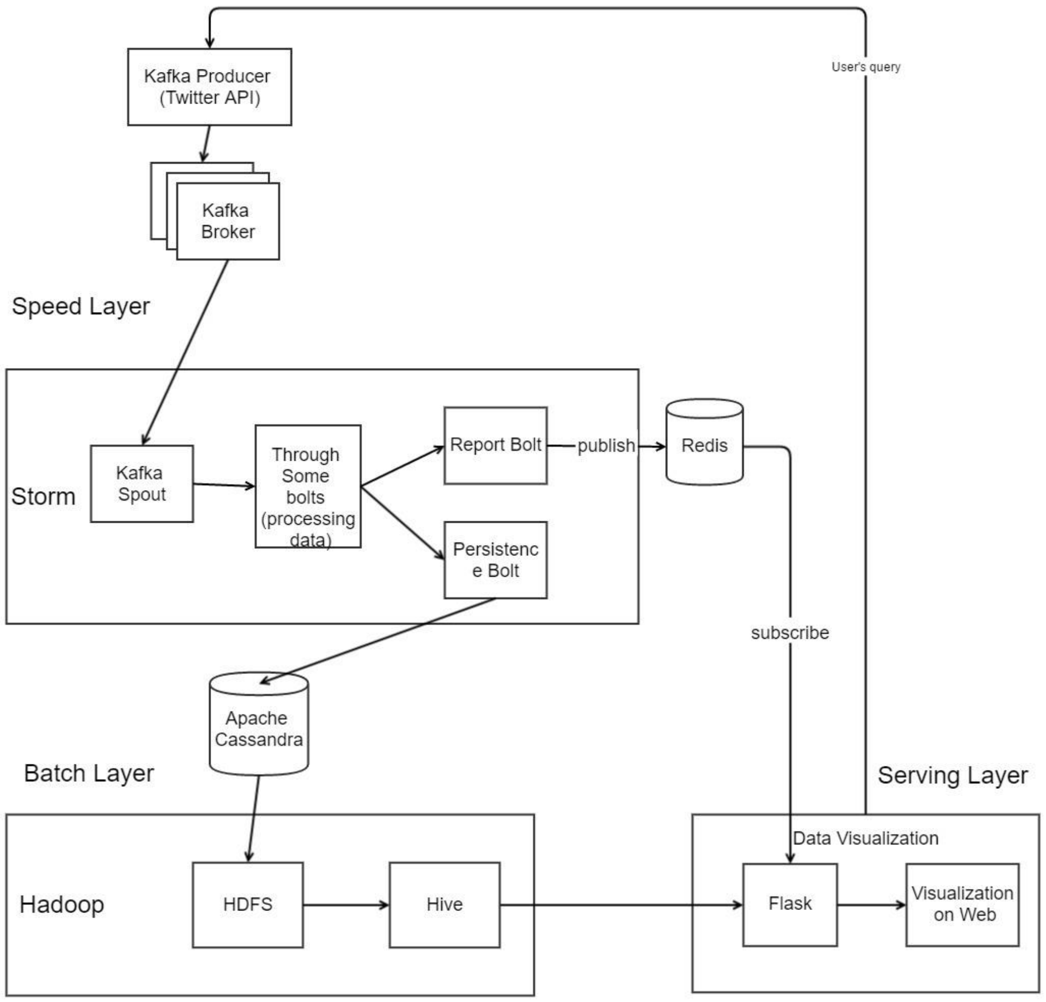


Figure 4.5: Implementation of Lambda Architecture

4.4. Component Implementation

The following section presents the details of every component in the Kafka centric system, including data collection components, data processing components, and data visualization components.

4.4.1. Data Collection Components

With Twitter APIs, developers can only access 1% of public Twitter data. Twitter also formulates rules of rate limits to restrict the handling of APIs. Twitter APIs utilize 15 minute windows to judge whether an application exceeds rate limits. Normally Twitter APIs authorize 180 queries in 15 minute time ranges, albeit for some expensive request, the rate limits are controlled within 15 queries per 15-minute windows [77]. Twitter status is the basic entity of Twitter message object, the system extracts five fields from it, namely tweet content which may contain URL, hashtags, mentioned user, whether it is retweeted, text, and emoji, screen name which is the account ID of user and is unique, created time which indicates when this message is made, geographic information including latitude and longitude, country code which is formatted in ISO alpha-2. The reason why we do not use country name is that sometimes country name may change by location, for example, the country name of Japan may become Japanese characters.

The system involves Twitter API implementations of Java version which is called Twitter4j and Python version which is called Tweepy. The usages and functions of these two libraries are almost the same, while their performance and accuracy are slightly difference, as discussed in Chapter 5.

To increase the varieties of data collection methods, the following are several implementations. In the Twitter API, there some useful parameters, **count** which indicates the number of tweets, **unti** which filters tweets created before given data, **lang** which filters tweets with given language, **geocode** which filters tweets within given area, **q** which indicates the contents of query string.

All implemented data collection components are listed as follow:

- **Real-time Data Collection Component:** Real-time data is most valuable in Twitter. Real-time data collection component collect real-time data by a Twitter stream listener and extracts desired information from Twitter statuses, then merges all information into one message.

- **User Timeline Collection Component:** According to the screen name input by users, user timeline collection component acquires all tweets in the timeline of this account and produces messages to Kafka clusters.
- **Text Query Collection Component:** By calling search APIs of Tweepy, text query collection component collects related tweets of input text and produces messages to Kafka clusters.
- **Favourite List Collection Component:** Based on input account name, favourite list collection component collects all tweets in user's favourite list and sends them to Kafka brokers.

4.4.2. Data Processing Components

Twitter data is up-to-date and informative, however it includes lots of fragments and noise. Before processing Twitter data, the project has to filter and clean the data first. In this project, we chose Apache Storm framework to implement data processing components and manipulate Maven to package Java files.

To guarantee the connectivity of topologies, every topology uses a Kafka spout as data source which consumes data from Kafka broker, and outputs processed data with Kafka producer. The project only involves three types of grouping methods, shuffle grouping, fields grouping, and global grouping. The definitions of these three grouping methods are introduced in the background section.

4.4.2.1. Sentiment Topology

The sentiment topology can process data from Twitter directly and generate personal and country sentiment. The personal sentiment is generated by classifying tweets contents, while country sentiment is calculated by counting average values of personal sentiments in the same country.

Tweets contain text, emoticons, emoji, URLs, hashtags, and punctuations. Among them, only text, emoticons, emojis, and exclamation points contribute to sentiment classification. The topology handles extractor of Twitter library to remove irrelevant content and keep related ones.

This topology applies Stanford NLP library to classify the content of tweets into different sentiments. The project manages its pipeline with “annotator”, “tokenize”, “ssplit”, “parse”, and “sentiment” operators. Emoticons are matched by regular expression. Owing to that emojis are images, there are no rules of detecting the sentiments of emojis. The system assists the list of emoji classification of Udacity’s code [78] to categorize emojis in tweets manually. Exclamation can enhance the extent of emotion, thus the project deduces that exclamation can increase level of sentiment.

The project stipulates five categories of sentiment, namely very **unhappy**, **unhappy**, **neutral**, **happy**, and **very happy**, and represents them with five numbers from 0 to four respectively.

The structure of sentiment analysis topology is shown in Figure 4.6. The following are details of its bolts, input and output:

- **Input:** The inputs of this topology are basic Twitter contents. The country sentiment function only works for tweets which have country code.

- **Output:** The outputs are messages which end with personal sentiment and country sentiment.
- **Sentiment Bolt:** The sentiment bolt adopts the pipeline of Stanford NLP library to analyze the sentiment of the tweet message.
- **Regex Bolt:** The regex bolt matches emoticons and emojis with regular expression and emoji classification lists.
- **Count Bolt:** The count bolt handles a distributed word count method to calculate the average sentiment of countries.

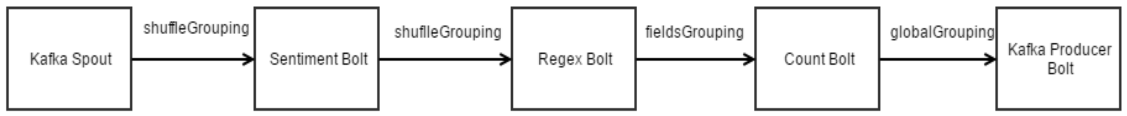


Figure 4.6: Storm Sentiment Analysis Topology

4.4.2.2. TF-IDF Topology

This topology implements a distributed version of TF-IDF algorithm. As mentioned in the background section, there are three values that should be collected to calculate the value of TF-IDF. Although storing intermediate results in the database is much easier, in order to increase speed the project keeps all data in the memory. On the grounds that TF-IDF function is important for documents and tweets are really short, the project only applies TF-IDF function on documents which are linked by URLs in tweets.

The structure of TF-IDF topology is shown in Figure 4.7. The following are details of its bolts, input and output:

- **Input:** The inputs of this topology are original Twitter messages.
- **Output:** The outputs are key-value pairs of TF-IDF score.

- **Document Fetch Bolt:** The document fetch bolt extracts web page contents through URL within tweets. The main library is Apache Tika which can incorporate a variety of formats.
- **Tokenize Bolt:** The tokenize bolt converts the document contents to words.
- **DfCount Bolt:** The dfcount bolt calculates df values.
- **TfCount Bolt:** The tfCount bolt calculates tf values.
- **Tfidf Bolt:** The tfidf bolt converge the result of dfCount bolt and tfCount bolt to generate TF-IDF score of each word.

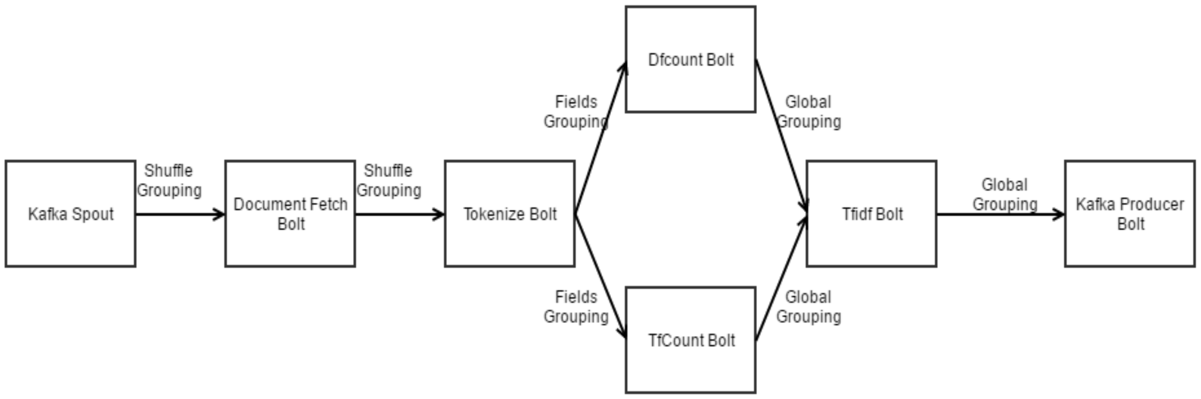


Figure 4.7: Storm TF-IDF Topology

4.4.2.3. Top N Topology

This component implements the algorithm of ranking keys by their values and returns top n keys. By reason that all processed tuples will be aggregated to the Kafka producer bolt, the parallelism of the final bolt would be 1. In order to increase the number of parallelism, the topology adds an intermediate ranking bolt which processes tuples first and then gathers them to a total ranking bolt. The structure of top n topology is shown in Figure 4.8. The following are details of its bolts, input and output:

- **Input:** The inputs of this topology can be any key-value pairs or messages containing key-value pairs.
- **Output:** The outputs are top n key-value pairs.
- **Parse Bolt:** The parse bolt converts string data to sortable objects which are easily processed by ranking algorithm.
- **Intermediate Ranking Bolt:** The intermediate ranking bolt increases the degree of parallelism of ranking algorithm.
- **Total Ranking Bolt:** The total ranking bolt converges all results of Intermediate Ranking Bolt and sends result to Kafka producer.

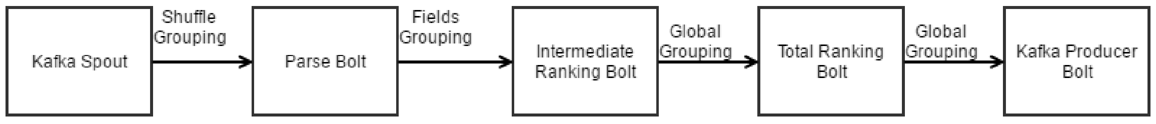


Figure 4.8: Storm Top N Topology

4.4.2.4. Word Count Topology

The word count is the “hello world” level algorithm in distributed systems. The structure of this topology is similar to MapReduce, because the splitter bolt can be treated as Map function which applies split function on each tuple and count bolt can be treated as Reduce function which gathers all count of keywords.

The structure of word count topology is shown in Figure 4.9. The following are details of its bolts, input and output:

- **Input:** The inputs of this topology are basic Twitter messages.
- **Output:** The outputs are Key-value pairs.
- **Splitter Bolt:** The splitter bolt converts tweet contents to word.

- **Count Bolt:** The count bolt calculates the number of each word.

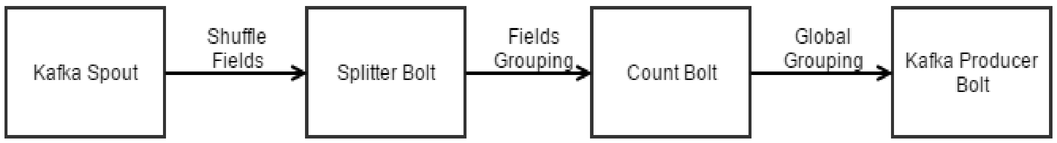


Figure 4.9: Storm Word Count Topology

4.4.2.5 Trend Analysis

Trend analysis topology is the most complex topology in this project, it is comprised of two components. The structure of word count topology is shown in Figure 4.10. As Figure 4.11 shows, if tfidf topology or word count topology are treated as upstream and top n topology is connected as downstream, there are two types of trend analysis topologies. The following are bolts which comprise trend analysis and their functions are similar to the bolts in tfidf and top n topologies.

The following are details of its bolts, input and output:

- **Input:** The inputs of this topology are basic Twitter messages.
- **Output:** The outputs are key-value pairs.
- **Document Fetch Bolt:** The document fetch bolt extracts web page contents through URL within tweets.
- **Tokenize Bolt:** The tokenize bolt converts the document contents to words
- **DfCount Bolt:** The dfCount bolt calculates df values.
- **TfCount Bolt:** The tfCount bolt calculates tf values.
- **Tfidf Bolt:** The tfidf bolt converges the result of dfCount bolt and tfCount bolt to generate TF-IDF score of each word.

- **Intermediate Ranking Bolt:** The intermediate ranking bolt increases the degree of parallelism of ranking algorithm.
- **Total Ranking Bolt:** The total ranking bolt converges all results of intermediate ranking bolt and sends results to a Kafka producer.

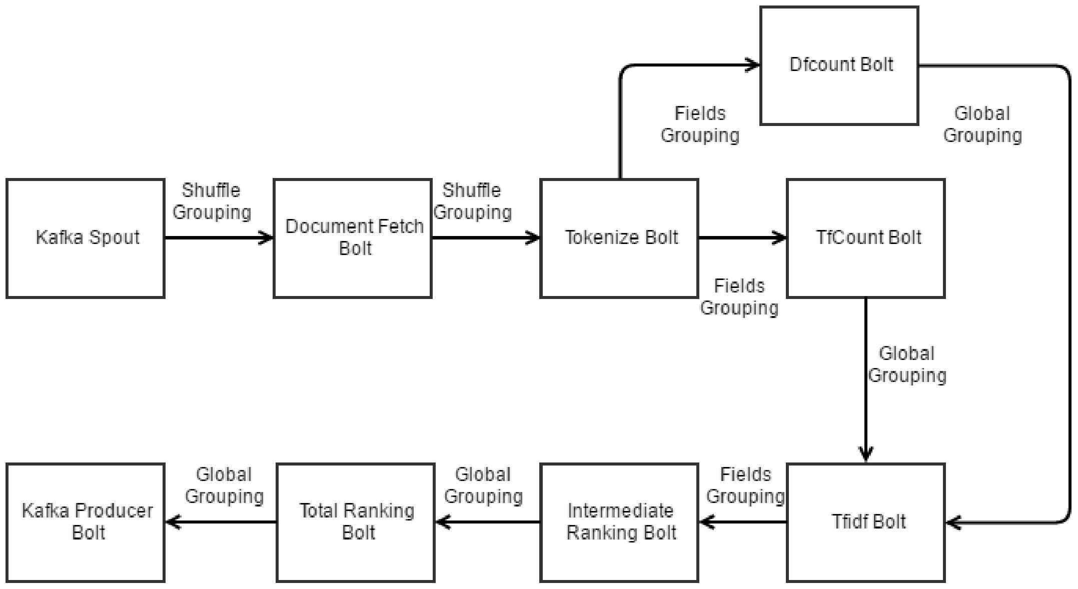


Figure 4.10: Trend Analysis

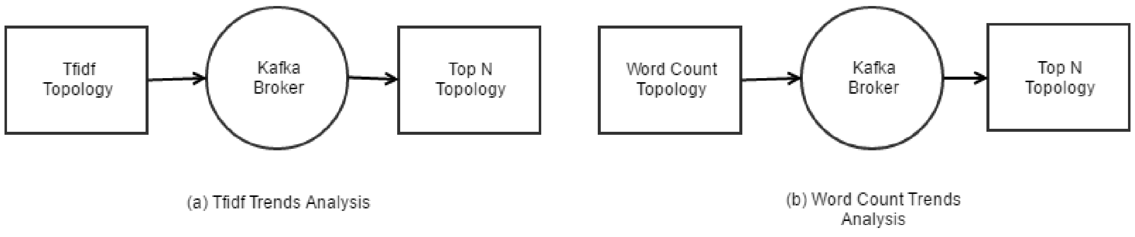


Figure 4.11: Trend Analysis with Components

4.4.3. Data Visualization Components

Data visualization components are implemented by D3.js and Google Charts. D3.js allows users to build visualization by themselves. Google Charts: To use Google Charts, developers only need to assign data set to the x-axis and y-axis. Google Charts provides function arrayToDataTable which can convert array to the desired visualization format. Compared to Google Charts, Baidu Echarts needed developers to separate data and assign it to x-axis and y-axis. In terms of map visualization, Echarts only provide Chinese map, which cannot satisfy the requirements of this project. The data of this project is sent from the server continuously.

To address real-time data, there are four options: polling, long polling, WebSocket and Server-Sent Event. Polling and long polling are techniques that the client side sends requests to the server side periodically. They are the easiest to implement, even though they are costly. WebSocket and Server-Sent Event (SSE) are more popular, in which WebSocket allows both clients and servers to send messages to each other, while SSE, as its name shows, is only responsible for sending messages from the server side to the client side. For only the server side to send data to the front-end periodically, the project exploits SSE to update stream data, which only need to set Content-Type to text/event-stream in HTTP header. In order to update data on graphs**,** the website checks whether data updated every second with window.setInterval function. The following are visualization components.

- **Map:** The map component is implemented by d3.js. JavaScript code check stream data update periodically.
- **Word cloud:** The word cloud is implemented by d3.js as well. The visualization function only needs key and value. The key will be presented on the screen and value decides the font size of key. The positions and angles of words are randomly assigned by algorithm. Word cloud library [79] is used in the project.
- **Pie chart:** The pie chart is implemented with Google Chart

- **Bar chart:** The bar chart is implemented with Google Chart

4.4.4. Data Persistence Components

Data persistence is an option in the project. As Kafka brokers store all messages as logs, there is no need to implement additional data persistence components.

4.5. Component Management

As mentioned in the data visualization section, webpages are the manifestation of data visualization, whereas its server side is responsible for managing components. By submitting selection forms, users request back-end calling corresponding components. In the index page, users can customize data collection, data processing, and data visualization components. Specifically, users can combine several Storm topologies into one data processing component.

To manipulate components, Python's subprocess module is involved. The project runs data collection and data processing components in the system background with function Popen of module subprocess [80]. The process IDs of component threads are stored in global variables, once users call function clear_threads, the threads will be terminated [81].

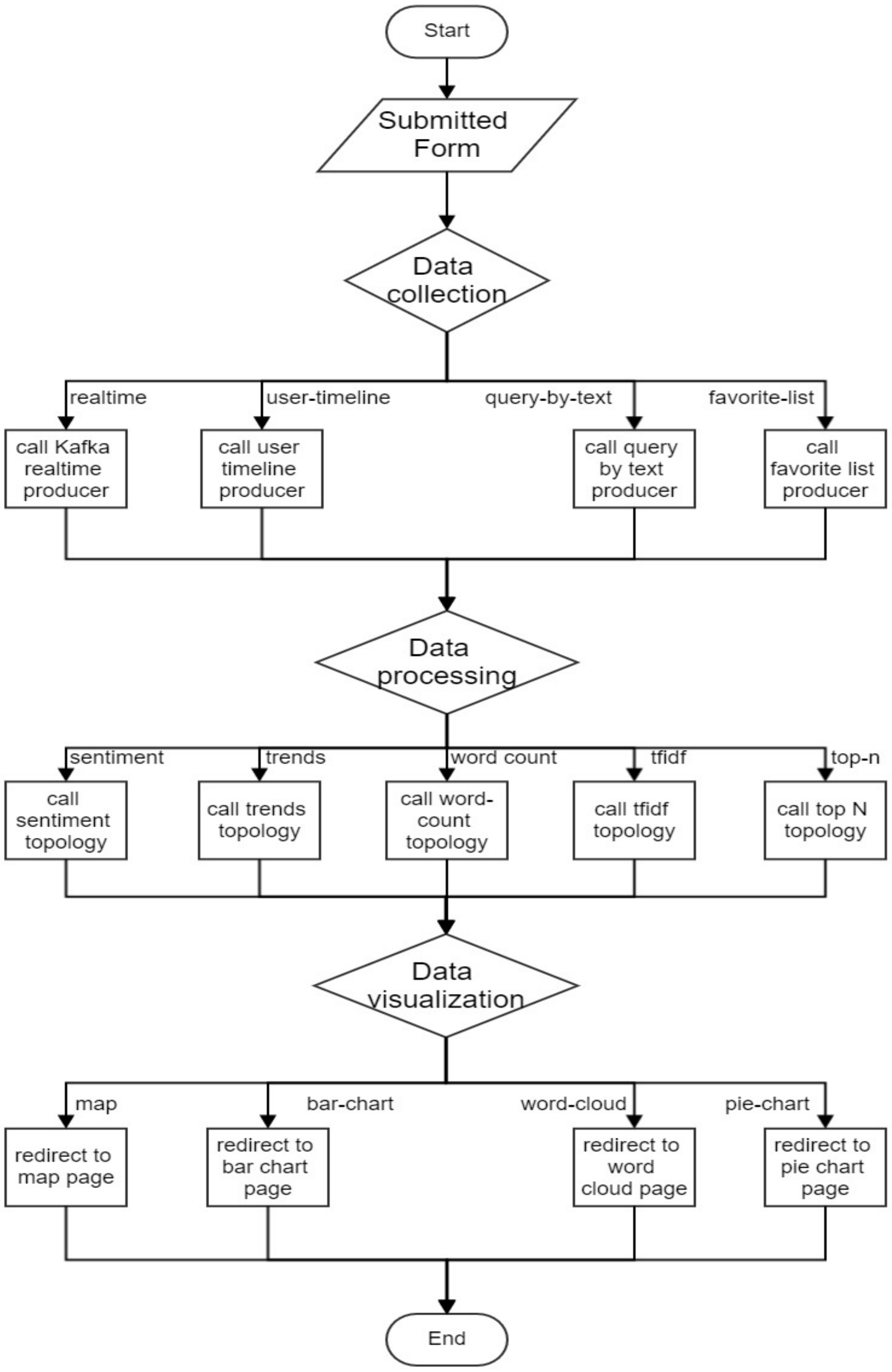


Figure 4.12: Web Page Flow Chart

4.6. Data Format

In order to allow data processing components to be connected freely, this project defines a general data format. In the system, data can only be transformed with single string in Apache Kafka, every message is a string. The system defines two types of messages, the first type is shown in Table 4.1, it contains at least 5 fields, and "DELIMITER" is used as separator of each data field. The first five fields are fixed: tweet contents, screen names, created time, latitude and longitude, and country codes. If corresponding data are not available, it will be set as "n/a". Other fields will be added after in sequence. Furthermore, the second type consists of only two fields and fields are split with '|', as Table 4.2 shows.

Table 4.1: Type 1 Message String

Tweet Content	Screen Name	Created Time	Latitude, Longitude	Country Code (ISO alpha-3)	Sentiment Value

Table 4.2: Type 2 Message String

Keyword	Value

4.7. Challenges and Solutions

Since the system involves several different technologies, integrating all technologies is the main challenge. The system involves web servers, data crawlers, and distributed frameworks which should be managed reasonably. We solve this problem by integrating all technologies with Kafka and treat Kafka brokers as the centre of the whole system. The functions of the entire system should be divided into reasonable pieces, depending on the size of the component the system will be more flexible but more expensive. Every component is integrated with Kafka. Different components communicate with other components by indicating the same topic of Kafka. They are connected by message transmission. In order to allow components to transmit messages universally, we define a unified data form which allows messages to be processed properly in different components.

The other challenge is that the architecture of the system should be modified with an increase in the complexity of system requirements. In the process of designing the system, we improve the structure of the system step by step. At first, we only want to implement a single system which can analyze the sentiment of real-time tweets. Thus, the simple linear architecture is implemented with Storm, Redis, Flask, and D3.js. However, this system only allows users to receive passive analysis results. In order to increase the interactivity of the system, we propose message feedback architecture which allows users to control the behaviours of data collection function. At this time, Kafka is integrated into the system. When the data analysis requirements increase, data processing and data collecting components become too complex to modify. Therefore, we decouple the functions into different components and connect them to compose an entire system. As a result, we implement the Kafka centric architecture. We also implement Lambda architecture with Hadoop, Storm, and Kafka. The reason is that Lambda architecture is a combination of batch processing and stream processing architectures, we employ it to improve the functionality of Kafka centric architecture

4.8. Summary

This chapter describes the details of the implementation of component-based distributed text mining system. Data collection components are implemented with Tweepy library and integrated with Kafka producer, data processing components which contain Kafka producer and consumer are implemented with Apache Storm, and data visualization components are implemented with D3.js or Google Charts. All components are managed with index web page. To simplify the data processing steps and decrease the volume of data which should be transmitted, the system accepts two types of message strings. Experiments, evaluation and results are presented in Chapter 5.

Chapter 5

Evaluation and Results

This chapter presents evaluation results of this component-based distributed text mining system. Section 5.1 presents the environments of experiments and evaluation, in local and cloud. The deployment methods are presented in Section 5.2. Section 5.3 evaluates the throughput of Twitter Stream APIs based on Tweepy and Twitter4j. In Section 5.4, scalability of the system is evaluated by testing the performance of data processing components, availability is evaluated in Section 5.5. Finally, several cases are studied in Section 5.6.

5.1. Testing Environment

All experiments are deployed with local virtual machines or cloud computing platforms. To run the system locally, we apply Vagrant [82] and VirtualBox [83] to host virtual machines of Ubuntu 14.04. In the cloud, we chose AWS EC2 as the cloud computing platform to build the cluster of Ubuntu machines.

5.1.1. Vagrant and VirtualBox

The project engages Vagrant and VirtualBox to deploy the local experimental environment. Vagrant is a virtual machine management tool and VirtualBox is a virtualization product. The project exploits synced folder which shares folders between the host and client machine, and forwarded port which map host port to guest port with aid from Vagrant. Table 5.1 shows the configuration file of virtual machine.

Table 5.1: Configuration File of Vagrant with Comments

```
Vagrant.configure(2) do |config|

  # indicates used image
  config.vm.box = "ubuntu/trusty64"

  # forward guest port 5000, 8080 to host port 5000, 8080
  config.vm.network "forwarded_port", guest: 5000, host: 5000
  config.vm.network "forwarded_port", guest: 8080, host: 8080

  # indicates shared folder, defaultly, the guest /vagrant folder map with the root folder of
vagrantfile
  config.vm.synced_folder "../data", "/vagrant_data"

  # set the memory size, default as 1024 MB
  config.vm.provider "virtualbox" do |vb|
    vb.memory = "2048"
  end
end
```

5.1.2. AWS EC2

Both AWS EC2 and Azure HDInsight can be used to host servers in the cloud. Although HDInsight provides automatic configuration, since HDInsight restricts the version of Storm and Kafka, we use AWS EC2 to host clusters in the cloud. Furthermore, the cluster of HDInsight is based on Apache Ambari [84] which takes at least 8 gigabytes of memory per machine. In the experiments, the system deploys three machines to build the cluster. These machines should be in the same local area network. The details of virtual machines on AWS EC2 are listed in Table 5.2. Due to the limitation of AWS free tier, the system can only apply t2.micro type instance whose details are listed in the Table 5.3. The following is the table of details of cluster.

Table 5.2: Details of Cluster

Hostname	Private IP Address	Operating System	Character	Instance Type
stormnode1	172.31.22.225	Ubuntu 14.04 TLS	Nimbus/ZooKeeper	t2.micro
stormnode2	172.31.24.92	Ubuntu 14.04 TLS	Supervisor/ZooKeeper	t2.micro
stormnode3	172.31.25.230	Ubuntu 14.04 TLS	Supervisor/ZooKeeper	t2.micro

Table 5.3: Details of t2.micro

Instance Type	Images	vCPU	Memory Size	Instance Storage
t2.micro	Ubuntu	1	1 GB	EBS

5.2. Deployment

To centrally manage all packages, ZooKeeper, Storm and Kafka are installed in the folder /opt. Section 5.2.1 has listed methods of installation of main dependencies, coupled with more details listed in Appendix A. The deployments of ZooKeeper, Storm and Kafka are more complex, thus deployments are described in detail in separate sections. Moreover, their configuration files are presented in Appendix B.

5.2.1. Single Machine

To build the system with a single machine, there is no need to change the network configuration of machines. Both Kafka and Storm can be run in standalone mode. In the standalone mode, it is unnecessary to change any configuration files. In this project, we apply Vagrant to manage virtual machine. Table 5.4 shows the basic information of virtual machine.

The following are steps of installing dependencies of this project:

1. **Installing JDK:** As long as JDK is the basic dependency of all Java programs, the project is built based upon JDK 1.7.
2. **Installing Maven:** “Apache Maven is a software project management and comprehension tool” [85]. Instead of keeping Java libraries locally, the system uses Apache Maven to manage the dependencies of Java which are declared in the pom.xml file.
3. **Installing Python dependencies:** All python dependencies are listed in the requirements.txt file and can be installed with python pip tool.
4. **Installing ZooKeeper and setting configurations:** Both clusters of Storm and Kafka are based upon ZooKeeper. We install Zookeeper by downloading ZooKeeper package from website [86] and unzipping it.
5. **Downloading Kafka and Storm packages:** To download Kafka and Storm software, we just download packages [87, 88] and unzip them.

Table 5.4: System Information of Single Machine

OS version	vCPU	Memory
Ubuntu 14.04 LTS	6	2048 MB

Server Configuration

Before setting distributed cluster, it is necessary to configure servers first. To decrease the difficulties of debugging, we map hostname to IP address. When client first consult the name of the machine, it will consult /etc/hosts first. If the client does receive results, it will send queries to DNS server. Thus, in this project we save the hostnames and map them with their private IP address. Additionally, the hostnames can be mapped with their public IP address, however the instance of AWS EC2 will change its public IP address when it restarts.

5.2.2. Deployment of ZooKeeper Cluster

ZooKeeper is the basis of both Kafka and Storm, thus building ZooKeeper cluster is the first phase. Every ZooKeeper server is a thread on machines and there is no restriction to how many ZooKeeper clusters can be run on the same machine. Standalone mode is a special case which runs all ZooKeeper threads on a single machine. However, every ZooKeeper matches with one configuration file and one dataDir which stores data and logs of the ZooKeeper server. Normally, to decrease the loads of ZooKeeper log files are stored in the path of dataLogDir separately. Considering Zookeeper cluster is available when half of Zookeeper servers are available, ZooKeeper cluster is built upon odd number of ZooKeeper servers. Furthermore, configuration files are defined in Appendix B. The following are steps of deploying ZooKeeper:

1. Configuring hostnames by mapping private IP addresses with hostnames in /etc/hosts file, then testing the connectivity of nodes by pinging hostname.
2. Downloading and extracting ZooKeeper package to specified folder
3. Modifying configuration file, zoo.cfg.
4. Building myid file under dataDir’s folder.
5. Setting environment variables.

5.2.3. Building Storm Cluster

To configure Storm, only storm.yaml should be changed. Details are presented in Appendix B3. In the configuration file, IP addresses or hostnames of ZooKeeper servers and nimbuses of Storm should be indicted. If ZooKeeper uses default port 2888, there is no need to indicate port of ZooKeeper servers. Otherwise, Storm needs to indicate port number. Aside from nimbuses, other ZooKeeper servers can be supervisors. By default, each supervisor has four ports, 6700, 6701, 6702, and 6703, which means each supervisor can run four workers. More details about Storm configuration are presented in Appendix C.

For the sake that ZooKeeper cluster is the basis of Storm cluster, the ZooKeeper servers should run first. ZooKeeper clients are integrated into Storm. To run ZooKeeper, zkServer.sh is the shell script which can be used to start, stop, or check status of ZooKeeper threads. Once cluster runs zookeeper thread, zookeeper will elect a leader and followers.

The followers are Storm commands：

- storm nimbus: ‘storm nimbus’ runs Storm nimbus thread.
- storm supervisor: ‘storm supervisor’ command runs Storm supervisor thread.
- storm ui: ‘storm ui’ command runs Storm UI on http://localhost:8080, in default.
- storm jar: ‘storm jar’ command submits jar package to Storm clusters.

5.2.4. Building Kafka Cluster

Similar to Apache Storm, the cluster of Kafka is based on ZooKeeper cluster as well. A single ZooKeeper cluster can run multiple Kafka brokers. There is no restriction on the physical location of Kafka brokers, Kafka brokers can be invoked on ZooKeeper servers or remote machines. In contrast to the deployment of ZooKeeper and Storm, running Kafka servers only need to configure the server.properties to replace the line of zookeeper.connect with designated Zookeeper server list. Furthermore, it is possible to run multiple Kafka servers with the creation of multiple server.properties files. Additionally, more details and comments of configuration items can be found in the Appendix D.

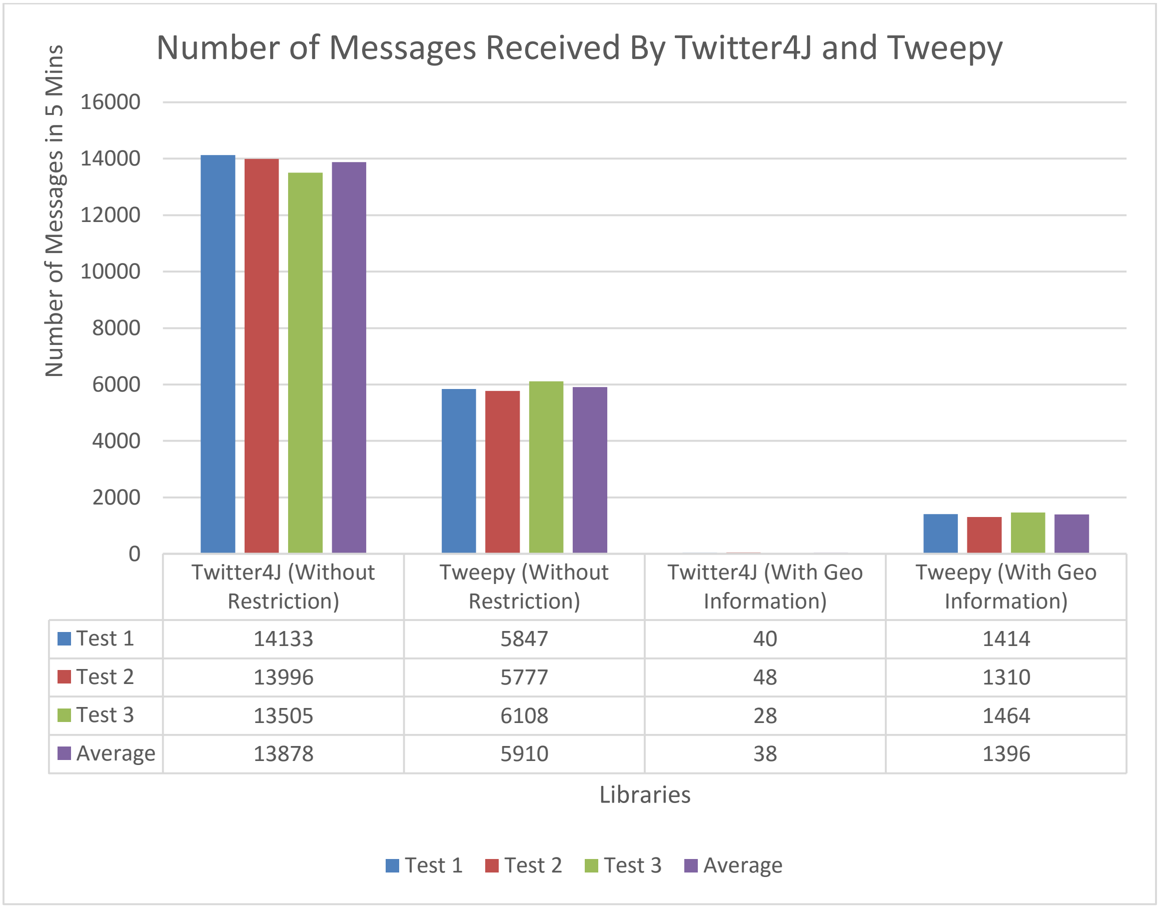
5.3. Twitter Stream Throughput Evaluation

We evaluate the rates of Twitter stream, namely how many messages can be received from Twitter. To design the experiment, we select two libraries of Twitter API, Twitter4J and Tweepy. As mentioned before, all data collection components integrate Twitter libraries with Kafka producer, thus the project handles a simple Kafka consumer to count the number of messages.

We evaluate the efficiency of Twitter APIs by calculating the number of messages which are fetched within 1 minute and 5 minutes respectively. The results are presented in Figure 5.1, which indicates that Twitter4J can fetch more Twitter messages than Tweepy within the same period, while Tweepy can more easily acquire messages with geographic information. There are two methods of retrieving tweets which contain geographic information and country name. The first one is filtering messages containing geographic information from all received messages. The other is listening to the Twitter stream with location filter directly. Due to the fact that the location filter of Twitter4j does not work properly, the evaluation of Twitter4j applies the first method. As a result, the first method can only fetch few messages. However, Tweepy practices the second method and receives more than 300 messages per minute.

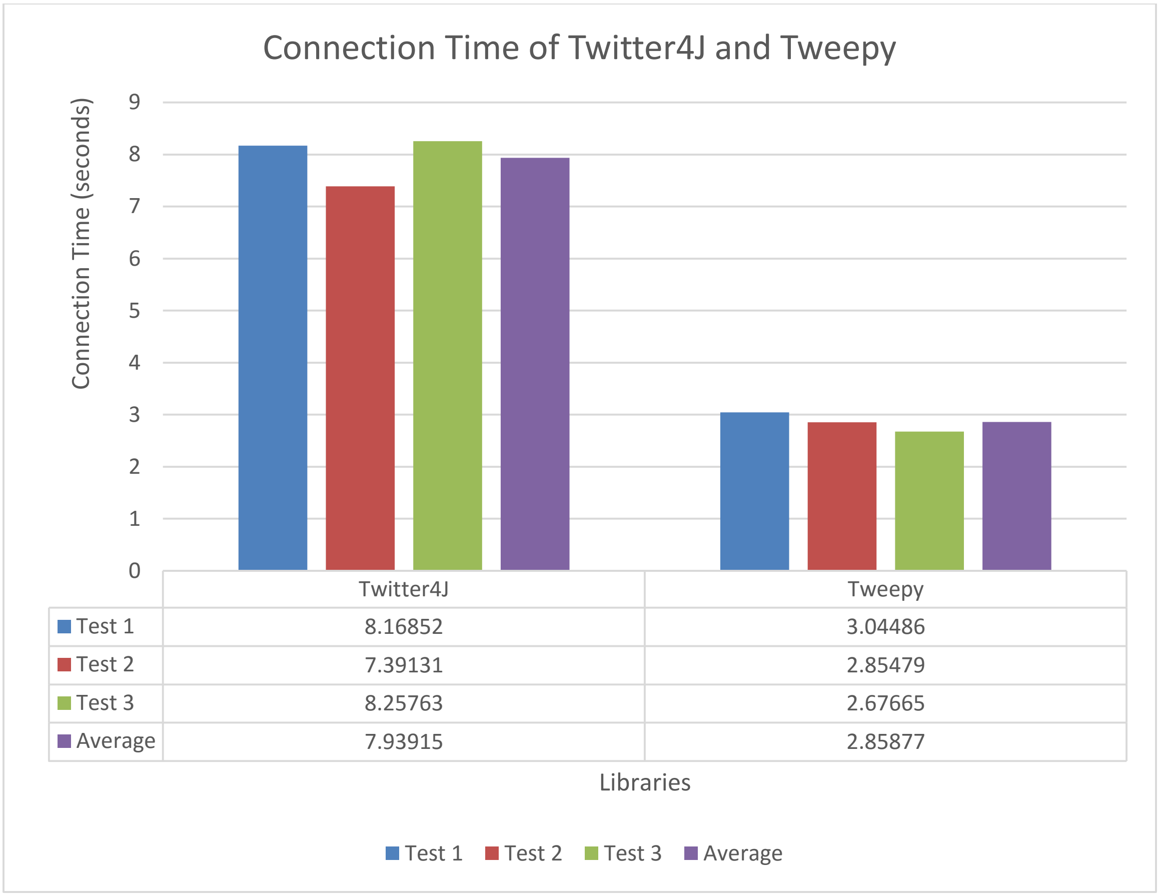
In the system, geographic information is important for data visualizations which require the locations of the messages, such as map visualization. Although, Twitter4J API can collect more generic tweets than Tweepy, the performance of its geographic information collection is abysmal, we deploy Tweepy API in the system to collect data.

The second experiment calculates the connection time of these APIs. As Figure 5.2 shows, the connection time of Twitter4J is more than twice that of Tweepy. The shorter connection time means better connectivity of the API. It is obvious that Tweepy connects to Twitter faster.



	Twitter4J (Without Restriction)	Tweepy (Without Restriction)	Twitter4J (With Geo Information)	Tweepy (With Geo Information)
🟦 Test 1	14133	5847	40	1414
🟥 Test 2	13996	5777	48	1310
🟩 Test 3	13505	6108	28	1464
🟪 Average	13878	5910	38	1396

Figure 5.1: Number of Messages Received By Twitter4J and Tweepy



	Twitter4J	Tweepy
🟦 Test 1	8.16852	3.04486
🟥 Test 2	7.39131	2.85479
🟩 Test 3	8.25763	2.67665
🟪 Average	7.93915	2.85877

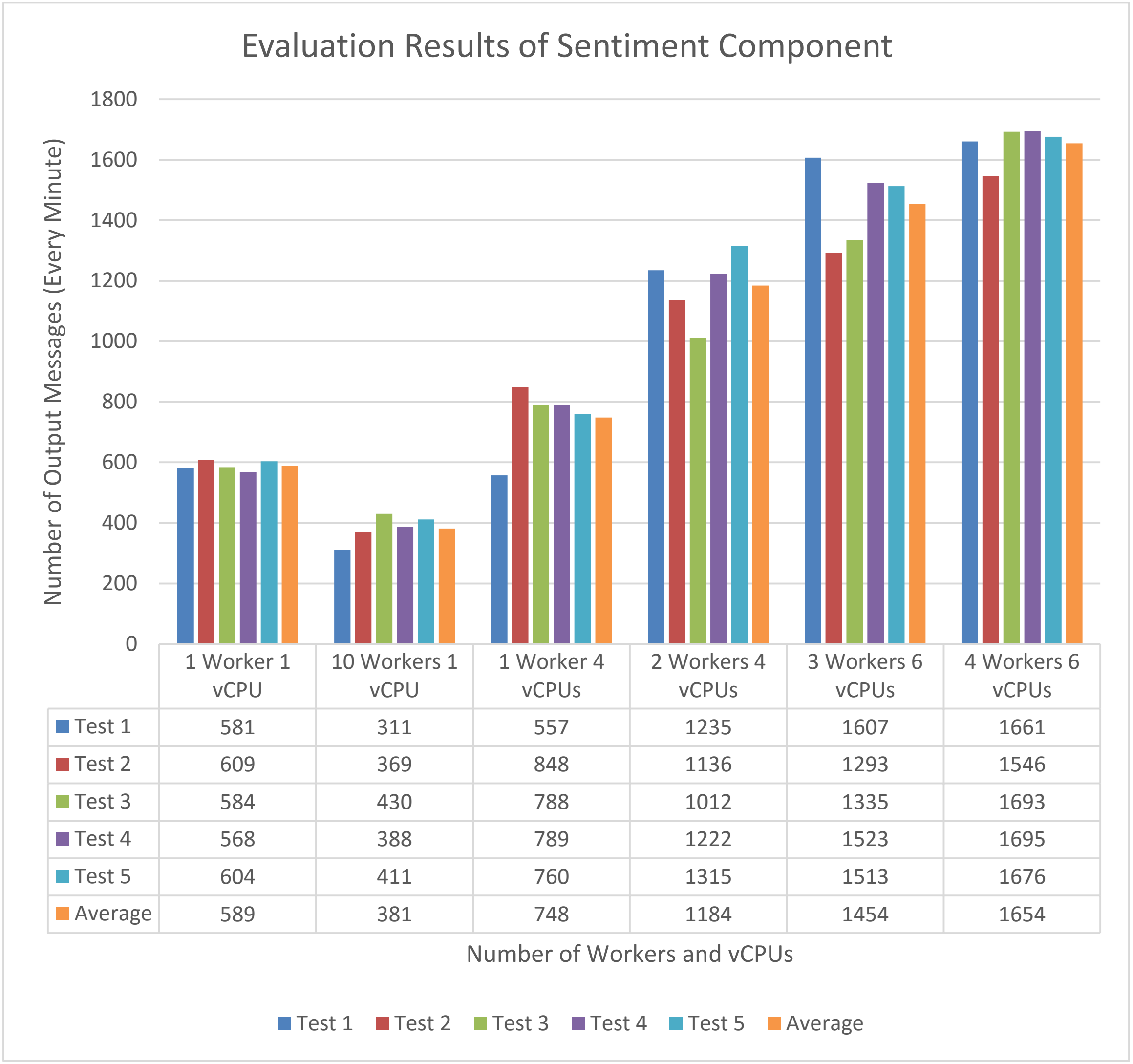
Figure 5.2: Connection Time of Twitter4J and Tweepy

5.4. Scalability of Data Processing Components

We evaluate the scalability of data processing components by modifying the number of worker threads of data processing components.

5.4.1. Evaluation of Sentiment Component

To simplify the evaluation process, the parallelism of every bolt will not be modified, while the results are evaluated by changing the number of workers and virtual CPU. At first, by analyzing the first two results, increasing the number of workers leads to the decrease of output rates. The main reason is that a single CPU cannot support multiple worker threads, in which one worker thread needs at least 120% usage of one core. In order to achieve more accurate results, the number of CPUs should be controlled 1.2 times beyond the number of workers.



	1 Worker 1 vCPU	10 Workers 1 vCPU	1 Worker 4 vCPUs	2 Workers 4 vCPUs	3 Workers 6 vCPUs	4 Workers 6 vCPUs
Test 1	581	311	557	1235	1607	1661
Test 2	609	369	848	1136	1293	1546
Test 3	584	430	788	1012	1335	1693
Test 4	568	388	789	1222	1523	1695
Test 5	604	411	760	1315	1513	1676
Average	589	381	748	1184	1454	1654

Figure 5.3: Evaluation Results of Sentiment Component

As the results in Figure 5.3 show, raising the number of workers increases the performance of sentiment component dramatically. When the number of virtual CPUs cannot satisfy the computing requirements of the component, increasing number of workers cannot enhance the performance of the component, it may even decrease the number of output messages. For example, comparing the first two rows of Figure 5.3, the number of workers is added from 1 to 10, the average outputs decrease from 589 to 381 messages per minute. By comparing the first row to the third row, which keeps the same workers and increases the number of virtual CPUs, the average message output increases from 589 to 748. We can conclude that one virtual CPU cannot cover the computing resources of one Storm sentiment worker. In the third and fourth row, we increase the number of workers, with 4 virtual CPUs, the average output increases from 748 to 1184 per minute. Likewise, in rows five and six, with 6 virtual CPUs, increasing worker number raises the output from 1454 to 1654 per minute. When computing resources are adequate, increasing the number of workers leads to increased output as well. As the system applies Tweepy to collect data, whose average data fetching rate is 1071 messages per minute, our system can completely process all collected messages, with more than 4 virtual CPUs and 2 worker threads.

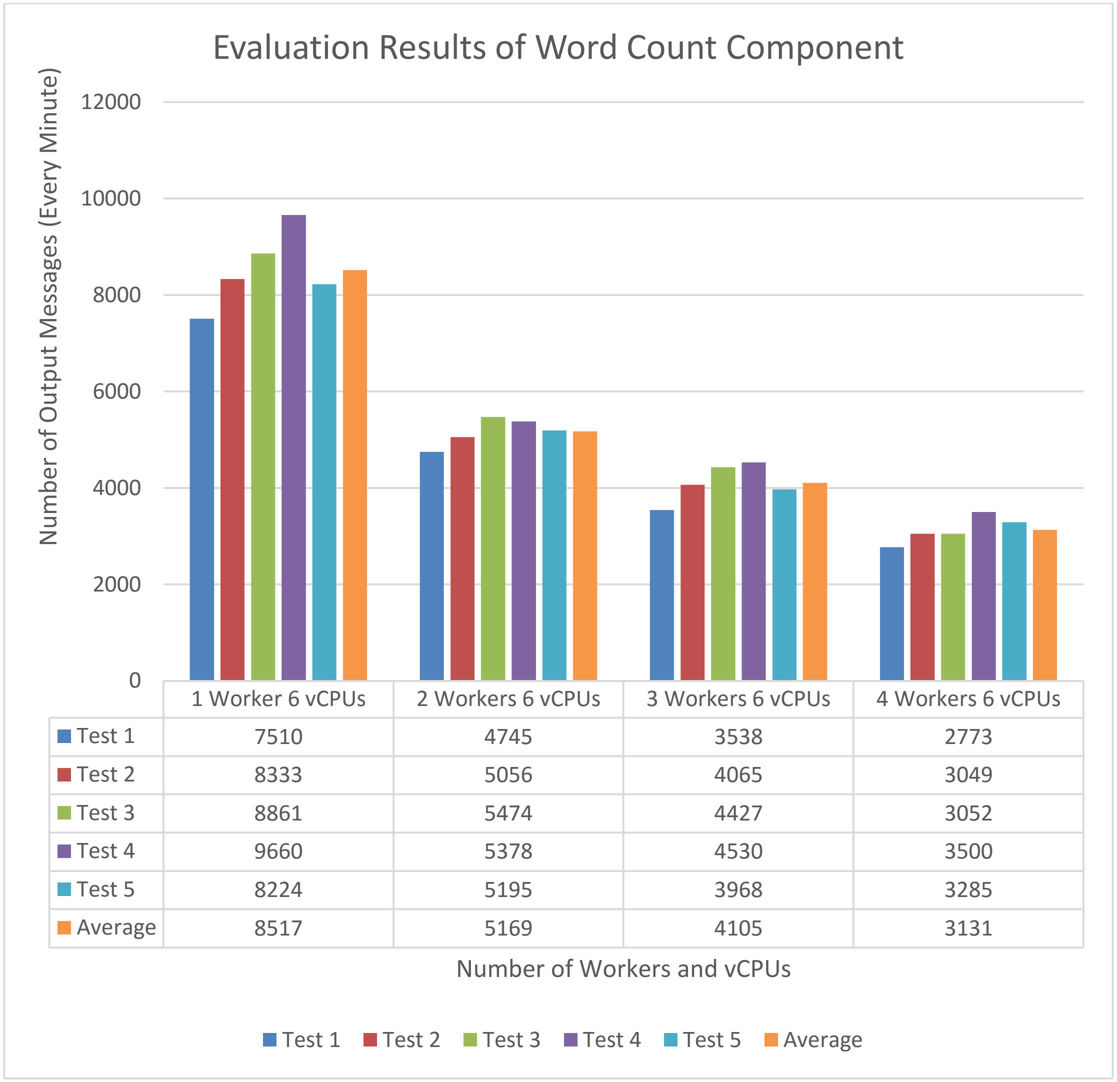
5.4.2. Evaluation of Word Count Component

The word count calculation is a simple task which does not involve complex calculations. With the environment of six virtual cores, applying more worker threads does not bring better performance, even diminishing the number of output messages.

In Figure 5.4, the number of virtual CPUs is fixed at 6, however, with the number of workers increasing, the average output decreases from 8517 to 3131 messages every minute. Thus we assume that the word count algorithm only fits the single thread environment. To demonstrate this assumption, we constructed another experiment which manipulated the parallelism of each bolt or spout.

As Table 5.5 presents, all the results are evaluated with three worker threads, in which the parallelism numbers correspond to Kafka spout, word split bolt, word count bolt, report bolt, in sequence. By adjusting the number of parallelism of word split and word count bolt, the performance of word count component reaches the peak when all bolts run with a single thread. It is easy to conclude that the word count algorithm performs best with a single thread.

We can conclude that the word count component can process all real-time data with one worker thread.



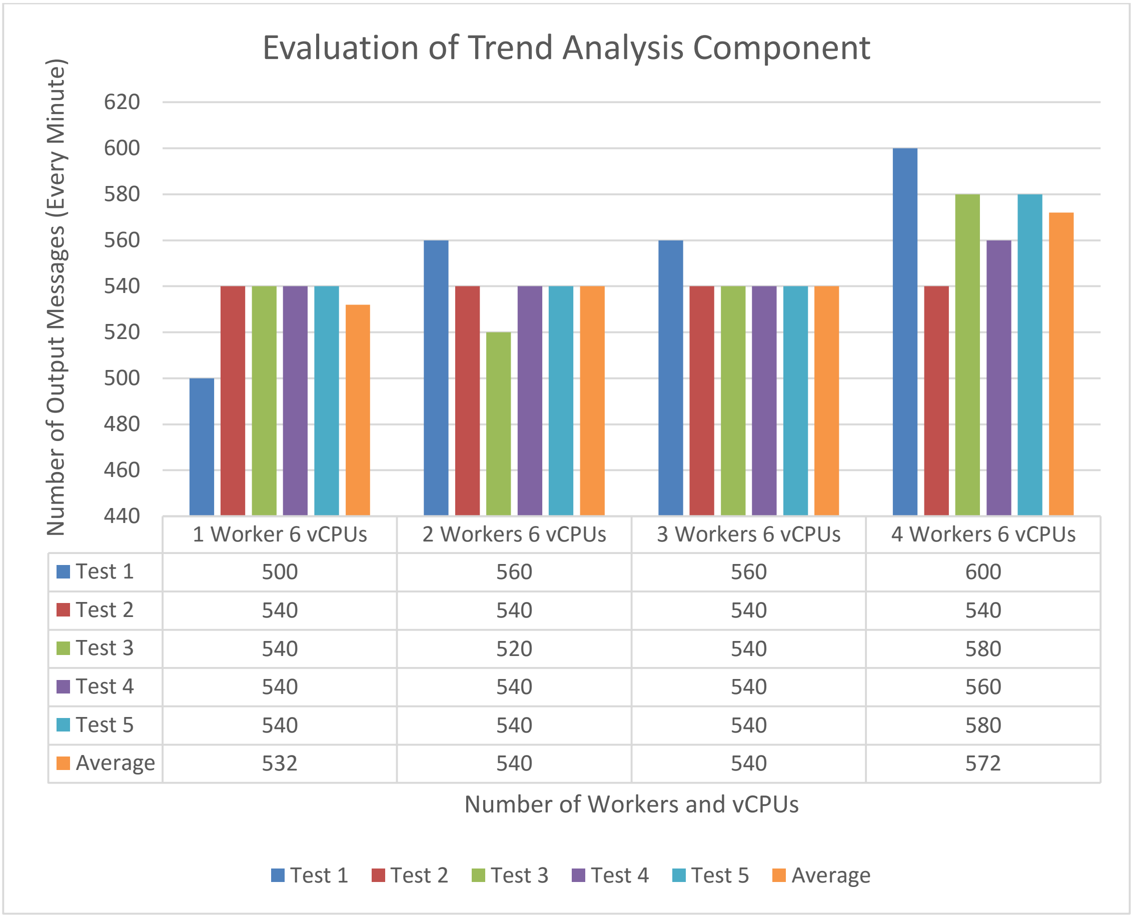
	1 Worker 6 vCPUs	2 Workers 6 vCPUs	3 Workers 6 vCPUs	4 Workers 6 vCPUs
Test 1	7510	4745	3538	2773
Test 2	8333	5056	4065	3049
Test 3	8861	5474	4427	3052
Test 4	9660	5378	4530	3500
Test 5	8224	5195	3968	3285
Average	8517	5169	4105	3131

Figure 5.4: Evaluation Results of Word Count Component (1)

Table 5.5: Evaluation Results of Word Count Component (2)

Number of Worker s	vCPU	Parallelism of Bolts and Spout				Number of Output Messages (Every Minute)					
						Test 1	Test 2	Test 3	Test 4	Test 5	AVG
3	6	1	1	1	1	6131	6877	8531	7590	7836	7393
3	6	1	1	10	1	4455	4721	4524	5201	4272	4634
3	6	1	10	1	1	4014	4746	4877	5614	5350	4920

5.4.3. Evaluation of Trend Analysis Component

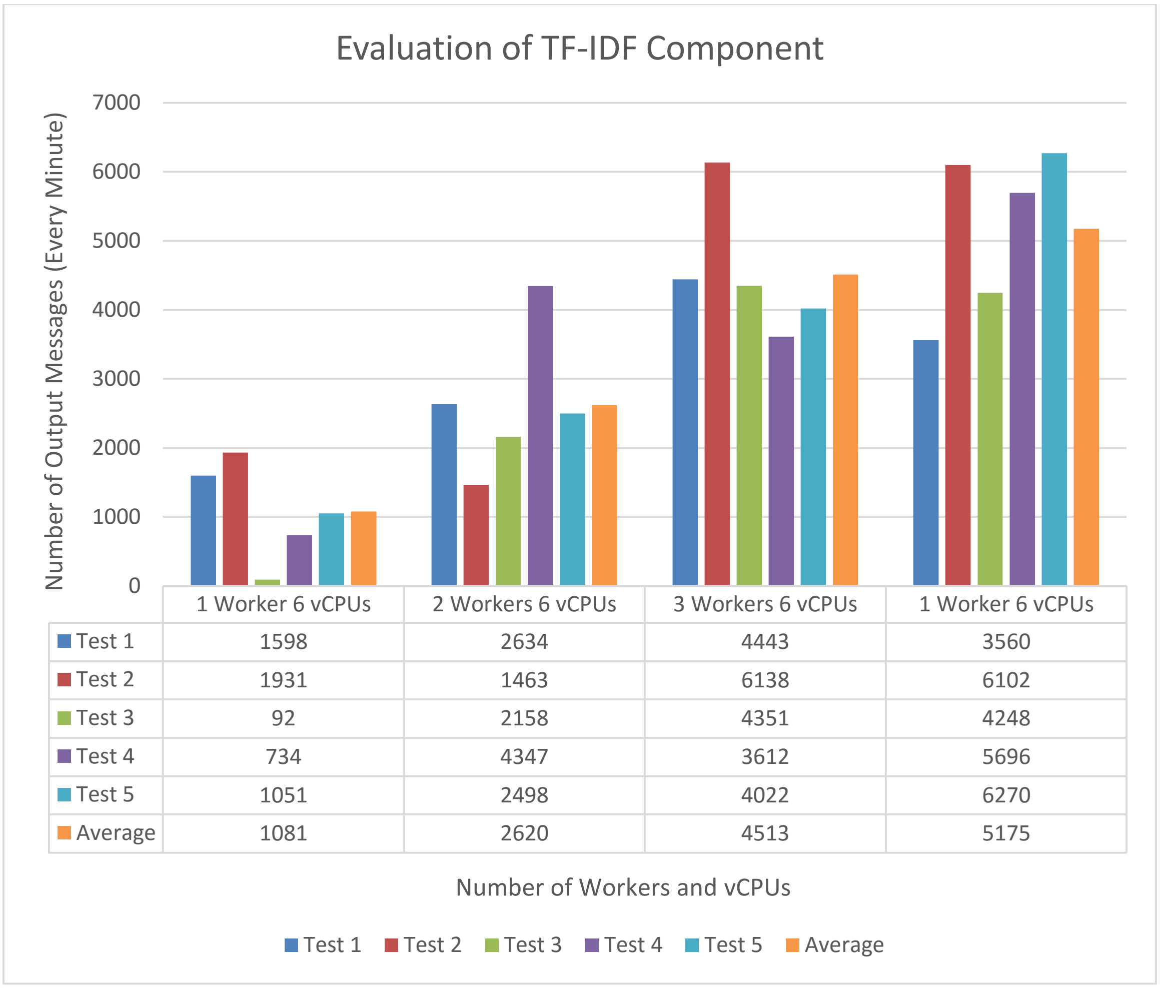


	1 Worker 6 vCPUs	2 Workers 6 vCPUs	3 Workers 6 vCPUs	4 Workers 6 vCPUs
Test 1	500	560	560	600
Test 2	540	540	540	540
Test 3	540	520	540	580
Test 4	540	540	540	560
Test 5	540	540	540	580
Average	532	540	540	572

Figure 5.5: Evaluation of Trend Analysis Component

Due to the implementation of trend analysis which returns top 20 results each time and only updates at most 10 results every second, the evaluation results of trend analysis would be multiples of 20 and around 600. Therefore, changing the number of workers will not influence the output rate of trend analysis topology. In Figure 5.5, the number of workers increases with the number of experiments. However, the output of each experiment is almost the same. Thus the results of Figure 5.5 demonstrate that the output rates of trend analysis are not influenced by the number of workers.

5.4.4. Evaluation of TF-IDF



	1 Worker 6 vCPUs	2 Workers 6 vCPUs	3 Workers 6 vCPUs	1 Worker 6 vCPUs
Test 1	1598	2634	4443	3560
Test 2	1931	1463	6138	6102
Test 3	92	2158	4351	4248
Test 4	734	4347	3612	5696
Test 5	1051	2498	4022	6270
Average	1081	2620	4513	5175

Figure 5.6: Evaluation of TF-IDF Component

As mentioned in the implementation chapter, TF-IDF component is a part of trend analysis component. It extracts messages from the URLs of tweets and generates the TF-IDF score of each word, thus its output depends on the content of URLs. If the webpage contents of

URLs are longer, the size of output messages will be larger. Thus, the results of TF-IDF component are not stable. In the first row of Figure 5.6, the result of test 3 is 92, which means there are few contents in this test. By analyzing the average results of Figure 5.6, it is easy to conclude that with the increase in number of workers, the number of output messages raises as well. The TF-IDF component can be scaled up by adding more worker threads to address more data.

5.5. Availability

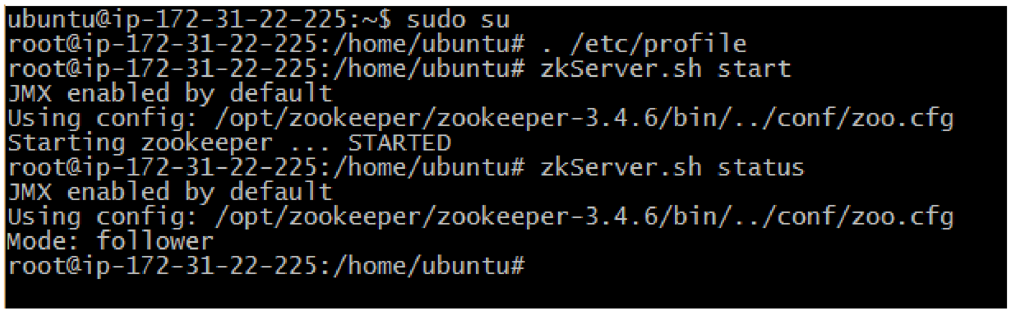
We designed two experiments to test the availability of Apache Kafka and Apache Storm. As a result of the visualization website being built upon the clusters of Kafka and Storm, if both Kafka and Storm equip high availability, the website is highly available as well.

5.5.1. Availability of Kafka

In the experiment, we prepare four machines containing three ZooKeeper servers and one Kafka server. With this experiment, we can conclude that the availability of Kafka is based on ZooKeeper cluster, when more than half of ZooKeeper machines disconnect, the Kafka broker will disconnect as well. In order to guarantee the availability of Kafka, it is necessary to increase the number of machines in ZooKeeper clusters. The following are steps of experiment:

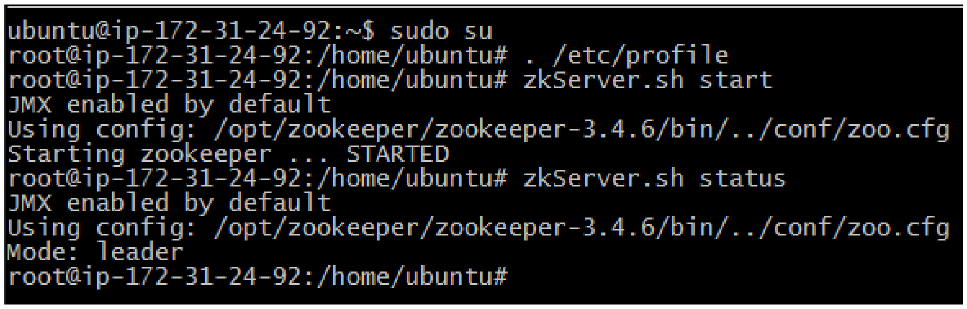
1. Getting root authority and active environment variable.
2. Running zkServer.sh script to start ZooKeeper threads on all ZooKeeper servers.
3. Checking status of ZooKeeper servers: Command ‘zkServer.sh status’ is used to check the status of Zookeeper server. As Figure 5.7, 5.8 and 5.9 display, node1 and node 3 are followers, while node 2 is the leader.

4. Checking whether machine that hosts Kafka server can connect to three ZooKeeper machines: In Figure 5.10, we use ping command to test the connection between three machines.
5. Configuring server.properties file: only change line ‘zookeeper.connect’ as following,‘zookeeper.connect=stormnode1:2181,stormnode2:2181,stormnode3:2181’.
6. Using tmux to open multiple console on the machine hosting Kafka server
7. Running Kafka broker to connect ZooKeeper cluster
8. Running Kafka console consumer and producer to check the connection
9. Stopping one ZooKeeper server: at this time, as Figure 5.11 shows, the consumer and producer can still communicate.
10. Stopping one more ZooKeeper server: As Figure 5.12 shows, the consumer loses connection to ZooKeeper cluster.
11. Restarting one ZooKeeper server: Kafka consumer becomes available again.



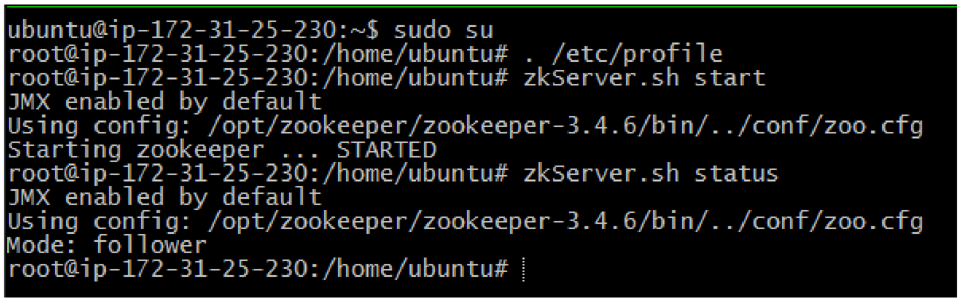
```
ubuntu@ip-172-31-22-225:~$ sudo su
root@ip-172-31-22-225:/home/ubuntu# . /etc/profile
root@ip-172-31-22-225:/home/ubuntu# zkServer.sh start
JMX enabled by default
Using config: /opt/zookeeper/zookeeper-3.4.6/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
root@ip-172-31-22-225:/home/ubuntu# zkServer.sh status
JMX enabled by default
Using config: /opt/zookeeper/zookeeper-3.4.6/bin/../conf/zoo.cfg
Mode: follower
root@ip-172-31-22-225:/home/ubuntu#
```

Figure 5.7: Stormnode1 Screen Shot



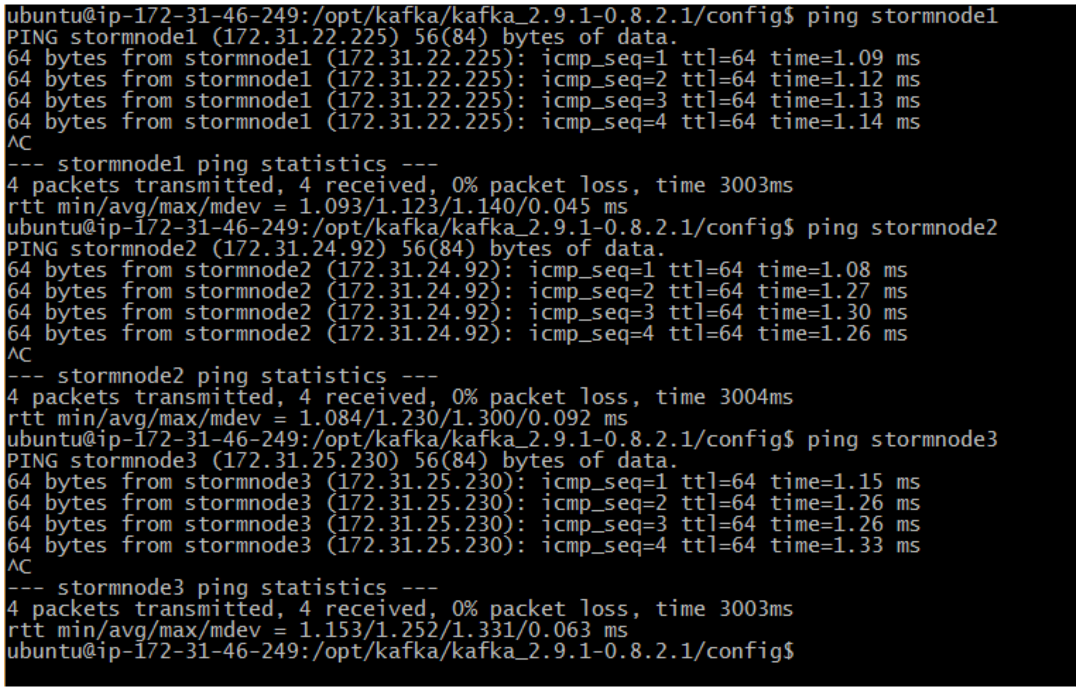
```
ubuntu@ip-172-31-24-92:~$ sudo su
root@ip-172-31-24-92:/home/ubuntu# . /etc/profile
root@ip-172-31-24-92:/home/ubuntu# zkServer.sh start
JMX enabled by default
Using config: /opt/zookeeper/zookeeper-3.4.6/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
root@ip-172-31-24-92:/home/ubuntu# zkServer.sh status
JMX enabled by default
Using config: /opt/zookeeper/zookeeper-3.4.6/bin/../conf/zoo.cfg
Mode: leader
root@ip-172-31-24-92:/home/ubuntu#
```

Figure 5.8: Stormnode2 Screen Shot



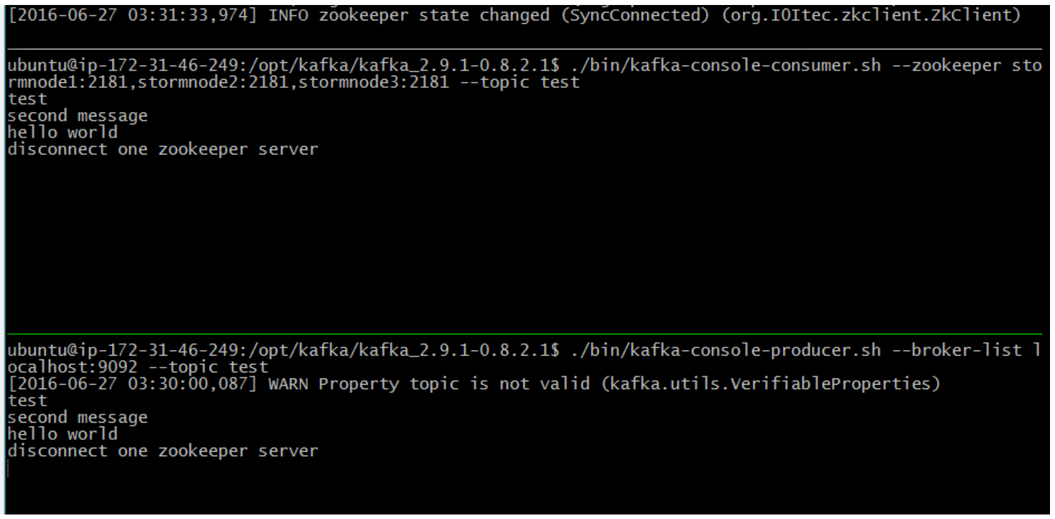
```
ubuntu@ip-172-31-25-230:~$ sudo su
root@ip-172-31-25-230:/home/ubuntu# . /etc/profile
root@ip-172-31-25-230:/home/ubuntu# zkServer.sh start
JMX enabled by default
Using config: /opt/zookeeper/zookeeper-3.4.6/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
root@ip-172-31-25-230:/home/ubuntu# zkServer.sh status
JMX enabled by default
Using config: /opt/zookeeper/zookeeper-3.4.6/bin/../conf/zoo.cfg
Mode: follower
root@ip-172-31-25-230:/home/ubuntu#
```

Figure 5.9: Stormnode3 Screen Shot



```
ubuntu@ip-172-31-46-249:/opt/kafka/kafka_2.9.1-0.8.2.1/config$ ping stormnode1
PING stormnode1 (172.31.22.225) 56(84) bytes of data.
64 bytes from stormnode1 (172.31.22.225): icmp_seq=1 ttl=64 time=1.09 ms
64 bytes from stormnode1 (172.31.22.225): icmp_seq=2 ttl=64 time=1.12 ms
64 bytes from stormnode1 (172.31.22.225): icmp_seq=3 ttl=64 time=1.13 ms
64 bytes from stormnode1 (172.31.22.225): icmp_seq=4 ttl=64 time=1.14 ms
^C
--- stormnode1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3003ms
rtt min/avg/max/mdev = 1.093/1.123/1.140/0.045 ms
ubuntu@ip-172-31-46-249:/opt/kafka/kafka_2.9.1-0.8.2.1/config$ ping stormnode2
PING stormnode2 (172.31.24.92) 56(84) bytes of data.
64 bytes from stormnode2 (172.31.24.92): icmp_seq=1 ttl=64 time=1.08 ms
64 bytes from stormnode2 (172.31.24.92): icmp_seq=2 ttl=64 time=1.27 ms
64 bytes from stormnode2 (172.31.24.92): icmp_seq=3 ttl=64 time=1.30 ms
64 bytes from stormnode2 (172.31.24.92): icmp_seq=4 ttl=64 time=1.26 ms
^C
--- stormnode2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 1.084/1.230/1.300/0.092 ms
ubuntu@ip-172-31-46-249:/opt/kafka/kafka_2.9.1-0.8.2.1/config$ ping stormnode3
PING stormnode3 (172.31.25.230) 56(84) bytes of data.
64 bytes from stormnode3 (172.31.25.230): icmp_seq=1 ttl=64 time=1.15 ms
64 bytes from stormnode3 (172.31.25.230): icmp_seq=2 ttl=64 time=1.26 ms
64 bytes from stormnode3 (172.31.25.230): icmp_seq=3 ttl=64 time=1.26 ms
64 bytes from stormnode3 (172.31.25.230): icmp_seq=4 ttl=64 time=1.33 ms
^C
--- stormnode3 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3003ms
rtt min/avg/max/mdev = 1.153/1.252/1.331/0.063 ms
ubuntu@ip-172-31-46-249:/opt/kafka/kafka_2.9.1-0.8.2.1/config$
```

Figure 5.10: Checking Connection of Three ZooKeeper servers

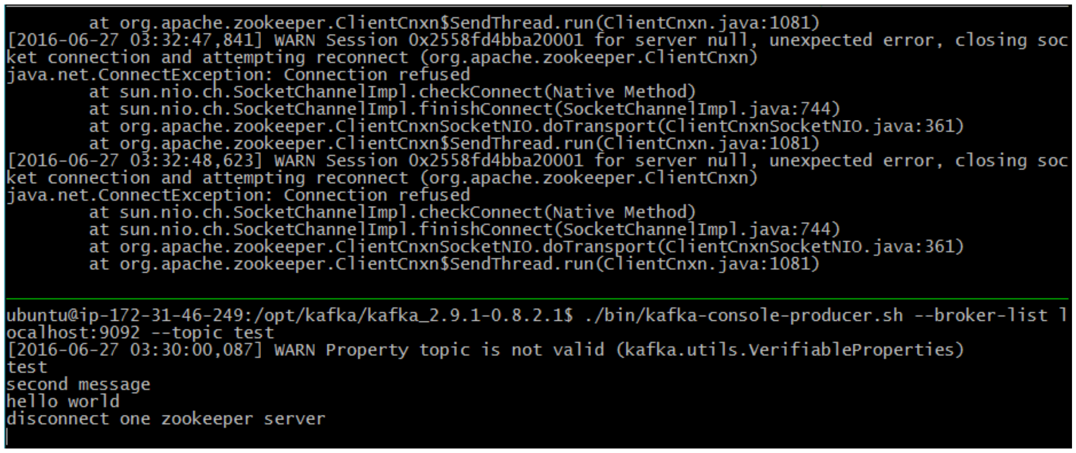


```
[2016-06-27 03:31:33,974] INFO zookeeper state changed (SyncConnected) (org.I0Itec.zkclient.ZkClient)

ubuntu@ip-172-31-46-249:/opt/kafka/kafka_2.9.1-0.8.2.1$ ./bin/kafka-console-consumer.sh --zookeeper sto
rmnode1:2181,stormnode2:2181,stormnode3:2181 --topic test
test
second message
hello world
disconnect one zookeeper server

ubuntu@ip-172-31-46-249:/opt/kafka/kafka_2.9.1-0.8.2.1$ ./bin/kafka-console-producer.sh --broker-list l
ocalhost:9092 --topic test
[2016-06-27 03:30:00,087] WARN Property topic is not valid (kafka.utils.VerifiableProperties)
test
second message
hello world
disconnect one zookeeper server
```

Figure 5.11: Kafka Consumer and Producer, When Closing one ZooKeeper Server



```
        at org.apache.zookeeper.ClientCnxn$SendThread.run(ClientCnxn.java:1081)
[2016-06-27 03:32:47,841] WARN Session 0x2558fd4bba20001 for server null, unexpected error, closing soc
ket connection and attempting reconnect (org.apache.zookeeper.ClientCnxn)
java.net.ConnectException: Connection refused
        at sun.nio.ch.SocketChannelImpl.checkConnect(Native Method)
        at sun.nio.ch.SocketChannelImpl.finishConnect(SocketChannelImpl.java:744)
        at org.apache.zookeeper.ClientCnxnSocketNIO.doTransport(ClientCnxnSocketNIO.java:361)
        at org.apache.zookeeper.ClientCnxn$SendThread.run(ClientCnxn.java:1081)
[2016-06-27 03:32:48,623] WARN Session 0x2558fd4bba20001 for server null, unexpected error, closing soc
ket connection and attempting reconnect (org.apache.zookeeper.ClientCnxn)
java.net.ConnectException: Connection refused
        at sun.nio.ch.SocketChannelImpl.checkConnect(Native Method)
        at sun.nio.ch.SocketChannelImpl.finishConnect(SocketChannelImpl.java:744)
        at org.apache.zookeeper.ClientCnxnSocketNIO.doTransport(ClientCnxnSocketNIO.java:361)
        at org.apache.zookeeper.ClientCnxn$SendThread.run(ClientCnxn.java:1081)

ubuntu@ip-172-31-46-249:/opt/kafka/kafka_2.9.1-0.8.2.1$ ./bin/kafka-console-producer.sh --broker-list l
ocalhost:9092 --topic test
[2016-06-27 03:30:00,087] WARN Property topic is not valid (kafka.utils.VerifiableProperties)
test
second message
hello world
disconnect one zookeeper server
```

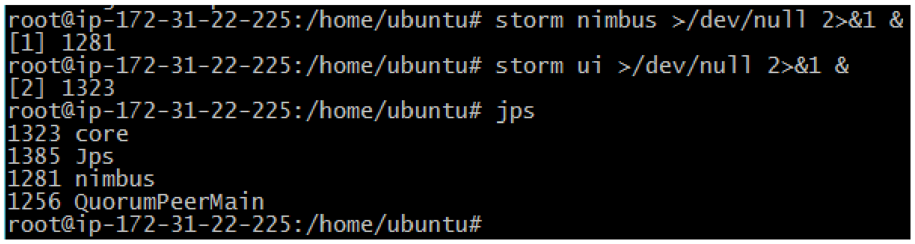
Figure 5.12: Kafka Consumer and Producer, When Closing Two ZooKeeper Server

5.5.2. Availability of Storm

In this experiment, we launch three machines installed with ZooKeeper and Storm. Since the nimbus node does not interact with ZooKeeper while supervisor connects to ZooKeeper cluster. Therefore in this experiment, availability of supervisor of Storm is examined. By analyzing the results of this experiment, we can conclude that the availability of Storm cluster relies on ZooKeeper clusters. All Storm supervisors connect to one ZooKeeper cluster. If more than half of ZooKeeper machines disconnect, all Storm supervisors threads will disconnect at the same time. However, the nimbus node of Storm is not influenced by

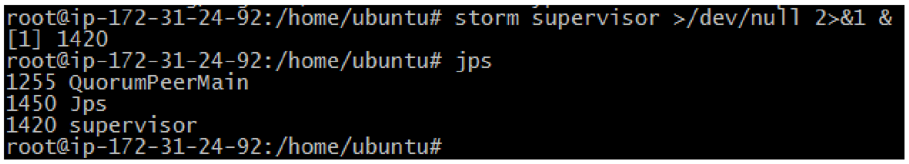
ZooKeeper clusters. The steps of starting ZooKeeper servers are the same as the former experiment:

1. Running nimbus on stormnode1 and supervisor threads on stormnode2 and stormnode3, and then checking background threads with jps command. As Figure 5.13 and 5.14 show, the core thread is the Storm UI, the nimbus thread runs the Storm nimbus, QuorumPeerMain is the thread of ZooKeeper, and supervisor is the thread of Storm supervisor.
2. Stopping one ZooKeeper thread and checking the Storm UI on the port 8080. Finding that there are still two supervisors, as Figure 5.15 shows.
3. Stopping one more ZooKeeper thread and checking Storm UI again. The result is shown as Figure 5.16.
4. Restarting ZooKeeper threads. The supervisor will become available again.



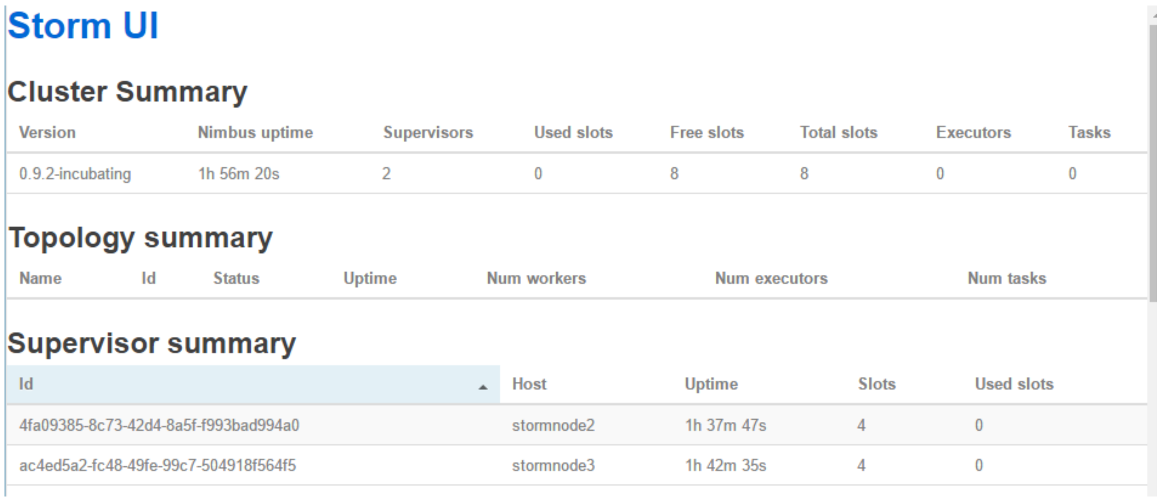
```
root@ip-172-31-22-225:/home/ubuntu# storm nimbus >/dev/null 2>&1 &
[1] 1281
root@ip-172-31-22-225:/home/ubuntu# storm ui >/dev/null 2>&1 &
[2] 1323
root@ip-172-31-22-225:/home/ubuntu# jps
1323 core
1385 Jps
1281 nimbus
1256 QuorumPeerMain
root@ip-172-31-22-225:/home/ubuntu#
```

Figure 5.13: The Screen Shot of Storm Nimbus Machine



```
root@ip-172-31-24-92:/home/ubuntu# storm supervisor >/dev/null 2>&1 &
[1] 1420
root@ip-172-31-24-92:/home/ubuntu# jps
1255 QuorumPeerMain
1450 Jps
1420 supervisor
root@ip-172-31-24-92:/home/ubuntu#
```

Figure 5.14: The Screen Shot of Storm supervisor Machine



Storm UI

Cluster Summary

Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
0.9.2-incubating	1h 56m 20s	2	0	8	8	0	0

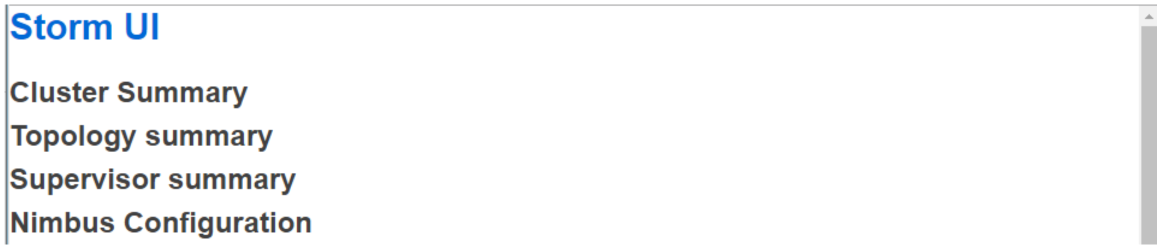
Topology summary

Name	Id	Status	Uptime	Num workers	Num executors	Num tasks

Supervisor summary

Id	Host	Uptime	Slots	Used slots
4fa09385-8c73-42d4-8a5f-f993bad994a0	stormnode2	1h 37m 47s	4	0
ac4ed5a2-fc48-49fe-99c7-504918f564f5	stormnode3	1h 42m 35s	4	0

Figure 5.15: Storm UI with Two ZooKeeper Servers



Storm UI

Cluster Summary

Topology summary

Supervisor summary

Nimbus Configuration

Figure 5.16: Storm UI with One ZooKeeper Server

5.6. Use Cases

We present the use cases of sentiment analysis and word cloud. The index page of website is shown in Figure 5.17, the options of each selection column corresponds to the flow chart of web page.

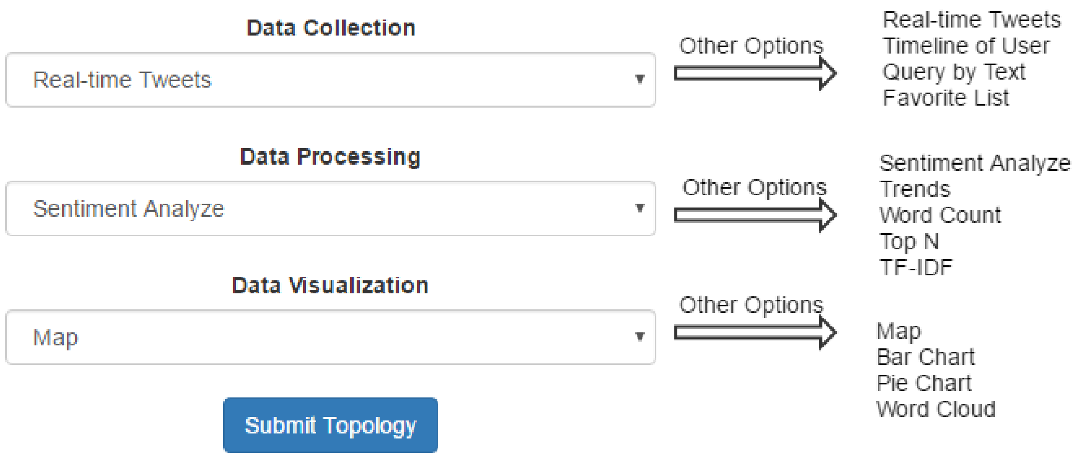


Figure 5.17: Web Index Page

5.6.1. Sentiment Analysis of Real-time Twitter Messages

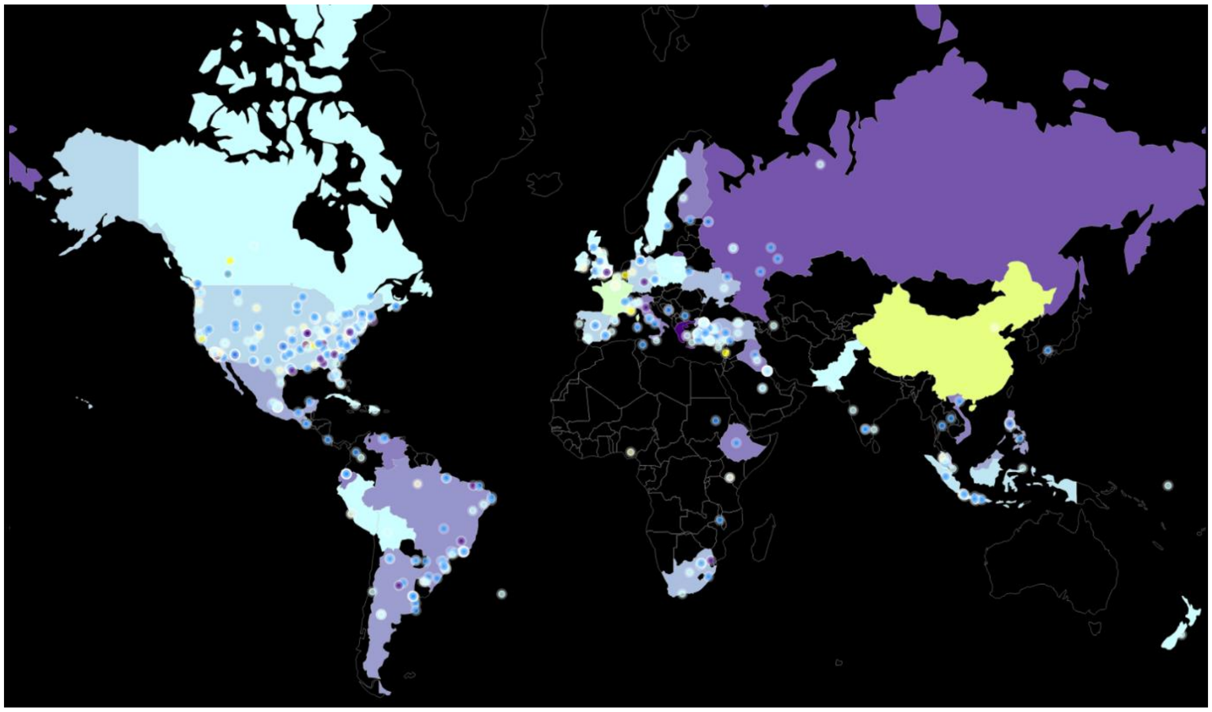


Figure 5.18: Sentiment Map Visualization

After launching the application, users can visit the website by entering the public IP of the machine. By choosing real-time tweets, sentiment analysis, and map options, the web page will be redirected to the map visualization page. At the same time, the real-time data collection component and sentiment analysis component are running in the background of the server side. The processed data are transmitted to the front end with Server-sent event. The launch of Storm topology requires around half a minute and then the webpage shows the processed results. After 5 minutes, the results are visualized on the map, as Figure 5.18 shows.

In this case, the system collects real-time data from Twitter with Twitter API. Since the user chooses map as the visualization method, tweets without geographical information are filtered out. The messages remaining are processed by sentiment analysis topology, and the system generates the sentiment of each message and the average sentiment of each country. On the visualization page, the personal sentiment is represented with coloured points, while the average sentiment of the country is represented with the colors of block. As Figure 5.18 shows, most tweets are sent in America, and there are fewer tweets in Africa and Asia.

5.6.2. Word Cloud of Real-time Twitter Messages

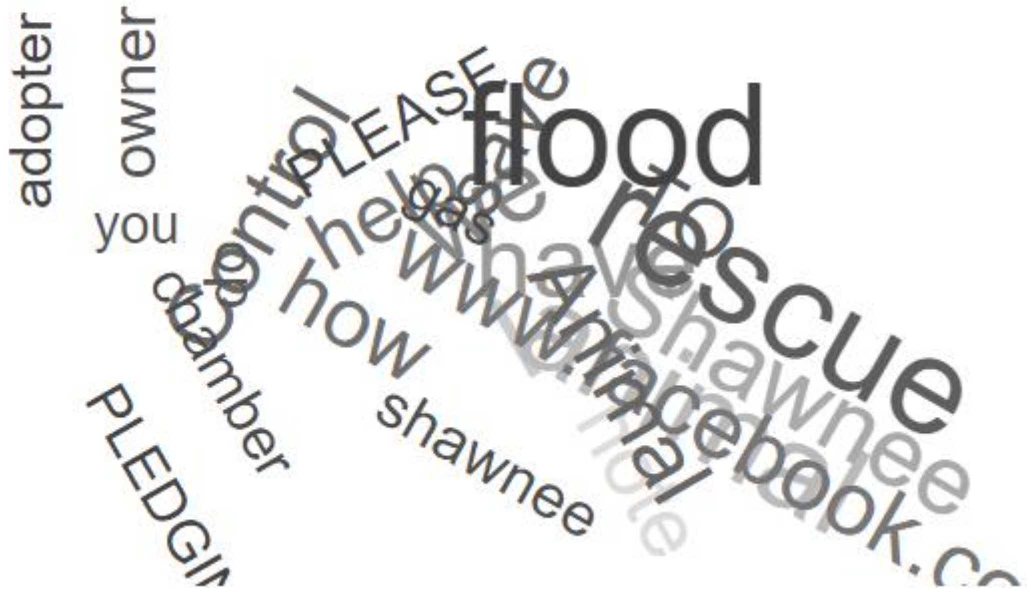


Figure 5.19: Word Cloud

Similar to the sentiment visualization case, users select real-time data collection, trend analysis, and word cloud components to build their social mining tools. Aside from the word cloud component, other components run in the background. Once the web page is redirected to the word cloud page, the data is visualized as word cloud, as Figure 5.19 shows.

In this case, the system collects Twitter real-time data as well. The collected data are processed with trend analysis topology which fetches data from the URLs of tweets and ranks words by its TF-IDF score. The system only transmits the top 20 words to the web page. As the system analyzes the trends of real-time data, the scope of data is really broad. Thus the results of visualization present general words.

5.7. Summary

In this chapter, we presented the methods of deploying system and evaluated the performance of data collection components and data processing components. The scalability, availability, and expandability of the systems are demonstrated in this chapter as well. In the final portion, there are several case studies of this project.

Chapter 6

Conclusion and Future Work

6.1. Conclusion

We implement a component-based Twitter text mining system, which can collect data, process data with Apache Storm, and visualize data on the website. The whole system can be run locally with Vagrant and VirtualBox or in the cloud with AWS EC2.

The main contributions of this project are:

- The project implements a component-based distributed text mining system. The system involves multiple data collection, data processing and data visualization components. The data format within this system is defined as two types of message strings.
- The system equips scalability, availability, and expandability. The scalability is guaranteed by the clusters of Kafka and Apache Storm. By increasing the number of machines in the Kafka cluster, the scalability of web access as well as data transmission within the system will increase. To scale the processing speed and the number of processing units, both parallelism and machine numbers can be adjusted. The availability of this system is guaranteed by ZooKeeper clusters which are the basis of Kafka and Storm clusters. The expandability is guaranteed by the unified data format and the mechanism of Kafka. Every component implementing producers or consumers of Kafka and

following the restriction of data format is available to integrate with other components.

- The project involves a novel Twitter trend analysis topology. Compared to common trend analysis method, our trend analysis calculates TF-IDF score of words in documents which are fetched on the basis of URLs in Twitter messages. The most common method is counting the number of words appearing in tweets, however most words in tweets are meaningless and the ideas of users are contained in its URLs.
- The project formulates the Kafka centric architecture, as a result of the evolution of simple linear architecture and message feedback architecture and compares them with the Lambda architecture.
- Instead of using bolts as basic data processing components, the framework treats Storm topologies as the basic computing units, and every topology implements both Kafka producers and consumers. Thus, every topology can be integrated with other topologies freely.

6.2. Future Work

This section describes the future work of this project, in three functionalities, data collection, data processing, and data visualization, respectively.

In terms of data collection, the project only involves data from Twitter, yet there are still a great number of social network websites or applications. In the future, data collection will involve more data sources, such as Facebook, Instagram, and LinkedIn.

From data processing perspectives, the project focuses on implementing a part of natural language processing algorithms, which limits the range of this project. Besides text files, images and videos contain beneficial information in the social networks as well. For example, Instagram is a kind of social network focused on presenting images and videos. Some social applications even contain a large amount of audio.

To implement the data visualization component, the project involves Google Charts and D3.js to present data. Data visualization can be improved by combining other libraries, such as Leaflet.js and Mapbox. Furthermore, the appearance of the website is another potential improvement of the system.

Once data flows through Apache Kafka, the data will be stored as logs, for this reason, the system does not provide additional data persistence function, the drawback of log storage is that it is hard to get specific information. On the one hand, some databases like Apache Cassandra, Neo4j, provide high speed queries. On the other hand, some search engines, like Apache Solr, and ElasticSearch, build index for information and enable users to search related information.

Finally, in order to implement dynamic topologies, Storm topologies are used as basic processing units which increases the flexibility of the system and decreases the time of developing new processing functions, however, the size of every topology becomes much bigger. In the future, the project intends to change the source code of Storm to allow topology to be changed dynamically.

Appendix A : Dependencies of Project

The following are development kits and software packages used in the system.

Name	**Version**	**Description**
JDK	1.7.0_95	Java SE Development Kit
Apache Maven	3.0.5	Apache Maven is used to pack Java file, in the project.
Apache ZooKeeper	3.4.6	The basis of several distributed system, such as Apache Storm and Apache Kafka
Apache Storm	0.9.2-incubating	Apache Storm is the basis of data processing components of the project.
Apache Kafka	kafka_2.9.1 version 0.8.2.1	Apache Kafka is used to connect different components, in the project.
Stanford NLP	3.4.1	Stanford NLP is applied to calculate the sentiment of Twitter message.
Apache Tika	1.12	Apache Tika is used to pre-process Twitter messages.
Apache Lucene	3.6.2	Apache Lucene is applied to fetch the contents of webpages.
Twitter4J	3.0	Twitter4J is used to collect stream data from Twitter.
Flask	0.10.1	A lightweight Python web framework
Python-pip	1.5.4	A Python package management tool, similar to apt-get as Ubuntu.
Kafka-python	supports Kafka version 0.10, 0.9, 0.8.2, 0.8.1, 0.8 Python version: 2.6, 2.7, 3.3, 3.4, 3.5	It is a Kafka client which is implemented by Python language.
Tweepy	3.5.0	Tweepy is used to collect data from Twitter.
Pycountry	1.20	Pycountry is used to transmit country code from ISO alpha-2 to alpha-3.

Appendix B : Configuration Files of ZooKeeper

There are configuration files of ZooKeeper server. ZooKeeper has cluster mode and standalone mode. In the cluster mode, every machine should be configured by modifying its zoo.cfg file to the first table. The standalone mode runs multiple threads on different ports of one machine, which requires developers to create three configuration files, as zoo1.cfg, zoo2.cfg, and zoo3.cfg.

zoo.cfg:

```
# the number of milliseconds of each tick
# here means zookeeper cluster will send hello message every 2 seconds
tickTime=2000
initLimit=10
syncLimit=5
dataDir=/opt/zookeeper/zkdata
dataLogDir=/opt/zookeeper/logs
clientPort=2181
# server list
# format server.<server number>=<ip address/hostname>:<connect port>:<election
port>
server.1=stormnode1:2888:3888
server.2=stormnode2:2888:3888
server.3=stormnode3:2888:3888
```

zoo1.cfg

```
tickTime=2000
initLimit=10
syncLimit=5
dataDir=/opt/zookeeper/zkdata1
dataLogDir=/opt/zookeeper/logs1
# the port at which the clients will connect
clientPort=2181

server.1=localhost:2888:3888
server.2=localhost:2889:3889
server.3=localhost:2890:3890
```

zoo2.cfg

```
tickTime=2000
initLimit=10
syncLimit=5
dataDir=/opt/zookeeper/zkdata2
dataLogDir=/opt/zookeeper/logs2
clientPort=2182

server.1=localhost:2888:3888
server.2=localhost:2889:3889
server.3=localhost:2890:3890
```

zoo3.cfg

```
tickTime=2000
initLimit=10
syncLimit=5
dataDir=/opt/zookeeper/zkdata2
dataLogDir=/opt/zookeeper/logs2
clientPort=2183

server.1=localhost:2888:3888
server.2=localhost:2889:3889
server.3=localhost:2890:3890
```

Appendix C : Configuration Files of Apache Storm Cluster

To deploy Storm clusters, only storm.yaml file needs to be changed. The meaning of each configuration item is written with comments.

Storm.yaml

```
# zookeeper server list
# If ZooKeeper use default port, there is no need to indicate port. Otherwise, Storm need
to indicate port number here.
storm.zookeeper.servers:
    - "stormnode1"
    - "stormnode2"
    - "stormnode3"

# nimbus ip or hostname
nimbus.host: "stormnode1"
storm.local.dir: "/opt/storm/status"
# one port can run one worker
supervisor.slots.ports:
       - 6700
       - 6701
       - 6702
       - 6703
```

Appendix D : Configuration Files of Apache Kafka

The following table only lists important configuration items and adds comments for them. In theory, the configuration file of Kafka server can be placed anywhere, nonetheless, it is recommended to store them in the config folder with suffix of properties.

Server.properties

```
# The broker id
broker.id=0

# The port the socket server listens on
port=9092

# The receive buffer (SO_RCVBUF) used by the socket server
socket.receive.buffer.bytes=102400
socket.request.max.bytes=104857600
log.dirs=/tmp/kafka-logs
num.partitions=1

# The maximum time where log can be kept
log.retention.hours=168

# The maximum size of log which can be kept in the system
log.retention.bytes=1073741824

#indicates the zookeeper server of Kafka broker need to connect. The Kafka broker is
available when more than half zookeeper server is alive
zookeeper.connect=localhost:2181

# enable deletion of topics
delete.topic.enable=true
```

Bibliography

[1] Boyd, D., & Ellison, N. (2007). Social network sites: Definition, history, and scholarship. *Journal of Computer-Mediated Communication, 13(*1), 210-230. doi:10.1111/j.1083-6101.2007.00393.x

[2] Apache Storm Use Cases - Edureka Blog. (2014). Retrieved June 29, 2016, from http://www.edureka.co/blog/apache-storm-use-cases.

[3] Tan, A. H. (1999, April). Text mining: The state of the art and the challenges. *Proceedings of the PAKDD 1999 Workshop on Knowledge Disocovery*, 65-70.

[4] Zeng, L., Li, L., Duan, L., Lu, K., Shi, Z., Wang, M. & Luo, P. (2012). Distributed data mining: A survey. *Information Technology and Management, 13*(4), 403-409. doi:10.1007/s10799-012-0124-y

[5] Yu, L., Zheng, J., Shen, W. C., Wu, B., Wang, B., Qian, L., & Zhang, B. R. (2012). BC-PDM: Data mining, social network analysis and text mining system based on cloud computing. *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '12,* 1496-1499. doi:10.1145/2339530.2339764

[6] Artola, X., Beloki, Z., & Soroa, A. (2014). A stream computing approach towards scalable NLP. *LREC*, 8-13.

[7] Aggarwal, C. C. (2011). An Introduction to Social Network Data Analytics. *Social Network Data Analytics,* 1-15. doi:10.1007/978-1-4419-8462-3_1

[8] Haythornthwaite, C. (1996). Social network analysis: An approach and technique for the study of information exchange. *Library and Information Science Research, 18*(4), 323-342. doi:10.1016/S0740-8188(96)90003-1

[9] Go, A., Bhayani, R., & Huang, L. (2009). Twitter sentiment classification using distant supervision. *CS224N Project Report, Stanford*, 1, 12.

[10] Papacharissi, Z. (2009). The virtual geographies of social networks: a comparative analysis of Facebook, LinkedIn and ASmallWorld. *New media & society*, *11*(1-2), 199-220.

[11] Wang, Y., Callan, J., & Zheng, B. (2015). Should we use the sample? Analyzing datasets sampled from twitter's stream API. *ACM Transactions on the Web (TWEB), 9*(3), 1-23. doi:10.1145/2746366

[12] Vural, A. G., Cambazoglu, B. B., & Karagoz, P. (2014). Sentiment-focused web crawling. *ACM Transactions on the Web (TWEB), 8*(4), 1-21. doi:10.1145/2644821

[13] Achrekar, H., Gandhe, A., Lazarus, R., Yu, S., & Liu, B. (2011). Predicting Flu Trends using Twitter data. *2011 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 702-707. doi:10.1109/infcomw.2011.5928903

[14] Driscoll, K., & Walker, S. (2014). Big data, big questions working within a black box: Transparency in the collection and production of big twitter data. *International Journal of Communication, 8*(2014), 1745-1764.

[15] Kaggle home page. (2016). Retrieved June 29, 2016, from https://www.kaggle.com.

[16] Ontario open data. (2015, October 2). Retrieved June 29, 2016, from https://www.ontario.ca/page/sharing-government-data.

[17] Extracting Information from Text. (2015, July 1). Retrieved June 29, 2016, from http://www.nltk.org/book/ch07.html.

[18] N-gram. (2016, May 9). Retrieved June 29, 2016, from https://en.wikipedia.org/wiki/N-gram.

[19] Twitter Wordcloud Bot. (2014, December 27). Retrieved June 29, 2016, from https://github.com/defacto133/twitter-wordcloud-bot/tree/master/assets.

[20] Tf–idf. (2016, May 5). Retrieved June 29, 2016, from https://en.wikipedia.org/wiki/Tf–idf.

[21] TF-IDF Explantion (2012, October 24). Retrieved June 29, 2016, from http://coolshell.cn/articles/8422.html.

[22] Zhang, W., Yoshida, T., & Tang, X. (2011). A comparative study of TFIDF, LSI and multi-words for text classification. *Expert Systems with Applications, 38*(3), 2758-2765. doi:10.1016/j.eswa.2010.08.066

[23] Apache™ Hadoop®! (2014). Retrieved July 05, 2016, from http://hadoop.apache.org.

[24] Ranjan, R. (2014). Streaming big data processing in datacenter clouds. *IEEE Cloud Computing, 1*(1), 78-83.

[25] Apache Storm. (2015). Retrieved June 29, 2016, from http://storm.apache.org.

[26] Karunaratne, P., Karunasekera, S., & Harwood, A. (2016, June 18). Distributed stream clustering using micro-clusters on apache storm. *Journal of Parallel and Distributed Computing*, doi:10.1016/j.jpdc.2016.06.004

[27] Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., & Bhagat, N. (2014, June). Storm@ twitter. *Proceedings of the 2014 ACM SIGMOD*

international conference on Management of data, 147-156. doi:10.1145/2588555.2595641

[28] Storm concepts. (2015). Retrieved June 29, 2016, from http://storm.apache.org/releases/0.9.6/Concepts.html.

[29] Trident tutorial. (2015). Retrieved June 29, 2016, from http://storm.apache.org/releases/0.9.6/Trident-tutorial.html.

[30] Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., &Taneja, S. (2015). Twitter Heron. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15*, 239-250. doi:10.1145/2723372.2742788

[31] Spark streaming programming guide. (2015). Retrieved June 29, 2016, from http://spark.apache.org/docs/latest/streaming-programming-guide.html.

[32] Nair, L. R., & Shetty, S. D. (2015). Streaming twitter data analysis using spark for effective job search. *Journal of Theoretical and Applied Information Technology*, *80*(2), 349.

[33] Apache ZooKeeper™. (2016). Retrieved June 29, 2016, from https://zookeeper.apache.org.

[34] Visual Variables. (2012, September 5). Retrieved June 29, 2016, from http://www.infovis-wiki.net/index.php?title=Visual_Variables.

[35] Bostock, M. (2015). D3.js - Data-Driven Documents. Retrieved July 05, 2016, from https://d3js.org.

[36] ECharts. (2015). Retrieved July 05, 2016, from http://echarts.baidu.com.

[37] Google Charts. (2016, June 13). Retrieved July 05, 2016, from https://developers.google.com/chart.

[38] Hunter, J., Dale, D., Firing, E., & Driettboom, M. (2016, June 18). Matplotlib: Python plotting — Matplotlib 1.5.1 documentation. Retrieved July 05, 2016, from http://matplotlib.org.

[39] Learning Path: Enterprise Messaging Techniques [Video]. (2016, June). Retrieved June 29, 2016, from https://www.safaribooksonline.com/library/view/learning-path-enterprise/9781491964965.

[40] Baron, C. A. (2016). NoSQL key-value DBs riak and redis. *Database Systems Journal*, (4), 3-10.

[41] Apache Kafka. (2016). Retrieved June 29, 2016, from http://kafka.apache.org.

[42] Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines). (2014, April 27). Retrieved June 29, 2016, from https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines.

[43] Namiot, D. (2015). On Big Data Stream Processing. *International Journal of Open Information Technologies*, *3*(8), 48-51.

[44] Ritter, A., Clark, S., & Etzioni, O. (2011, July). Named entity recognition in tweets: an experimental study. *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 1524-1534.

[45] Ritter, A., Etzioni, O., & Clark, S. (2012, August). Open domain event extraction from twitter. *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, 1104-1112. doi:10.1145/2339530.2339704

[46] Socialsensor: multimedia-geotagging. (2015, July 29). Retrieved June 29, 2016, from https://github.com/socialsensor/multimedia-geotagging/tree/demo.

[47] Prediction API - Pattern Matching in the Cloud. (2016). Retrieved June 29, 2016, from https://cloud.google.com/prediction.

[48] Monkey Learning Documentation. (2016). Retrieved June 29, 2016, from http://docs.monkeylearn.com.

[49] SAMOA. (2016, March 14). Retrieved June 29, 2016, from https://github.com/apache/incubator-samoa.

[50] De Francisci Morales, G. (2013, May). SAMOA: A platform for mining big data streams. *Proceedings of the 22nd International Conference on World Wide Web,* 777-778.

[51] Neumeyer, L., Robbins, B., Nair, A., & Kesari, A. (2010, December). S4: Distributed Stream Computing Platform. *2010 IEEE International Conference on Data Mining Workshops*, 170-177. doi:10.1109/icdmw.2010.172

[52] Bifet, A., & Morales, G. D. (2014, December). Big Data Stream Learning with SAMOA. *2014 IEEE International Conference on Data Mining Workshop*, 1199-1202. doi:10.1109/icdmw.2014.24

[53] Trident-ml. (2015, October 2). Retrieved June 29, 2016, from https://github.com/pmerienne/trident-ml.

[54] Han, Z., & Xu, M. (2015, December). Machine Learning Techniques in Storm. *2015 Seventh International Symposium on Parallel Architectures, Algorithms and Programming (PAAP),* 139-142. doi:10.1109/paap.2015.35

[55] StormCV. (2015, Sepetember 26). Retrieved June 29, 2016, from https://github.com/sensorstorm/StormCV.

[56] Generating a network graph of Twitter followers using Python and NetworkX. (2014). Retrieved June 29, 2016, from http://mark-kay.net/2014/08/15/network-graph-of-twitter-followers.

[57] Calculating a Network Map by analyzing Tweets. (2014). Retrieved June 29, 2016, from http://mark-kay.net/2014/01/15/calculating-a-network-map-by-analyzing-tweets.

[58] Liu, S., Wang, X., Chen, J., Zhu, J., & Guo, B. (2014). TopicPanorama: A full picture of relevant topics. *2014 IEEE Conference on Visual Analytics Science and Technology (VAST)*, 183-192. doi:10.1109/vast.2014.7042494

[59] Byron, L., & Wattenberg, M. (2008). Stacked Graphs – Geometry & Aesthetics. IEEE Trans. Visual. Comput. *Graphics IEEE Transactions on Visualization and Computer Graphics, 14*(6), 1245-1252. doi:10.1109/tvcg.2008.166

[60] D3 Interactive Streamgraph. (2016, June 14). Retrieved June 29, 2016, from http://bl.ocks.org/WillTurman/4631136.

[61] Streamgraph. (2016, March 2). Retrieved June 29, 2016, from http://bl.ocks.org/mbostock/4060954.

[62] Cui, W., Liu, S., Wu, Z., & Wei, H. (2014). How hierarchical topics evolve in large text corpora. *IEEE transactions on visualization and computer graphics*, *20*(12), 2281-2290.

[63] Heer, J., & Boyd, D. (2005, October). Vizster: Visualizing online social networks. *IEEE Symposium on Information Visualization,* 1277-1284. doi:10.1109/infvis.2005.1532126

[64] Cui, W., Zhou, H., Qu, H., Wong, P. C., & Li, X. (2008). Geometry-based edge clustering for graph visualization. *IEEE Transactions on Visualization and Computer Graphics, 14*(6), 1277-1284.

[65] Force Directed Edge Bundling (FDEB) in Javascript. (2016, March 11). Retrieved June 29, 2016, from https://github.com/upphiminn/d3.ForceBundle.

[66] Kafka Connect. (2015). Retrieved June 29, 2016, from http://docs.confluent.io/2.0.0/connect.

[67] How to beat the CAP theorem - thoughts from the red planet - thoughts from the red planet. (2011, October 13). Retrieved June 29, 2016, from http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html.

[68] Gilbert, S., & Lynch, N. (2012). Perspectives on the CAP theorem. *Computer, 45*(2), 30-36. doi:10.1109/MC.2011.389

[69] Liu, X., Iftikhar, N., & Xie, X. (2014). Survey of real-time processing systems for big data. *Proceedings of the 18th International Database Engineering & Applications Symposium on - IDEAS '14*, 356-361. doi:10.1145/2628194.2628251

[70] TweetStats. (2008). Retrieved July 05, 2016, from http://www.tweetstats.com.

[71] Twitter Charts. (2008). Retrieved July 05, 2016, from http://xefer.com/twitter.

[72] Twitter Counter. (2016). Retrieved July 05, 2016, from http://twittercounter.com.

[73] Word Cloud Bot. (2014, November). Retrieved July 5, 2016, from https://twitter.com/wordnuvola?lang=en.

[74] Automatic terms extraction for Domain-specific corpora. (2014). Retrieved June 29, 2016, from https://www.datacrucis.com/research/automatic-terms-extraction-for-domain-specific-corpora.html.

[75] Preprocessing — Text Analysis with Topic Models for the Humanities and Social Sciences. (2016). Retrieved June 29, 2016, from https://de.dariah.eu/tatom/preprocessing.html.

[76] Iqbal, M. H., & Soomro, T. R. (2015). Big Data Analysis: Apache Storm Perspective. *International Journal of Computer Trends and Technology IJCTT, 19*(1), 9-14. doi:10.14445/22312803/ijctt-v19p103.

[77] Twitter Rate Limits. (2016). Retrieved June 29, 2016, from https://dev.twitter.com/rest/public/rate-limits.

[78] Udacity and Twitter bring you Real-Time Analytics with Apache Storm. (2014, November 20). Retrieved June 29, 2016, from https://github.com/udacity/ud381.

[79] Word Cloud Layout. (2015, October 15). Retrieved June 29, 2016, from https://github.com/jasondavies/d3-cloud.

[80] Storm Command Line Client. (2015). Retrieved June 29, 2016, from http://storm.apache.org/releases/1.0.1/Command-line-client.html.

[81] Stopping Storm: The right way. (2013, November 12). Retrieved June 29, 2016, from http://stackoverflow.com/questions/19926548/stopping-storm-the-right-way.

[82] Vagrant. (2016). Retrieved July 05, 2016, from https://www.vagrantup.com.

[83] VirtualBox (2016, June 28). Retrieved July 05, 2016, from https://www.virtualbox.org.

[84] Apache Ambari. (2016). Retrieved July 05, 2016, from https://ambari.apache.org.

[85] Apache Maven. (2016). Retrieved June 29, 2016, from https://maven.apache.org.

[86] Apache ZooKeeper™ Releases. (2016). Retrieved June 29, 2016, from http://zookeeper.apache.org/releases.html.

[87] Storm downloads. (2015). Retrieved June 29, 2016, from http://storm.apache.org/downloads.html.

[88] Kafka Releases. (2016). Retrieved June 29, 2016, from http://kafka.apache.org/downloads.html.