A Framework for Hadoop Based Digital Libraries of Tweets

Matthew Bock

Thesis submitted to the Faculty of
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Masters of Science
in
Computer Science and Applications

Edward A. Fox, Chair
Andrea Kavanaugh
Chandan Reddy

May 3, 2017
Blacksburg, Virginia

Keywords: Big Data, Digital Libraries, Data Structures

Virginia Tech

Abstract

Masters of Science

A Framework for Hadoop Based Digital Libraries of Tweets

by Matthew BOCK

The Digital Library Research Laboratory (DLRL) has collected over 1.5 billion tweets for the Integrated Digital Event Archiving and Library (IDEAL) and Global Event Trend Archive Research (GETAR) projects. Researchers across varying disciplines have an interest in leveraging DLRL's collections of tweets for their own analyses. However, due to the steep learning curve involved with the required tools (Spark, Scala, HBase, etc.), simply converting the Twitter data into a workable format can be a cumbersome task in itself. This prompted the effort to build a framework that will help in developing code to analyze the Twitter data, run on arbitrary tweet collections, and enable developers to leverage projects designed with this general use in mind. The intent of this thesis work is to create an extensible framework of tools and data structures to represent Twitter data at a higher level and eliminate the need to work with raw text, so as to make the development of new analytics tools faster, easier, and more efficient.

To represent this data, several data structures were designed to operate on top of the Hadoop and Spark libraries of tools. The first set of data structures is an abstract representation of a tweet at a basic level, as well as several concrete implementations which represent varying levels of detail to correspond with common sources of tweet data. The second major data structure is a collection structure designed to represent collections of tweet data structures and provide ways to filter, clean, and process the collections. All of these data structures went through an iterative design process based on the needs of the developers.

The effectiveness of this effort was demonstrated in four distinct case studies. In the first case study, the framework was used to build a new tool that selects Twitter data from DLRL's archive of tweets, cleans those tweets, and performs sentiment analysis within the topics of a collection's topic model. The second case study applies the provided tools for the purpose of sociolinguistic studies. The third case study explores large datasets to accumulate all possible analyses on the datasets. The fourth case study builds metadata by expanding the shortened URLs contained in the tweets and storing them as metadata about the collections. The framework proved to be useful and cut development time for all four of the case studies.

Virginia Tech

General Audience Abstract

Masters of Science

A Framework for Hadoop Based Digital Libraries of Tweets

by Matthew BOCK

The Digital Library Research Laboratory (DLRL) has collected over 1.5 billion tweets for the Integrated Digital Event Archiving and Library (IDEAL) and Global Event Trend Archive Research (GETAR) projects. Researchers across varying disciplines have an interest in leveraging DLRL's collections of tweets for their own analyses. However, due to the steep learning curve involved with the required tools, simply converting the Twitter data into a workable format can be a cumbersome task in itself. This prompted the effort to build a programming framework that will help in developing code to analyze the Twitter data, run on arbitrary tweet collections, and enable developers to leverage projects designed with this general use in mind. The intent of this thesis work is to create an extensible framework of tools and data structures to represent Twitter data at a higher level and eliminate the need to work with raw text, so as to make the development of new analytics tools faster, easier, and more efficient.

The effectiveness of this effort was demonstrated in four distinct case studies. In the first case study, the framework was used to build a new tool that selects Twitter data from DLRL's archive of tweets, cleans those tweets, and performs sentiment analysis within the topics of a collection's topic model. The second case study applies the provided tools for the purpose of sociolinguistic studies. The third case study explores large datasets to accumulate all possible analyses on the datasets. The fourth case study builds metadata by expanding the shortened URLs contained in the tweets and storing them as metadata about the collections. The framework proved to be useful and cut development time for all four of the case studies.

For my parents Jean and Tom, without whose constant love and support I would not be who I am today. I owe them the world, and dedicate this thesis to them.

Acknowledgements

This project would not have been possible without the assistance and support of many people. I would especially like to acknowledge and thank the following:

- Project and graduate advisor Dr. Edward Fox
- Committee members Dr. Andrea Kavanaugh and Dr. Chandan Reddy
- Integrated Digital Event Archiving and Library (IDEAL) Grant: IIS-1319578
- Global Event and Trend Archive Research (GETAR) Grant: IIS-1619028 and 1619371
- Other members of the Digital Library Research Laboratory, especially:
  - Abigail Bartolome, for case study participation and proofreading assistance
  - Prashant Chandrasekar, for helpful feedback and guidance
  - Islam Harb, for case study participation
  - Liuqing Li, for case study participation and assistance with the cluster
  - Dr. Sunshin Lee, for assistance with Spark and the cluster
- Other students in CS 5604 Information Retrieval and CS 6604 Digital Libraries, especially:
  - Radha Krishnan Vinayagam, for case study participation
  - Rahul Krishnamurthy, for case study participation
- All of my friends and family who helped me along the way:
  - Lauren, Allison, Nanny, and Poppy for constantly supporting me in all that I do
  - Cate, for proofreading help and for providing endless encouragement and stress relief
  - All of my Blacksburg friends, for making my college career the best years of my life

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

This thesis describes simplifying some of the research connected with the Digital Library Research Laboratory (DLRL). It targets a wide variety of researchers: data science researchers doing unique and complex low-level analysis using programs that they write; non-technical researchers who want to apply those techniques to learn more about some data set they are interested in; and even those in between who simply want easy access to and preprocessing of the collections stored in our digital library so that they can apply some other tools or techniques. Providing endpoints and supplying core functionalities at many levels like this will hopefully go a long way towards making digital library research faster, easier, and more accessible.

The initial goal of this study was to build a tool that would allow researchers to analyze sentiment and topic analysis over time. "Over time" in this case refers to comparing results across a set of collections within DLRL's library of tweet collections. Since many collections represent specific events, running an analysis on a set of events that are all within the same topic area (for example: school shooting incidents, hurricanes, related political events) would expose trends in discussion over time. The work has since broadened in scope since the sentiment/topic analysis goal has been turned into a class project for CS 6604: Digital Libraries. Consequently, time and resources are available to make code much more modular and extensible and turn it into a more generalized framework for people to expand upon. The sentiment/topic analysis research is now a case study of how this tool can: help acceletate and simplify the development process, and help make new data analytics tools which can process arbitrary tweet collections (as opposed to within a specific domain).

1.2 Motivation

Many experts with wildly varying requirements are interested in the kind of research that happens in the Digital Library Research Laboratory [10], especially when it comes to leveraging our digital library of over one thousand collections containing over 1.5 billion tweets [11]. However, leveraging this data is a complex process, and the learning curve involved with getting to know the tools can make getting started very difficult. With this in mind, I have designed this framework to make it much faster and easier to work with the Twitter data that the lab uses for a key portion of its research.

In addition, I would like this framework to be extensible so that other researchers in the lab can eventually add what they create for other projects, to its suite of tools. Given enough time and participation, the framework will become a robust toolkit

that those involved in digital library research can use to do research involving the tweet collections maintained by the lab.

1.3 Problem

The main obstacle that this system is built to solve is the difficulty of undertaking digital library research with large tweet collections. "Big Data" research is notoriously difficult, but also very powerful in multiple domain areas. From sociology to psychology to engineering and so on, the ability to extract useful information from large collections of data is incredibly useful for those interested in improving or innovating within those domains. As of today, the Digital Library Research Laboratory at Virginia Tech maintains, among other things, a library of over 1000 collections of over 1.5 billion tweets. We aim to provide an easier way for researchers and data scientists to access and analyze this wealth of knowledge and make use of it for their own efforts.

1.4 Research Questions

Does representing tweets as well-defined data structures rather than arbitrary text data help with the development of tools to analyze that data? What kinds of tools would benefit researchers trying to use our digital library to learn things about their areas of interest? Can the resulting framework, with its set of tools, be shown to be of help through multiple case studies?

1.5 Hypothesis

With the aforementioned research questions in mind, we present the following hypothesis: The creation of an extensible framework of tools and data structures to represent Twitter data at a higher level, and to eliminate the need to work with raw text, would make the development of new analytics tools faster, easier, and more efficient – as will be shown in a realistic set of case studies.

1.6 Organization of Thesis

The rest of this document is organized as follows: Chapter 2 presents a literature review which discusses research into the tools available, and into the needs of projects which could leverage a framework like this, to help motivate the design of the framework. Chapter 3 gives an outline of the design and architecture of our system, including a brief discussion of the functionality of each of the provided classes. Chapter 4 discusses the implementation of the system and important decisions made along the way. Chapter 5 details the case studies, which serve to evaluate the success of the system by having developers with varying levels of knowledge of the preexisting software develop projects using the framework to suit their needs. Chapter 6 summarizes the previous sections. Chapter 7 presents final conclusions and proposals for future work.

Chapter 2

Literature Review

2.1 Motivating Works

When approaching background research for this project, it was important to get a good grasp on the types of projects this framework would be supporting. To do this, two main areas were researched. First, a survey of the past two years of CS5604 (Information Retrieval) classes was conducted. The goal was to get a sense of the kinds of work teams are doing with the tweet collections. Second, a review of the IDEAL and GETAR projects was conducted. These two NSF-funded efforts have motivated much of the research work that has gone into creating, maintaining, and researching the digital library of tweets.

2.1.1 CS 5604: Information Retrieval

CS 5604 at Virginia Tech is a class which uses a project-based learning approach to teach students about information retrieval techniques. The project-based learning approach makes use of a single, semester-long project to guide students through the learning process [16] [19]. The projects for the past two years have focused on developing systems which utilize the DLRL's archive of over one thousand collections containing a combined total of over 1.5 billion tweets. The class normally functions by grouping the students into teams, with each team being responsible for a small part of the overall project goal. These teams will have to work together under the guidance of Dr. Fox and the class GRAs to come up with a way to use the resources provided by the lab to accomplish an overarching, semester-long project goal. Reviewing these projects provides an overview for the requirements of a wide variety of the kinds of research projects that would be conducted in the DLRL. When reviewing these projects, special attention was paid to work that was replicated between multiple projects, as this is the kind of functionality that would save a significant amount of development time if supported by the framework. Several key issues were uncovered.

Data I/O

Every team had some kind of data input and output that had to be developed, documented, and coordinated with all of the other teams working on the project. Often, the "Collection Management" teams would be the ones responsible for reading in raw data and processing it into a format that the other teams would be able to work with more easily. The most recent team read the tweet data in the form of raw MySQL databases, and wrote the data to HBase tables for later reference by other teams [1]. The Spring 2016 team [20] loaded the raw relational data into the Hadoop Distributed File System (HDFS) using the Sqoop tool provided by the

Apache Hadoop software stack [31]. Once the data was processed into HDFS, the team needed to read those newly created files, clean and process them, and then store them back in HBase.

In addition to the collection management teams, many other teams had need of various kinds of Input/Output functionality. For example, several teams during the Spring 2016 class needed to read `.tsv` data files that had been cleaned by the collection management team, alter or analyze them, and then store their results in HBase [7] [21] [32]. The following semester, several teams decided to do all of their Input and Output via HBase tables [4] [7]. In both classes, each team was responsible for writing their own code for accessing the data and writing it to its destination.

From all this, two main points can be reached. First, teams in large projects like this generally work with other teams by reading data in from a shared source (like an HBase table or a set of source files), processing the data, and then writing their results back to another (or the same, in the case of HBase) shared source. Reading and writing data are widely known to be expensive processes. If the collection of data could be held in a data structure in memory, the process would see a potentially massive increase in efficiency. Second, each team ended up writing their own versions of data reading and data writing code to suit their needs. For each team to spend time writing such similar code seems like a waste of developer hours. If there was a common way provided to them to read and write data in the formats they need, the development process would see more rapid progress towards solving the problem the developers set out to solve, since less time is needed to learn how to access the data in the first place to start experimenting. Both of these points were taken into consideration with many of the design choices made during the development of the framework. They also provide some justification that a system like this would benefit developers working on DLRL research projects.

Cleaning Tweets

The next topic that emerged frequently was the need to clean the text data. The teams that did the most cleaning were generally the collection management teams [1] [20], since they were responsible for taking the raw text input, processing it into a workable format, and maintaining that content for use by the other teams. These teams described several steps which they took to have a clean, easily usable set of tweets. This includes things like filtering out profanity and spam, removing stop words, and removing non-ASCII characters. These teams were also responsible for extracting information such as URLs, hashtags, and mentions from the tweet content for use by other teams later. Some project teams went further and applied more advanced cleaning and preprocessing techniques. Some examples include lemmatizing the data to prepare for classification [7] and removal of irrelevant content to prepare for clustering and topic analysis [4].

These papers serve to highlight that cleaning is often an essential pre-processing step. It must be performed on raw text data before any analytics are performed, or the results may be inaccurate (if they exist at all). The framework will take this into account by enabling cleaning processes to be done quickly and easily.

Spark/Hadoop Learning Curve

The last topic of interest gathered from the project reports was the amount of time teams dedicated strictly to learning Spark and Hadoop (and their associated tools) or reworking code that was written in some other environment to work with the

suite of tools on the cluster. There were teams who needed time to understand what tools were available and how to use them to address the problem they were trying to solve. Teams with some prior experience were able to accomplish this in one or two weeks [20] [8], while teams with little to no experience working in environments like this took three or more weeks to learn the tools and techniques [8]. There were also some teams who decided to do their development in an environment that was easier to set up and interface with, such as the Cloudera Virtual Environment [5], and then push their implementation back to the DLRL cluster once they were comfortable with it [8] [7] [4].

The amount of time teams spent learning the software tools or working with other systems to avoid having to learn the tools right away indicates that the Spark and Hadoop suites have a very steep learning curve. Developers of varying abilities and backgrounds will handle this learning curve very differently. Some will be able to quickly become good at the tools and use them to their full potential, while some may spend several weeks learning how to use the tools. With this in mind, a priority of the framework is to attempt to smooth out this learning curve without hindering the ability of developers who are more comfortable with the tool suites to still make use of the more advanced features of Spark and Hadoop.

Summary

These class projects provide a lot of insight into what developers aiming to do research involving the DLRL's tweet archives may struggle with. The students entering these classes often have little to no background knowledge of the Apache Hadoop and Spark tools, and need to learn them quickly so as to not fall behind in the semester project. Addressing the needs of these students will go a long way towards addressing the needs of DLRL researchers in general. The three main points discussed above all came together to form the key ideas behind the design of the framework.

2.1.2 IDEAL and GETAR

The archiving work done in the DLRL is motivated by two NSF-funded projects: the Integrated Digital Event Archiving Library (IDEAL) project [15], and the Global Event and Trend Archive Research (GETAR) project [14]. The IDEAL project uses digital library and archive technologies to support researchers who are studying important events that manifest on the Internet. The GETAR Project aims to create digital library and archive systems to help researchers analyze trends in global-scale events that have been occurring over the past two decades. Both of these grants have spawned a number of projects within the DLRL related to digital libraries and archives. Enabling and simplifying further research within these grants is a primary goal of the framework. By making it easier to interface with the archives maintained by DLRL, the framework enables research to be done faster and more easily by researchers with varying backgrounds.

2.2 Technical Specifications

This section presents a summary of some of the hardware and software available to developers in the DLRL. It discusses the Hadoop cluster [12] housed in the lab, and components of the Cloudera software stack [5] used therein.

2.2.1 Hadoop

Apache Hadoop is an open source software suite for "reliable, scalable, distributed computing. "[2] The DLRL maintains a computing cluster containing 21 nodes (twenty workers and one head node) for a combined total of 704 GB of RAM and 154.3 TB of disk space [12]. The cluster is designed to be used with Hadoop to run parallel, distributed algorithms on large data sets. The Hadoop software this framework is most interested in is the Hadoop Distributed File System (HDFS). The cluster makes use of HDFS to provide fast and efficient distributed storage of its data archives. Researchers who are interested in working with archived data will have to interface with HBase to access the archived content. The Hadoop tool suite also provides the Avro data serialization system [36]. Avro is a binary format designed to be easily serialized across distributed file systems like the one contained in the cluster. Hadoop serializes text data (like the tweet data the framework is designed to work with) into Avro format when it is archived on the cluster's file system. Developers normally need to de-serialize the Avro format back into normal text in order to use the data.

2.2.2 Spark

Apache Spark is "a fast and reliable engine for large scale data processing" [3]. Spark is included in the Hadoop software suite as a way to process large data sets in memory, and in parallel across a cluster of computers [2]. The cluster uses Spark to enable massively parallel analysis of the data sets contained in its archives. Spark enables parallel processing by abstracting complex and hard-to-write parallel computation code, and instead providing developers with a large suite of tools and data structures that will handle the parallel processing while behaving like local data structures [25]. One such data structure is the Resilient Distributed Dataset (RDD) [26]. The RDD is designed to represent a collection of any kind of data (including user-defined classes) in memory, and provides functionalities to manipulate, filter, and process the data contained within it [28]. RDDs enable parallel processing of the data contained within them by splitting the data into logical partitions and sending each of those partitions to a node in the cluster to be processed [27]. Once the processing is complete, the individual partitions can be collected locally again and treated as one continuous data structure. This process is done lazily, meaning that all transformations are queued until some action requires a result to be returned, at which point all transformations so far are executed at the same time and the result is returned [28].

2.2.3 Using Hadoop and Spark

The framework will be designed to work on top of Hadoop and Spark, to apply their functionalities to collections of tweets. Similar to the way Spark abstracts the need for developers to be concerned with parallel programming, the framework will further abstract the need for developers to be concerned with processing raw text data with low-level Spark programs. Instead, the framework will present them with data structures that can be operated upon using provided functionalities, and all interfacing with Spark and Hadoop will be handled for the developer. This will enable developers to operate at a higher level, focusing on working with tweets and collections of tweets rather than raw text.

Chapter 3

Design and Architecture

3.1 Design Goals

3.1.1 Motivating Problems

Our framework is designed to support the research and development activities of researchers in the Digital Library Research Laboratory. There are multiple projects happening in the lab at any given time, many of which revolve around our archive of data collected from Twitter through various methods. With so much going on, a tool to help smooth development would save a lot of time in the long run. To assist in development, this framework aims to address two main issues which come up frequently.

The first major issue is that the Apache Spark and Hadoop tool suites have a very steep learning curve. These are the tools that our hardware and software systems support, and so they are used often. However, learning the basic fundamentals can still be a daunting task. It may take students who are just learning to use the tools for the first time several weeks to begin feeling comfortable working with them, and even longer to gain a strong grasp of how to leverage these tools to their full potential. Smoothing out this learning curve would afford researchers more time to focus on the problems they are trying to solve, since they could spend less time learning the tools. This framework solves this issue by abstracting some of the more complicated aspects of working with Spark and Hadoop, and handling them for developers.

The second issue is that there is no well established "best" way to handle reading, writing, and processing of our archived tweet collections. Whenever a new project begins, the first step is frequently to figure out how to retrieve data from the archive, decode it into a workable format, and clean away any unwanted content. There are some tools available to do this including interfacing with Spark and manually transforming the text, using Python scripts and libraries, and using .pig scripts to run on top of Hadoop. However, none of these tools are designed to work directly with our data, and so they all require some other setup or additional programming. This framework is designed with the DLRL's digital Twitter library and the needs of the lab's developers in mind. Hence, there is minimal setup and very little code required to get a collection pulled, formatted, and cleaned.

3.1.2 Design Principles

To address these two problems, the framework is designed with a few guiding principles in mind. These principles were formed over time based on the needs of researchers being supported by the case studies presented in Chapter 5. There are three main design principles.

Enable High Level Programming

The first design principle of the framework is to eliminate the need to work with raw text data. This is handled by the data structures provided by the framework, so that developers do not have to interface directly with the software on the cluster to manually parse raw text. Developers should be able to focus on the problems they need to solve with minimal time spent learning the required low level techniques, especially if those low level techniques are addressing problems that have been solved before.

Ease of Access to Tweet Collections

Secondly, the tool should provide developers with quick and convenient access to the collections contained in our archives, as well as a suite of functions to clean and filter the collections. Tweet collections can come from various sources and in various formats. The framework aims to simplify the process of accessing data by making it so that as many of these various sources and formats of data as possible can be processed into one uniform data structure. The currently supported formats and discussion about extending the framework to support new formats of data are documented below in the *Tweet Data Structures* section of this chapter. In addition, cleaning of data is often a first step in many text mining problems, so performing that first step for developers will accelerate the process of addressing the problem that they are trying to solve.

Support of Tool Development

Finally, the framework will support the development of tools to be used by other researchers in the future. A tool in this context refers to anything that can take a tweet collection as an input, process it, and produce some output. Examples might include classification tools, topic modeling techniques, and feature extraction tools. By designing tools to work with these data structures rather than with raw text data, we can make it easier for the tools to support arbitrary collections of tweets, rather than requiring whatever text format the developers happened to be working with when the need arose for a tool to process the data.

3.2 Design Overview

As mentioned previously, the main way this framework aims to help developers is by eliminating the need to interface directly with the software on the cluster as much as possible. This is accomplished primarily by the creation of data structures to represent tweets and collections of tweets. By organizing the raw data using such data structures, we are able to represent the text content in a higher-level, easier to work with manner. The two primary data structures are `Tweet` and `TweetCollection`. The `Tweet` data structure contains fields which represent important information about tweet content, as well as functionality to clean the text content and extract important features. The `TweetCollection` data structure provides a higher level representation of a collection of tweet objects, and provides functionality to clean and filter the collection, etc. By representing tweets and collections of tweets in this way and "black-boxing" the code which reads and deciphers the data files, cleans the individual tweets, and filters the collection, we can speed up development and

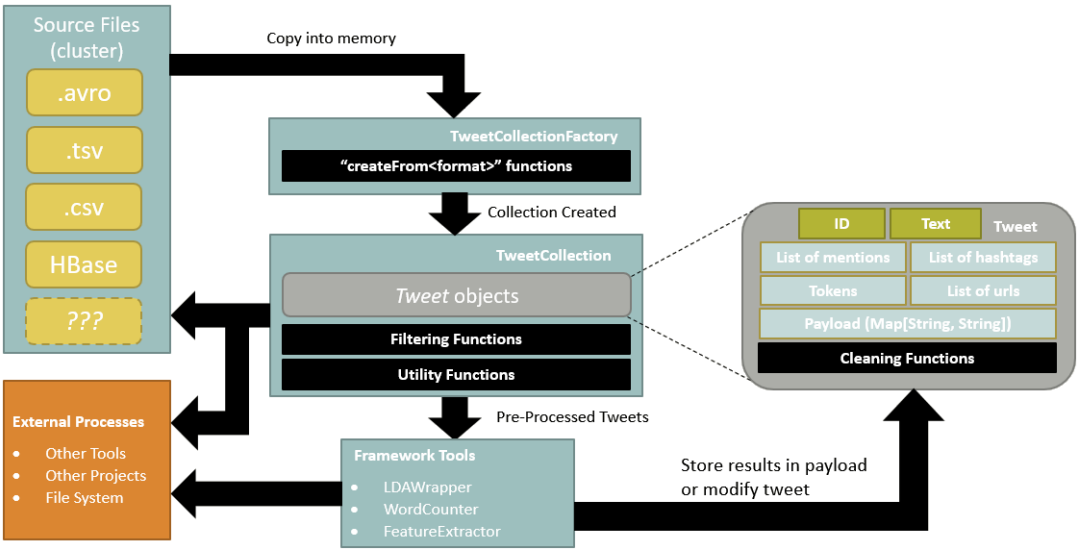

FIGURE 3.1: The flow of data through the framework, from raw data on disk to pre-processed data to results of analytics tools

ease the learning curve of future projects that need to work with collections of tweets from the lab's archive, local files, and other sources.

3.2.1 Data Flow and Scope

Figure 3.1 shows an overview of how data flows through the framework. The goal is a seamless flow from files stored on disk to a cleaned and processed data structure which can be written back to disk or read in by analytics tools which are designed to take `TweetCollection`s as parameters. Those tools can then use the data structure rather than the raw text to perform analyses. We break down the flow step-by-step here, highlighting important details that go into each step.

Step 1: Reading Data from Disk

As mentioned previously, data sets can come from a multitude of places. Some examples include .avro archive files, plain text files, and "separated values" files (.csv or .tsv). If a developer has direct access to the files, they can specify a path to the appropriate `TweetCollectionFactory` build method to have the file read from disk. `TweetCollectionFactory` also provides functionality to simply specify a collection ID number as given on the DLRL Hadoop Cluster Page [11]. The framework will then handle reading the data out of the Hadoop File System (HDFS) on its own.

Step 2: Parsing into Tweet Objects

Once the file has been located and opened, the `TweetCollectionFactory` instructs Spark to read the contents of the file and parallelize its contents across the cluster. This allows for massively parallel (and therefore very fast) reading and processing of the data contained in the file. The tweet data is then remapped from its original format and wrapped in the appropriate `Tweet` implementation. Each `Tweet` object is designed to automatically process any metadata it can gather about the tweet based on the data provided. The exact metadata gathered depends on

which `Tweet` implementation is being used. Every implementation is required to extract (or attempt to extract) hashtags, mentions, and URLs; determine whether or not this is a retweet; and break the raw text into tokens. More specific implementations which extend the base `SimpleTweet` class (explained below in the *System Architecture Details* section of this chapter) may yield additional metadata based on what is provided by the data it is reading in.

Step 3: Cleaning and Filtering the Collection

Once the collection is created and populated with tweet content, developers can manipulate the collection in a variety of ways to clean and filter its contents. There are several built-in functions which can be used to filter the contents of the collection, removing unwanted elements based on certain conditions. Developers can also define custom filter functions which take tweets as parameters and return a Boolean value to tell the collection whether or not to keep that tweet. The framework will handle packaging the function and distributing it across the cluster to be run on every `Tweet` in the collection.

Step 4: Running Analysis Tools on the Data Structures

Once the developers have cleaned the tweets and filtered the collection to their liking, they can proceed to pass the collection on to the various tools. The framework provides several tools, as well as ways to dump the cleaned collection to a file to be passed on to external tools. Tools which are designed to work with the `TweetCollection` data structure rather than raw text data are able to use its functionality to their advantage. They may do more of their own cleaning or filtering, and/or use `TweetCollection`'s `applyFunction` functionality to apply some analysis to every tweet contained in the collection. For example, the `LDAWrapper` (explained in Chapter 4) defines a function which, once topic analysis is completed, takes each tweet, adds a set of topic labels to it, and returns it to the `TweetCollection`. The topic labels are stored in the `Tweet`'s payload field.

Scope

It is important here to consider the limits of the scope of the framework as well. In simple terms, as long as tools connected with the framework are doing their work within `TweetCollection`'s provided functionalities, they are within the scope of the framework. That being said, `TweetCollection` does provide developers with access to the Spark data structures with which it wraps around and interfaces. Some developers may want to access these inner data structures directly rather than via the framework's wrapper functionalities to do a more "low-level" analysis using some more advanced features of Spark. Once this happens, though, they are operating outside of the scope of the framework and are responsible for handling the results on their own. Note, though, that in some cases it is possible to use these data structures to do some analysis and then write the results back to the individual `Tweet`s within the collection. For example, the `LDAWrapper` tool explained in Chapter 4 does this to give each tweet a set of topic labels while also returning an array of overall topic analysis results.

Name	Type	Description
text	String	Unfiltered text of the tweet
id	String	Unique ID of this tweet
isRetweet	Boolean	True if the tweet is a retweet
tokens	Array[String]	Tweet text broken up into tokens
hashtags	Array[String]	Array of hashtags found in tweet text
mentions	Array[String]	Array of mentions found in tweet text
urls	Array[String]	Array of URLs found in tweet text
payload	Map[String, String]	Payload for holding extra data read in from files or derived by tools

TABLE 3.1: Fields represented in the Tweet data structure. Text and ID are required as constructor parameters; the rest are derived based on those

3.3 System Architecture Details

3.3.1 Overview

The system architecture is designed to be modular and extensible. The intention is that developers will be able to extend it to suit their needs and the needs of other researchers doing similar work in the future. Several key design decisions were made with this goal in mind.

3.3.2 Tweet Data Structures

The fundamental data structure of the whole framework is the `Tweet` Data structure. `Tweet` is an abstract data structure; it is not meant to be directly instantiated. There are multiple reasons for this which are discussed in detail in Chapter 4, but the most important reason is that `Tweet`s are constructed very differently based on the format of the data they are being built from. The framework deals with this by representing a tweet in an abstract, high level manner by defining a set of fields in the data structure which all tweets will have in common. It is the responsibility of the concrete implementation of the abstract `Tweet` class to populate those fields once the data is passed to its constructor. In addition, `Tweet` requires some basic cleaning functions and utility functions. Most of the functions are abstract, to be implemented by subclasses, but it does provide base functionality for the `toString` and `equals` functions. Details about the fields are shown in Table 3.1, and details about the required functions are shown in Table 3.2.

It is also worth noting that `Tweet` requires text content and a string ID as constructor parameters. The text content should represent the text written in the tweet. The ID is a unique identifier to represent this tweet. The ID could be provided by the source of the data (for example, Twitter provides unique IDs for every tweet [35]) or created by the developer, but they should be unique to each individual tweet for reasons discussed in Chapter 4.

There are several concrete implementations of `Tweet` provided by the framework, which are discussed in detail here. A long term goal for the framework is to make it easy for developers in the future to write new implementations of `Tweet` to meet their needs if the current set of implementations is not compatible with their data.

Function	Description
cleanRTMarker	Remove the RT marker
cleanMentions	Remove mentions from tokens
cleanHashtags	Remove hashtags from tokens
cleanURLs	Remove URLs from tokens
cleanPunctuation	Remove non-alphanumeric characters except @ and #
cleanRegexMatches(regex*)	Remove tokens matching specified regular expression
cleanRegexNonmatches(regex*)	Remove tokens that don't match specified regular expression
cleanTokens(Array[String])	Remove all instances of specified tokens
toLowerCase	Turn all text lowercase
addToPayload(key: String, value: Any)	Add a key and value to the payload
toStringVerbose	String representation of all data about tweet
toString	String representation of tweet in form "id \t tokens"
equals	Returns true if comparing two tweets with the same ID

**scala.util.matching.Regex*

TABLE 3.2: The functions required by the abstract `Tweet` class. `toString()` and `equals()` are implemented

SimpleTweet

`SimpleTweet` is the basic implementation of `Tweet`. It provides a concrete implementation of everything required by the abstract class. It can be thought of as the most straightforward and simple (hence the name) representation of a tweet. A `SimpleTweet` requires only an ID and text content to be constructed. When provided with these, it will automatically mine the text content for hashtags, mentions, and URLs; determine if the tweet is a retweet based on the presence of an RT token; and split the text up into tokens for later processing. It also provides implementations of all of the abstract methods required by `Tweet`. Since it handles generation of all of the fields and implements all of the abstract methods, `SimpleTweet` serves as both a rudimentary tweet representation and a base concrete implementation for other, more complex tweet representations (like `AvroTweet`, `SVTweet`, and `HBaseTweet`), discussed next.

AvroTweet

`AvroTweet` is an extension of `SimpleTweet` that extracts content and metadata from `.avro` files. It takes a single `GenericRecord` [18] as a parameter, from which it then pulls all of the necessary data. The framework, given a file, automatically handles deciphering the binary Avro data format and building a `GenericRecord` for each tweet in the file, which then gets parsed into the data structure for ease of access to all of the data. All of the data present in an `AvroTweet` is explained in Table 3.3.

SVTweet

`SVTweet` is another extension of `SimpleTweet` that builds data structures out of lines in a "Separated Values" (sv) file. These files use a separator character to split lines of data into columns, with each column containing a different part of the data. Common sv files include `.csv` files (**c**omma separated values) and `.tsv` files (**t**ab separated values). `SVTweet` will take these columns and store them in the key/value payload defined in the base `Tweet` class. By doing this, we can support any amount of arbitrary data that may be stored in the files. This is an important functionality

Field	Type	Description
archivesource	String	Archive tweet was pulled from (ex: “twitter-search”)
to_user_id	String	Unfiltered tweet text
from_user	String	Username of user who posted this tweet
from_user_id	String	ID of user who posted this tweet
iso_language_code	String	Short code representing tweet’s language (ex: “en” for English)
source	String	How tweet was posted (ex: Twitter Web, Twitter for Android, TweetDeck)
profile_image_url	String	URL of poster’s profile image
geo_type	String	How tweet’s geo tag is represented (ex: “point”, Empty string if n/a)
geo_coordinates_1	Double	First geo tag component (0 if n/a)
geo_coordinates_2	Double	Second geo tag component (0 if n/a)
created_at	String	Human-readable time tweet was created (ex; “Fri Oct 28 22:16:39 +0000 2016”)
time	Int	Epoch time tweet was created (ex 1477692999)

TABLE 3.3: The fields contained in an `AvroTweet`

to support, since sv files do not have a defined structure like Avro files do. Some will specify the column names in the first line of the file, but this is not required, and so we cannot depend on it to determine the file schema. Instead, we define a new configuration class called `SVConfig` which is passed as a parameter to `SVTweet`.

SVConfig

`SVConfig` is a class defined to be passed as a parameter to `SVTweet`’s constructor. It is used to define several things that are required to be able to parse an sv file, including:

- The separator value that defines the columns
- The number of columns contained in the file
- The column that contains the text of each tweet
- The column that contains the ID of each tweet
- The names of any other columns of interest in the file to be stored in the tweet’s payload.

By defining these as options, the framework provides the developer with maximum flexibility to parse any kind of sv file containing any kind of data.

One other important thing to note about `SVTweet` is that the framework cannot currently handle files where the tweet text contains extra newline or separator characters. This would cause the data to become corrupt since the columns would not match. To handle this, the framework will disregard any lines in the file which do not have the correct number of separator characters to match the number of columns specified in `SVConfig`. The ideal solution would be to combine subsequent lines until the correct number of columns are represented, but due to the way collections are distributed across the cluster by Spark, this would be a very difficult and expensive process. In this case, it is the responsibility of the developer to make sure the data is formatted correctly.

HBaseTweet

At the time of this writing, HBaseTweet is still a work in progress. The plan is to have a config parameter similar to SVTweet which specifies what data is contained in which columns. The data structure will automatically handle connecting to HBase and reading data from the table. Thanks to the Spark API, building a collection of these should be a very fast process even though it will require many lookups, since it is all done in parallel.

Creating New Tweet Structures

Extensibility is a primary goal of this project. Making the creation of new data structures (to meet the needs of future developers) as easy as possible was a high priority when designing the structure of the system. More of the technical information that would be of interest to developers is covered in Chapter 4.

3.3.3 The TweetCollection Data Structure

`TweetCollection` is the second core data structure of the whole system. It is the main structure that will hold all of the tweets parsed out of a collection, and that with which all of the tools designed for this framework will be designed to work. A `TweetCollection` is just what the name indicates: a collection of `Tweet` objects. At its most basic, it is a wrapper around Spark's `RDD` (Resilient Distributed Dataset) data structure [26]. `RDD` is one of the data structures Spark uses to parallelize data sets across the cluster [25]. This massive parallelization allows for huge amounts of data to be processed very quickly, which is exactly what we need to accomplish with the data structure. The `TweetCollection` data structure interfaces with the `RDD` data structure to create a distributed dataset of `Tweet`s. In this way, we have abstracted away the need for the developer to work directly in Spark. Instead, they can simply specify a data source, set up a `TweetCollection` using the `TweetCollectionFactory`, and have the framework handle the various mapping operations and distributing the data across the cluster.

`TweetCollection` offers a suite of ways to clean, filter, and otherwise modify the tweet collection in ways that will be beneficial to the kinds of digital library research that will need this kind of data structure. Filtering functions modify the collection, as opposed to the contents within the collection. Included are ways to remove tweets which meet (or do not meet) specific criteria. Explicit functions filter matches or non-matches based on the contents of a tweet. For example, a researcher may only be interested in tweets within a larger collection that contain a specific hashtag. Maybe the larger event sparked some smaller discussion with the new hashtag and it all was collected together. A developer could very easily load the data into a `TweetCollection`, filter out all of the tweets which do not contain the hashtag of interest, and pass the new sub-collection along to the various provided tools for analysis or write the results to a file to be processed again later. There are also several utility functions, and room to add more in the future.

Table 3.4 shows a list of all of the different built-in ways to format the tweet data and return it to the caller. Table 3.5 lists all of the available filter methods, including the generic "filter" method which takes a function defined by the developer, to accept the appropriate Tweet type and return a Boolean which can be used to build a custom filter function. Table 3.6 shows the currently available utility methods, including the `applyFunction` method which allows the developer to define a function which takes a tweet as a parameter, modifies it in some way, and then returns it to be stored

Function	Returns	Description
getCollection	RDD[T <: Tweet]	Returns the raw RDD of tweets
getPlainText	RDD[String]	Returns RDD containing tweet text only
getPlainTextID	RDD[(String, String)]	Returns RDD of tuples containing the ID and text of each tweet
getTokenArrays	RDD[Array[String]]	Returns RDD of tweet text broken up into tokens
getTokenArraysID	RDD[(String, Array[String])]	Returns RDD of tuples containing ID and tokens of each tweet

TABLE 3.4: `TweetCollection` provides a set of "getter" methods which return the data formatted in convenient ways

Function	Description
filter(t => Boolean)	Filters collection based on result of function parameter
filterRetweets	Removes all retweets from the collection
filterByID(id: String, keepIf)	Removes all tweets with(out) specified ID
filterByMention(mention: String, keepIf)	Removes all tweets which do (or do not) contain specified mention
filterByHashtag(hashtag: String, keepIf)	Removes all tweets which do (or do not) contain specified hashtag
filterByUrl(url: String, keepif)	Removes all tweets which do (or do not) contain specified URL
filterByTokens(tokens: Array[String], requireAll: Boolean, keepIf)	Removes all tweets which do (or do not) contain specified tokens. Can specify whether a tweet needs some or all of given tokens
filterbyPayloadKeyValue(key:String, value: String, keepIf)	Removes all tweets which do (or do not) contain the required value in its payload given the key

TABLE 3.5: `TweetCollection`'s filter methods

back in the collection. In all of these figures, "T" or "t" denotes the generic type parameter, which must be a type that extends `Tweet` so that the data structure knows it will have access to the `Tweet` methods. For both of the mapping functions, the framework handles packaging up the function and distributing it across the entire collection, which is normally a complex task when generics are involved.

Creating `TweetCollections`

The creation of `TweetCollections` requires four things:

1. An ID for the whole collection
2. A reference to the SparkContext
3. A reference to the SQLContext
4. An RDD [26] of Tweets to wrap around

The first three requirements are fairly straightforward. The ID can be any string. It should provide some high level description of what is in the collection (for example: "HurricaneMatthew" or "#Blacksburg"). The Spark Context [29] and SQL

Function	Returns	Description
randomSample(withReplacement: Boolean, fraction: Double)	RDD[T <: Tweet]	Wrapper for RDD.randomSample
sanitize	n/a	Remove tweets with no text content (usually as a result of cleaning)
applyFunction(function: T => T)	n/a	Applies the given function to every tweet in the collection
Union(otherCollection: TweetCollection[T], filterDuplicates: Boolean)	n/a	Combines this collection with the specified collection.

TABLE 3.6: `TweetCollection`'s utility methods

Context [30] are provided to the developer at run time by the Spark runtime environment [27]. They are generally referred to as "sc" and "sqlContext" respectively. These should be passed as constructor parameters to the `TweetCollection` so that they can be referenced later by tools that may need to communicate with Spark.

The fourth parameter is the more complicated one. The collection needs to be passed an `RDD` of `Tweet` objects. At this point, it is important to address why the system is designed this way. Requiring the `RDD` to be created external to the data structure affords developers a lot of freedom. More advanced developers may wish to manipulate the `RDD` using some more advanced Spark functionalities before passing them to the `TweetCollection`. However, since this framework is primarily aimed at developers who wish to avoid working with Spark, this presents a complication. To remedy this, the framework provides a factory class [13] to generate `TweetCollection`s automatically. `TweetCollectionFactory` is designed to use the Spark Context and SQL Context to create `TweetCollections` from several supported sources. Currently, it supports the following:

- Creation from `.avro` files given a path
- Creation from separated values files given a path and an `SVConfig` object
- Creation from an `RDD` of `(String, String)` tuples representing tweet IDs and their corresponding text content
- Creation from a `Seq` of IDs and a corresponding `Seq` of text content
- Creation from an `HBaseConfig` object

The developer can choose whichever one of the creation functions suits his or her needs, and the entire process of creating an appropriate `TweetCollection` will be handled automatically. For example, if the developer wants to create a set of collections corresponding to some `.avro` files stored in the archive, he or she would simply need to instantiate a `TweetCollectionFactory` with a Spark Context and an SQL Context, and call the `createFromAvro` function specifying a collection ID (any string) and the collection number found in DLRL's Tweet Archive DB page [34]. The factory will then handle locating the file, decoding it from Avro format, mapping its contents to an `RDD` of `AvroTweet`s, and creating a `TweetCollection` to return to the caller. Using the `TweetCollectionFactory` to generate collection data structures automatically is the recommended way to create `TweetCollection`s.

Chapter 4

Implementation

4.1 Data Structure Implementation Details

Much of the effort behind this project was spent refining the way the data structures function. Several radically different iterations were considered until the current generics-based approach was settled upon. Early versions of the project relied too heavily on the `TweetCollection` data structure, creating separate implementations for every type of data that may have to be read and processed. Each implementation would extend a base collection class that provided some functionality and required all data to be represented as tuples containing the tweet IDs and text content. A later implementation introduced the concept of `Tweet` data structures to allow for a more complete representation of a tweet, but a large majority of the data processing was still handled by the `TweetCollection` data structure, and a separate collection class was still necessary for every different kind of data. Finally, it was decided to move the processing of the tweet data into the tweet objects themselves, and handle dealing with different kinds of tweets with Scala Generics [17]. A detailed discussion of each of these major implementation steps is presented here to justify the development process that ultimately led to the framework as it stands today.

4.1.1 Initial Implementation

The original intention was to create multiple different tweet collection structures that would be designed to process various different kinds of data files. At the time, the goal was to be able to handle separated values files and `.avro` files stored in HDFS, with a general "catch-all" structure to handle `RDD`s. Within these collections, tweets were represented as `(String, Array[String])` tuples [24] where the first element was the tweet's ID, and the second element was the tweet's text broken up into tokens. The notion of a data structure to represent the tweet content did not yet exist. Figure 4.1 displays a hierarchy of the polymorphism between the collection types.

The implementation at this point was a very naive one. It was sufficient at the time for initial proof of concept experiments, but quickly fell apart due to a few major flaws. The biggest problem, however, was that representing the tweets with nothing more than an ID and the tokens was extremely restrictive. There was no way to represent more metadata about a tweet without storing it somewhere else or extracting and re-extracting it every time it was needed.

The biggest take-away from this version of the implementation was that polymorphism was, as expected, a powerful way to make the data structures more robust and extensible. The base `TweetCollection` class was able to provide most of the functionality that the sub-classes would need. In fact, the sub-classes did not need to do anything other than process the input data into the (ID, tokens) tuple.

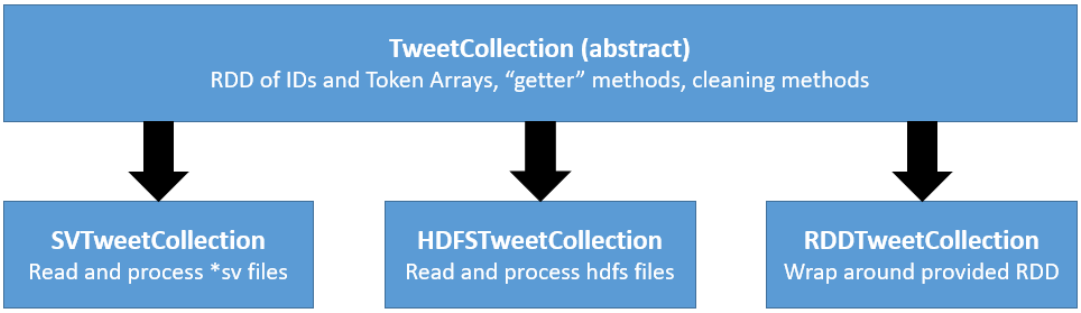

FIGURE 4.1: The first approach to the collection data structures

Once that was done, all of the functionality present in `TweetCollection` would function with no other overriding needed. This was a very flexible approach, and similar polymorphic approaches were used in every subsequent implementation of the framework, including the final release version.

4.1.2 Introduction of the Tweet Class

It quickly became necessary to address the lack of flexibility inherent in using `(String, Array[String])` tuples rather than a more detailed representation. One approach that was considered at this point was to use Spark's DataFrame data structure [23]. DataFrame would allow the framework to specify a column-based structure within each collection. Each collection structure would be able to define an appropriate column structure according to the data it was processing, and maintain the data within those columns as changes were made to the collection using the functions provided by the data structure. Ultimately, this approach failed for several reasons. First, the implementation was quite complicated and required a lot of setup, meaning that creating new collection data structures to represent new kinds of input data proved to be a needlessly time-consuming process when compared to other implementation approaches. Second, DataFrames were not quite flexible enough for what the framework needed to accomplish. Being able to define different columns for different data structures was helpful, but adding new columns to represent data derived after the initial construction of the DataFrame (for example, the topic labels generated by the LDA Tool) was a complicated process that required a fair amount of code. The intention of the framework is to enable developers to manipulate the collection using small amounts of easy to read code, so DataFrames were not a great choice to meet this goal.

Once DataFrames were ruled out, the next approach was to define a Scala class to represent a tweet object. Doing this would afford the system several key advantages, namely:

1. Maximum flexibility for representing tweet metadata,
2. Modular code through defining functions in the tweet classes, and
3. Simplified adding of new data to data structures when tweets implement a "payload".

With this in mind, the framework was refactored to revolve around the newly defined `Tweet` data structure. The original version of this data structure is shown in Table 4.1. Its similarity to the current implementation of `AvroTweet` (recall Table

Field	Type	Description
archivesource	String	Archive tweet was pulled from (ex: “twitter-search”)
text	String	Unfiltered text of the tweet
to_user_id	String	Unfiltered tweet text
from_user	String	Username of user who posted this tweet
id	String	Unique ID of this Tweet
from_user_id	String	ID of user who posted this tweet
iso_language_code	String	Short code representing tweet’s language (ex: “en” for English)
source	String	How tweet was posted (ex: Twitter Web, Twitter for Android, TweetDeck)
profile_image_url	String	URL of poster’s profile image
geo_type	String	How tweet’s geo tag is represented (ex: “point”, Empty string if n/a)
geo_coordinates_0	Double	First geo tag component (0 if n/a)
geo_coordinates_1	Double	Second geo tag component (0 if n/a)
created_at	String	Human-readable time tweet was created (ex; “Fri Oct 28 22:16:39 +0000 2016”)
time	Int	Epoch time tweet was created (ex 1477692999)
tokens	Array[String]	Tweet text broken up into tokens
payload	Map[String, String]	Payload for holding extra data not covered by the fields

TABLE 4.1: The first implementation of the `Tweet` data structure

3.3) is worth noting. This implementation was based on the same fields upon which `AvroTweet` is now based. The intention was to mirror these fields, since they provided a very thorough description of tweet content that a wide variety of projects would potentially be able to leverage. This approach eventually proved to be rather naive as well, since not every data source would be able to populate all of the fields, or even contain enough data to derive all of them.

With the introduction of the `Tweet` data structure, the framework started to look similar to the way it does today. The overall structure of the framework at this point is shown in Figure 4.2. This iteration had multiple advantages over the first iteration. The data representations were more complete and encompassed more of the content that could be extracted from the source files. More core functionality was contained in the tweet classes, which increased extensibility. And, finally, the tweet classes contained a "payload" data field, which provided a way to store arbitrary key/value data. However, even with all of these newfound advantages, there was still room for improvement. First, having to create a new tweet data structure and a new collection data structure for each new type of data source was an overly complex process that could be avoided with some more clever design. Second, having several different kinds of collection structures that all accomplished ultimately the same thing was a needlessly clunky design with a high amount of reused code. These two issues highlighted the weaknesses of the multiple data structures approach. Even though having multiple data structures to represent multiple types of data proved to be convenient for tweets, the same was not true for the collection structures.

4.1.3 Applying Generics

To address the issue of redundancy and to simplify development of future data structures, a design was needed that would avoid having a different `TweetCollection` data structure for each new `Tweet` implementation. With this in mind, the `TweetCollection` data structure was redesigned to be a generic data structure [17]. Generic data structures are structures which take a type as a parameter. In this case, the type parameter is used to define which type of `Tweet` will be contained within this collection. The `TweetCollection` now only needed one implementation. That implementation would take a type `T` as a parameter, with the requirement

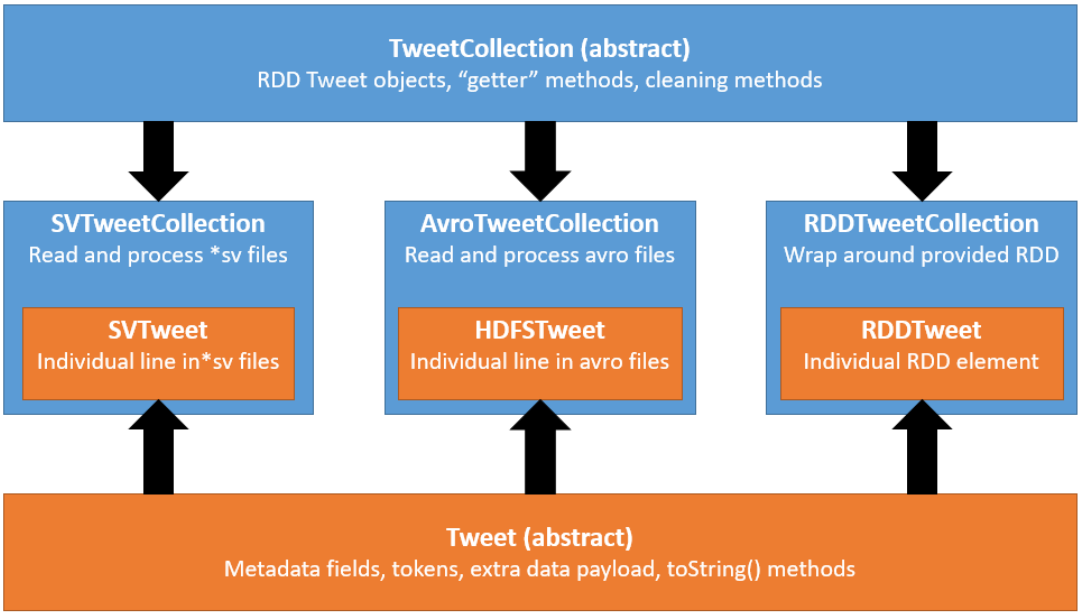

FIGURE 4.2: The framework with the new inclusion of the `Tweet` data structures

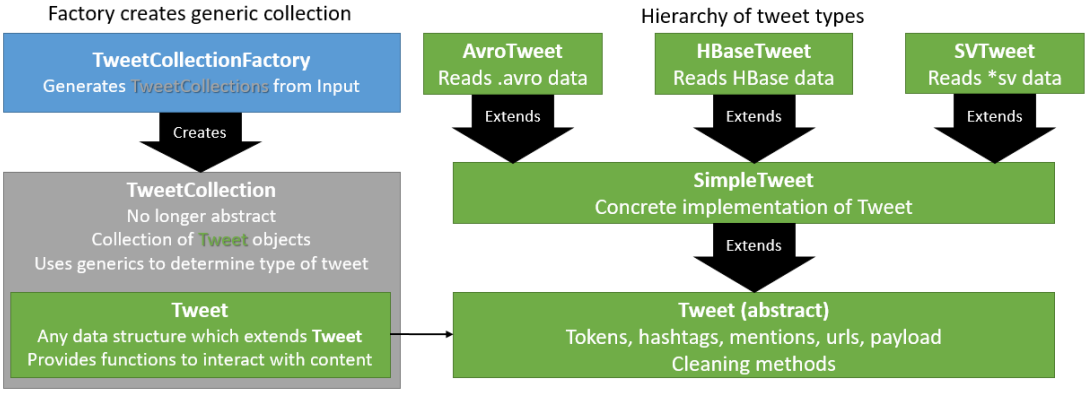

FIGURE 4.3: The current design of the framework, after the introduction of generics

that `T` is any type which extends `Tweet`. The layout of this new framework is shown in Figure 4.3.

This approach has several advantages. First and foremost, it is much more easily extended than any previous approach. The base `Tweet` class is abstract, and requires a set of functions and fields that rely only on the text data and ID of the tweet. This means that very little data is required in order to have a basic representation of a tweet, and therefore even minimal data sources will be able to be represented by these data structures.

Second, the framework is easy to extend thanks to its highly polymorphic approach. All that is required for a new tweet representation to be accepted by the `TweetCollection` is that it extend the abstract `Tweet` class. This can be done by extending the `Tweet` class directly, providing functionality for each of the abstract methods, and providing content for each of the fields. However, it is even easier to extend the framework by extending the `SimpleTweet` class. `SimpleTweet` is the most basic implementation of the `Tweet` class. It provides implementations for

each of the functions and fills all of the necessary fields using only text content and a string ID. `AvroTweet`, `HBaseTweet`, and `SVTweet` all extend `SimpleTweet` rather than `Tweet` and leverage its implementations of the functions and fields, while also providing additional data fields and functionality on top of the basics required by `Tweet`.

Finally, the use of generics allows the `TweetCollection` class much more flexibility. First, it allows the class to make some assumptions about the contents of the collection by requiring that the only types allowed to be stored in the collection be types which have `Tweet` as a superclass. This allows it to implement several filter methods to make it easier for the developer to prune the collection of unwanted content. Generics also allow `TweetCollection`s to safely implement certain functions which rely on other functions. Since Scala is a functional programming language [37], the ability to define functions to map across data collections to be run in parallel by Spark is a very powerful functionality to have, and affords developers a large amount of freedom in allowing them to manipulate the tweets contained in the collection. Developers have the ability to define functions which take and return an object of the generic parameter type `T`, and have the collection handle mapping that functions across all of the tweets in that collection using Spark.

4.2 Framework Tools

As discussed previously, one of the goals of the framework is to support the development of tools which use the data structures to run analyses on collections of tweets. These tools will be able to work with the data without having to interface with Spark and without having to manipulate raw text data. The data structures will provide data for them to process without having to extract it manually. This section details the implementation of three tools and explains how they highlight different capabilities of the framework. The explanations aim to provide some insight for future developers who want to create a tool with this framework.

4.2.1 Basic Processing of Collections

The first provided example tool is the `WordCounter`. This tool was created very early in the framework's development as a proof of concept. This tool was always the first example tool that would be reworked during a new iteration of the framework since it is a very straightforward application of the framework. The tool is designed to take a collection and create a data set of `(String, Int)` pairs representing each word present in the collection and the number of times each word appears. This is accomplished with a function called `count()` which takes a `TweetCollection` as a parameter, and counts all of the words in the collection. This example is simple because it does not use any of the functionalities of the framework beyond taking the `TweetCollection` data structure as a parameter. This basic utilization of the collection is still powerful because it allows the tool to assume that the collection has already been cleaned and filtered in a way that suits the interests of the developer.

Figure 4.4 shows the `WordCounter` class, which contains the implementation of the word counter example. There are a few things to note about this code. First, the `count` method takes a `TweetCollection` as a parameter and returns an `RDD[(String, Int)]` as a result. As discussed, taking `TweetCollection` as a parameter allows the word counter to operate on a collection containing any kind of `Tweet`

```
class WordCounter() {

    import org.apache.spark.rdd.RDD
    import java.io._

    def count(collection: TweetCollection) : RDD[(String, Int)] = {

        return collection.getPlainText()
                .flatMap(line => line.split(" "))
                    .map(word => (word, 1))
                    .reduceByKey(_ + _)
                    .sortBy(_._2, false);
    }

    def writeCountsToLocalFile(path: String, counts: RDD[(String, Int)]) = {
        // Write the results back to local disk using standard java io
        val countFile = new File(path)
        val bufferedWriterCounts = new BufferedWriter(new FileWriter(countFile))
        println("Writing counts to local file")
        for(count <- counts.collect()) {
            bufferedWriterCounts.write(count._1 + "\t" + count._2 + "\n")
        }
        bufferedWriterCounts.close()
    }
}
```

FIGURE 4.4: The provided word counter tool

implementation since it can assume that all of the data has been processed out of the format it was originally stored in and represented as data structures rather than plain text. The other important point to note is that the `count()` method does not manipulate the collection at all and does not return a modified collection since that is not the intention of this algorithm. Instead, it returns a data set of words and their corresponding counts. Third, note the `writeCountsToLocalFile()` function. This function is designed to write the results of the count method to a local file in some usable format for reference later. To accomplish this, the function takes a file path and an `RDD[(String, ID)]`, which is the same thing that the `count()` method returns. Creating functions to write analytics results to a file is a suggested functionality for all tools.

4.2.2 Creating and Extracting Metadata

The next included tool is the `FeatureExtractor` tool. This tool provides functionality to extract specific metadata details about tweets and return them as pairs of tweet IDs and content. For example, the developer might want to gather all of the mentions, hashtags, or URLs from a `TweetCollection` and hold them in one place for further analysis. Some of the possible outputs are shown in Figure 4.5.

Figure 4.6 shows three of the functions provided by the Feature Extractor. There are two things that are worth noting about the way these functions behave. First, the `TweetCollection` parameters enforce a generic type. The syntax `TweetCollection[_<:Tweet]` is Scala syntax meaning "A `TweetCollection` containing any class which extends Tweet." This specification is required to avoid compile-time errors since the functions need to make direct reference to fields contained within the `Tweet` data structures. The second thing to note, which is related to the first, is that these functions are able to directly access the `Tweet` class fields to form their result sets. This will enable tools to reference and even manipulate the data contained within the tweets to perform whatever kind of analysis the tools wish to perform.

```
279655215194587136  #Newton #BREAKINGNEWS
279655215140044800  #Connecticut #Newtown #School
279655214976487424  #RIP
279655214150197248  #LordHelp
279655214066323457  #Newtown
279655213953056768  #PrayForNewtown
279655213948882945  #Newtown
279655213453950976  #NewReport #Connecticut
279655213080649729  #sad #prayersgoout
279655213030338560  #PrayForNewton
279655211365191680  #fb
279655211251924992  #Connecticut #shooting
279655211235151872  #guncontrol
279655211084161024  #Connecticut
```

```
279655215773384704  @Telegraph
279655215144251394  @BloombergNow
279655214900994050  @USATODAY
279655214125031424  @TheAtlanticWire @ABC
279655213948882945  @Willdenario
279655210568257537  @nytimes
279655209519706114  @CNNmobile
279657716010586112  @BarackObama
279657715662471168  @foxnews
279657713250754560  @AP
279657712931987457  @MariaGodga
279657712898420736  @BarackObama
279657712810348546  @CNN
279660363836952577  @TheAtlanticWire @ABC
```

```
279655214699671552  http://t.co/yTLvl2fT
279655214125031424  http://t.co/KJvvT7Z1
279655214066323457  http://t.co/AVUTcUX6
279655213776904192  http://t.co/psw6QbPG
279655213030338560  http://t.co/VZxT8DbD
279655212233404416  http://t.co/ZuiqbBv9
279655212107563008  http://t.co/bMNYRl6r
279655211470036992  http://t.co/kB1X6Hwl
279655211084161024  http://t.co/mKNA4rA8
279655210853482496  http://t.co/8PVxpk0r http://t.co/M8XdVfB8
279655210845077504  http://t.co/nwlQeJcA
279655210639581184  http://t.co/R4TBRZUg http://t.co/xL8IyUjJ
279655210568257537  http://t.co/3xs2f8nG
279655210408882176  http://t.co/WC7ZSJhM
```

FIGURE 4.5: Example results returned by the Feature Extractor tool

```
def extractMentions(collection: TweetCollection[_ <: Tweet]) : RDD[(String, String)] = {
    return collection.getCollection().map(tweet => (tweet.id, tweet.mentions.mkString(" ")))
}

def extractHashtags(collection: TweetCollection[_ <: Tweet]) : RDD[(String, String)] = {
    return collection.getCollection().map(tweet => (tweet.id, tweet.hashtags.mkString(" ")))
}

def extractURLs(collection: TweetCollection[_ <: Tweet]) : RDD[(String, String)] = {
    return collection.getCollection().map(tweet => (tweet.id, tweet.urls.mkString(" ")))
}
```

FIGURE 4.6: The Feature Extractor tool making use of `Tweet`'s provided fields

The power of this functionality is displayed in more depth in the discussion of the next tool, the `LDAWrapper`.

4.2.3 Creating New Metadata

The most complex of the provided tools is the `LDAWrapper` tool. The intention of the `LDAWrapper` is to provide an easy way for developers to perform LDA topic analysis [6] on `TweetCollections`. It accomplishes this by leveraging many of the capabilities of the framework. There are many key points to be highlighted when discussing the `LDAWrapper`, including the ability to:

1. provide default/suggested parameters for complicated analyses,
2. very easily tweak those parameters to optimize results, and
3. read and write metadata to a `Tweet` object while also returning a set of results.

Each of these key points is highlighted by the `LDAWrapper` in different ways. The first two points are highlighted in Figure 4.7, which shows an example of code from one of the case studies that is using the framework to easily optimize topic

```
val ldaWrapper = new LDAWrapper()
ldaWrapper.numTopics = 7
ldaWrapper.termsToIgnore = Array("appalachian", "trail", "at", "atc")

. . .

val topics = ldaWrapper.analyze(collection)
ldaWrapper.writeTopicsToLocalFile(path,topics)
```

FIGURE 4.7: Using the `LDAWrapper` to optimize LDA parameters

```
. . .

var tweetProbabilities = idKey.join(docTopics).map(_._2).collect().toMap

val result: Array[(Array[String], Array[Double])] = topicIndices.map {
    case (termIndices, weights) => (termIndices.map(index => vocabArray(index)), weights)
}

def storeData(tweet: T): T = {
    val topicProbabilities = tweetProbabilities.apply(tweet.id)
    val topicNumber = topicProbabilities.indexOf(topicProbabilities.reduceLeft(_ max _))
    val topicLabel = result(topicNumber)._1

    tweet.addToPayload("topicProbabilities", topicProbabilities.mkString(" "))
    tweet.addToPayload("topicNumber", topicNumber.toString)
    tweet.addToPayload("topicLabel", topicLabel.mkString(" "))

    return tweet
}

collection.applyFunction(storeData)

return result
```

FIGURE 4.8: `LDAWrapper` stores data about each tweet's topic analysis results in the payloads, and also returns overall topic results to the caller

analysis results and expose interesting topics. We can see that the developer is able to manipulate the parameters of the LDA algorithm while leaving the implementation and creation of results up to the framework. The third point is highlighted in Figure 4.8. `LDAWrapper` has two different kinds of results. First, it provides a summary of the topics as arrays of most important terms and term weights. Second, it provides a topic tag for each tweet in the collection. This idea of "overall" results and "per-tweet" results is encoded here. Overall results should be returned to the caller of the function so that they can be further processed or written to a file. Per-tweet results can be written back to the tweet's payload for later reference.

Chapter 5

Case Studies

Since this project was designed with DLRL developers and researchers in mind, several case studies were conducted to test the functionality of the framework, and to gather feedback on its flexibility and ease of use. The case studies cover a wide variety of applications of the framework to justify its design, flexibility, and applications for future work.

5.1 Case 1: Building a New Tool

The first case study involves a project for the class CS 6604: Digital Libraries. This class allowed students to explore projects of their own design within the domain of digital libraries. In the class, I was on a team with Abigail Bartolome, Radha Krishnan Vinayagam, and Rahul Krishnamurthy. The goal of our project was to build a system which takes a tweet collection and performs sentiment analysis within topics. The system functions by performing user-guided topic analysis on a set of tweets, creating sub-collections of tweets which fall within the same topics, and performing sentiment analysis on those sub-collections. The end result is a set of these sub-collections, which can be used to gather insight into public opinion about these events.

The first step in the process is to load the collections and clean them. This process is accomplished quickly and easily with the help of the framework. Since the tool being created for this project is specifically designed to work with the collections archived on the DLRL cluster, it was straightforward to use the `TweetCollectionFactory` to generate collections of `Tweet`s by simply specifying a collection ID and a collection number. The collection ID was a string describing the contents of the collection (for example "Winter Storm", "Blacksburg", or "Hurricane Matthew"), and the collection number was the number corresponding to that collection on the DLRL Tweet Archive DB web page [34]. There was a point in the development process when the tool was being tested with local files. The ability to easily switch between data sources without having to manually read new data into the Spark tools proved very useful in this regard. Cleaning was also very straightforward, and was accomplished using the framework's provided cleaning functions. Specifically, we applied `cleanStopWords()`, `cleanRTMarker()`, `cleanMentions()`, `cleanURLs()`, `cleanPunctuation()`, and `toLowerCase()`. This resulted in a collection that was well-suited for topic analysis.

The next step was to perform topic analysis, which was also accomplished with the help of the framework. The framework provides the `LDAWrapper` tool, which is a tool that applies Spark's built in LDA tool [9] to `TweetCollection`s. The project's user interface allows a user to adjust the topic analysis parameters to refine the results, and to specify terms to completely disregard from the topic analysis process. The `LDAWrapper` applies a topic label to each of the tweets, which can be

used by the tool to separate the collection into sub-collections upon which to perform topic analysis. Once the topic analysis step is complete, we make use of the `LDAWrapper`'s built in ability to write its results to a local file, and then pass those files along to the sentiment analysis step which is performed with external tools.

This case study highlights two very important points. First, it highlights the ease of use of tools which are designed to be run on the data structures. Spark's LDA tool requires a fair amount of setup, but having a tool which is designed to take a `tweetCollection` and use its functionality to handle that setup for you saves much development time as well as lines of code. Making use of the framework tools saved the developers roughly 25 lines of complex Spark code by performing all of the setup required to run Spark's LDA library, and roughly 20 additional lines of similarly complex code by transforming the results of Spark's LDA analysis into a more human-readable and easier to process format. Recall Figure 4.7 which shows example code that performs LDA analysis on a tweet collection using only five lines of code. In addition, the framework also saved the extra effort of creating a new data structure to map the topic analysis results back to the individual tweets by storing the results directly in the associated tweets' payload fields. The framework also saved the developers many weeks of development time, especially with the topic analysis portion of the project. Radha was a member of the topic analysis team in a previous offering of CS 5604: Information Storage and Retrieval. According to his feedback, it took that team most of the semester to get Spark's LDA tool functioning. Abigail was also in a previous offering of CS 5604 where she and her team were working on a topic analysis tool based on the code implemented by Radha and his team in the prior semester. According to Abigail's feedback, it took her team several weeks to get LDA running, even with Radha's team's code guiding their efforts. This semester, with the help of the framework, the CS 6604 team was able to get LDA running in a single afternoon. The ability to save massive amounts of time like this is one of the main strengths of the framework. Work that has been done in the past should be easily accessible and reusable.

The second major point demonstrated by this case study is the flexibility that the framework provides to projects like this. For example, at one point during the project's development, the decision was made to switch from implementing the system on the DLRL cluster to implementing on a local Cloudera virtual machine. This meant that testing now needed to be done with local `.tsv` files rather than the cluster's archive files. Since the project was making use of the framework and not manually deciphering data files, only one line of code needed to be changed to read in the new data. Everything else still worked with no issue even though the files were completely different and provided different data because the analysis was being performed on the data structures rather than raw text data. This saved roughly a day's worth of work that would have been needed to rewrite all of the code which reads in the data, and to rework the analysis code to function with the new format.

5.2 Case 2: Applying Provided Tools

The second case study is Abigail Bartolome's initial work towards her thesis project, which focuses on linguistic analysis of tweets that discuss the three major hiking trails in the United States: the Appalachian Trail, the Continental Divide Trail, and the Pacific Crest Trail. Abigail is still refining her approach to her thesis work, and so she was interested in seeing what the topic models of her collections would look like. She used the framework's `LDAWrapper` in a similar manner as the CS 6604 case

```
ldaWrapper.numTopics = 7
ldaWrapper.numIterations = 100
ldaWrapper.termsToIgnore = Array("continental", "trail", "cdt", "cdnst", "divide", "hike", "cdtc", "#cdt")
. . .
ldaWrapper.termsToIgnore = Array("pacific", "trail", "pct", "pcta", "crest", "hike", "#pacificcresttrail", "#hiking", "#pct")
. . .
ldaWrapper.termsToIgnore = Array("appalachian", "trail", "at", "atc", "hike", "appalachiantrail", "#appalachiantrail", "hike", "#hike")
```

FIGURE 5.1: Optimized LDA parameters for topic analysis on the hiking trail data sets

study. She was able to leverage the tool's flexibility to refine the LDA parameters and generate meaningful topics that would help her focus her research. Figure 5.1 shows the parameters that were ultimately used to arrive at a meaningful set of topics. In the future, these parameters would be defined in a configuration file, rather than directly in the code, but this suited Abigail's needs and allowed her to quickly make the changes she wanted to make.

Abigail was interested in using the tool because she knew what datasets she wanted to study, but she was not entirely sure what she wanted to learn from them. This tool allows her to learn about each trail community's vocabulary and the topics that they care about. She is planning on using the framework more in the coming months to help make the process of loading, cleaning, and analyzing collections easier. She plans on working with many different collections and adding more content to each of them as time goes on. Based on feedback she has reported, the framework has already saved her several weeks of development time by eliminating the need for her to run topic analysis manually. She anticipates that the framework will save her several more weeks of development time in the future by allowing her to use pre-existing tools to explore the data sets.

5.3 Case 3: Exploring Data Sets

The third case study explores Islam Harb's work with sets of tweets discussing Hurricane Matthew. This case is particularly interesting for two reasons. First, the Hurricane Matthew data sets are quite large, some of the largest in DLRL's archives [11]. Data sets of this magnitude push the limits of the cluster's resources and test the efficiency of the framework. Second, Islam is not looking for a specific type of analysis. Rather, he is seeking to learn anything he can about the data set. To accomplish this, he sought to run all of the available tools provided by the framework on each individual collection and search through the results for anything that might be interesting to the project.

Islam was unable to find time to write code himself due to other obligations this semester, but I was able to run the files he provided through the framework tools and give him a set of results. There were four provided data files, ranging in size from just under 153,000 tweets (roughly 55.5 MB) to over 200,000 tweets (roughly 74.5 MB). Each of the files was run through five processes:

1. Word Count - Find the most common words mentioned in the collection
2. Hashtag Extraction - Find the hashtags that appear in the collection
3. Mention Extraction - Find the mentions that appear in the collection
4. URL Extraction - Find the URLs that appear in the collection
5. LDA Topic Analysis - Find the topics that are being discussed

Each of these analyses was completed, and produced results within a useful amount of time. The extraction processes were the fastest, completing in under a minute each. The topic analysis was the longest, but still completed in only a few minutes thanks to the power of the cluster and the ability to process the collection in a massively parallel manner. In this case study, the suggested parameters provided in the default `LDAWrapper` were refined to suit our environment. Namely, it was decided to use 100 LDA iterations because with large files like this, going above 100 begins to massively slow the process down. The jump from 100 to 200 slows the process to a few hours rather than a few minutes.

The result files were given back to Islam for review, and he was quite satisfied with the ability of the framework to deliver a variety of results in a short amount of time, and with minimal code needing to be written.

5.4 Case 4: Building Metadata

The final case study documents Liuqing Li's work towards extracting the URLs contained in tweets and expanding them into full length URLs. Twitter's API automatically shortens every URL that gets posted in a tweet to a "t.co" link [33]. The goal is to collect these shortened URLs, expand them, and store them in an HBase table of metadata about the collections. This will be accomplished with the creation of a new tool based on the `FeatureExtractor` tool already included in the cluster. This tool will take a collection and return a set containing an ID, creation time, and short URLs found in the tweet. This case is especially interesting because Liuqing's goal is to be able to create metadata like this for every collection in the archive [11], which will put the robustness of the system to the test.

Liuqing was given a recent version of the compiled framework to run on the cluster, the source code for the framework itself and all of the currently existing tools, and a brief overview of all of the available functionalities. He was then left to use everything provided to create a new tool which would extract the short URLs, expand them, and fetch the Web content from the Internet Archive. Unfortunately, there were some complications with the implementation due to the fact that the worker nodes on the cluster are not connected to the Internet. This meant that the worker nodes would have no way to fetch content from the Internet Archive. With this in mind, Liuqing decided to instead create a tool which would, for each tweet that contained one or more URLs, print its ID, Timestamp, and the URLs it contains to a file. That file could then be fed into other tools which would gather the Web content and archive it for later use.

Liuqing was able to create a tool which accomplished this and functioned on any tweet collection archived on the cluster in two afternoons' worth of work. He was very satisfied with the functionalities provided by the framework, and estimated that the project may have taken up to a week's worth of work without the framework simplifying the text analysis process for him.

Chapter 6

Discussion

This chapter will touch on a few main points that should be addressed. Thus, it preemptively answers some common questions that readers may have about the nature of my work.

6.1 Choice of Tools

The tools provided by this iteration of the framework were designed with two main goals in mind. First is to support the case studies described in Chapter 5. Second is to provide examples of different kinds of tools for future developers to base their work on. For example, LDA was the chosen topic analysis approach because it was natively supported by Spark, was needed by the CS6604 class project case study, and was a complex algorithm that would show off several different functionalities of the framework – all at the same time. Another topic analysis approach could have been implemented just as easily, but LDA met the needs of the project at the time. I would encourage future developers to explore other topic analysis algorithms and write more tools to run the algorithms on `TweetCollection` data structures.

6.2 Supported Data Formats

Similar to the choice of tool implementations, the supported data formats were chosen to meet the needs of development happening at the time. Directly supporting HBase, SV files, and Avro files enables a wide variety of data types to be processed into data structures with very little work. Supporting their creation from Scala sequence collections and pre-made RDD structures will enable any data format to be turned into a `TweetCollection` with only a small amount of extra work from the developer. It would be impractical to attempt to support every single kind of data source, but the provided functionalities encompass a large majority of the kinds of work being done right now. I would encourage future developers to develop new `Tweet` data structures as the need arises, rather than manually parsing the data and forcing it to conform to already-provided implementations.

6.3 TweetCollection Functionality

As in the case of the two previous sections, `TweetCollection` functionality was also motivated by the needs of current projects. In addition, however, it was designed to be easily extended and to support future tweet data structure representations with minimal effort. Hence it was decided to use one data structure with a generic type parameter rather than implement several different kinds of collections. This is also why the `TweetCollection` supports the mapping of arbitrary

functions across the collection. This functionality, in combination with the provided filtering functions which are designed to work with all tweets and give powerful functionality "out of the box," makes the data structure flexible enough to work with any tweet data structures future developers might create.

6.4 Lessons Learned

Throughout the project, there were several occasions where I would start exploring a new problem space and realize that while the framework may have been able to handle the relevant types of problems, it would have been in a clunky or inefficient way. There were multiple occasions where the entire framework had to be refactored to better accommodate a problem that should have been anticipated. The main lesson here is that I should have put more time upfront into explicitly identifying and planning for the problems the framework is trying to address. The other common issue was that of the scope of the framework. Since the intention of the framework is to help researchers solve problems, it was very tempting to spend lots of time implementing tools and functionalities for every project I knew of. Realistically, however, there needed to be a defined scope of what could be developed now and what needed to be left for future work. As the project went on, this scope eventually evolved into directly supporting the four case studies described in Chapter 5 and providing documentation for those who would like to extend the framework in the future to support other problem spaces. A well-defined scope early on would have helped the project be more focused and increased my productivity.

Chapter 7

Conclusions

7.1 Conclusion

This project addresses some issues which the DLRL has been facing for some time, even if they have never been seen as major issues. In the past, developers have used various methods to interface with our tweet archive, and to apply various different techniques to represent the data contained within the archive. This approach has been functional, but not optimal. Different developers doing similar things in different ways can lead to complications when other developers need to study or apply others' work to theirs. In this thesis, I have presented a new approach which, based on the case studies presented in Chapter 5, will help development run faster and more smoothly, facilitate collaboration between research projects, and enable developers to create their work in such a way that would make it easy for other developers to reuse it in the future. It is my hope that this framework will serve to benefit the future research endeavors of those in the lab, and that it will be expanded upon by those who have needs which extend beyond its current functionalities.

7.2 Future Work

This project was designed with future work in mind, since it is meant to support the research efforts of researchers who want to work with the DLRL tweet digital library. With this in mind, I outline several areas where this project could be expanded in the future.

7.2.1 Additional Framework Tools

The most obvious way this project could be expanded in the future would be to use it to create new tools for researchers to use in their projects. One potential tool would save a tweet collection as a set of metadata and a list of tweet IDs. This kind of export tool would be within Twitter's terms of service while allowing other teams to re-populate the collection by collecting the tweet content for each ID. There could then also be a similar tool which could take an exported collection and "refill" it with content pulled from the Twitter API based on the provided IDs.

Other potential tools include more wrapper tools, like the `LDAWrapper` for Spark's LDA implementation. Spark provides a large set of useful machine learning tools in their MLLib library [22]. Any of these could have a wrapper class formed around it to handle running the tool on a `TweetCollection`.

Another potential tool would be a more robust cleaning suite that includes functionalities not currently present. One notable functionality that the framework is missing in its current state is the ability to filter profanity words and other explicit content from the collections. There are also some feature extraction tools which

could be implemented. For example, named entity recognition could be implemented to identify people, places, or organizations within the collection. A similar system could be implemented to identify things like addresses that are not effectively represented by the way the framework currently handles splitting the tweet text into tokens.

7.2.2 Additional Tweet Implementations

The next area which could be expanded upon is the set of `Tweet` data structure implementations. As discussed in Chapter 6, the current set of `Tweet` implementations was motivated by the needs of the projects the framework was supporting as case studies. I believe they form a solid foundation to help support research endeavors, but there are bound to be data formats which may be used in the future which are not currently supported. One example would be supporting other database technologies like MongoDB or MySQL. While those formats could be accommodated by using the "catch-all" sequence based factory method of `TweetCollectionFactory`, it would be beneficial for developers to directly support those formats with concrete implementations of the `Tweet` class.

7.2.3 Use in Future Classes and Research Projects

As mentioned in Chapter 2, this project was designed to be used by researchers in the Digital Library Research Laboratory to help with future projects and research efforts. I would encourage future offerings of CS 5604 and CS 6604 to make use of the framework to simplify their work and help the students spend less time learning how to use the tools, and therefore more time focusing on learning the concepts and addressing the problems presented in the class.

Bibliography

[1] Faiz Abidi, Shuangfei Fan, and Mitchell Wagner. *Term Project: Collection Management Tweets Final Report*. Blacksburg, VA, USA. Dec. 2016. URL: http://hdl.handle.net/10919/73739.

[2] *Apache Hadoop project*. http://hadoop.apache.org/. Accessed Jan 2017. Apache Software Foundation.

[3] *Apache Spark - Lightning-Fast Cluster Computing*. http://avro.apache.org/. Accessed January 2016. The Apache Software Foundation.

[4] Abigail Bartolome, Md Islam, and Soumya Vundekode. *Clustering and Topic Analysis*. Blacksburg, VA, USA. Dec. 2016. URL: http://hdl.handle.net/10919/73712.

[5] *Big Data | Machine Learning | Analytics | Cloudera*. https://www.cloudera.com/. Accessed January 2016. Cloudera, Inc.

[6] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. "Latent Dirichlet Allocation". In: *Journal of Machine Learning Research* 3 (Jan. 2003). URL: http://www.jmlr.org/papers/volume3/blei03a/blei03a.pdf.

[7] Matthew Bock, Michael Cantrell, and Hossameldin Shahin. *Classification Project in CS5604, Spring 2016*. Blacksburg, VA, USA. May 2016. URL: http://hdl.handle.net/10919/70929.

[8] Saurabh Chakravarty and Eric Williamson. *Final Project Report - CLA Team*. Blacksburg, VA, USA. Dec. 2016. URL: http://hdl.handle.net/10919/73713.

[9] *Clustering - RDD-based API*. https://spark.apache.org/docs/2.1.0/mllib-clustering.html. Accessed September 2016. Apache Software Foundation.

[10] *DLRL | Virginia Tech*. http://www.dlib.vt.edu//. Accessed August 2016. Digital Library Research Laboratory at Virginia Tech.

[11] *DLRL Hadoop Cluster*. http://hadoop.dlib.vt.edu/. Accessed August 2016. Digital Library Research Laboratory at Virginia Tech.

[12] *Facilities | Events Archiving*. http://www.eventsarchive.org/node/12. Accessed January 2016. Virginia Tech - Events Archiving.

[13] *Factory Pattern | Object Oriented Design*. http://www.oodesign.com/factory-pattern.html. Accessed August 2016. Object Oriented Design.

[14] Edward Fox, Donald Shoemaker, and Chandan Reddy Andrea Kavanaugh. "Global Event Trend and Archive Research (GETAR)". In: (Nov. 2015). NSF grant IIS-1619028 and 1619371. URL: http://www.eventsarchive.org/sites/default/files/GETARsummaryWeb.pdf.

[15] Edward Fox et al. "Integrated Digital Event Archiving and Library (IDEAL)". In: (Sept. 2013). NSF grant IIS-1319578. URL: https://www.nsf.gov/awardsearch/showAward?AWD_ID=1319578.

[16] Edward A. Fox et al. *Conversation: Problem/project-based Learning with Big Data*. Proc. 2016 Conference on Higher Education Pedagogy, Feb. 10-12, 2016. Blacksburg, VA, USA.

[17] *Generic Classes - Scala Documentation*. http://docs.scala-lang.org/tutorials/tour/generic-classes.html. Accessed March 2017. Scala Community Documentation Team.

[18] *GenericRecord*. https://avro.apache.org/docs/1.7.6/api/java/org/apache/avro/generic/GenericRecord.html. Accessed September 2016. Apache Software Foundation.

[19] Tarek Kanan et al. *Big Data Text Summarization for Events: a Problem Based Learning Course*. Proc. of the Joint Conference on Digital Libraries (JCDL 2015). June 21-25, 2015. Knoxville, TN, pp. 87-90. URL: http://dx.doi.org/10.1145/2756406.2756943.

[20] Yufeng Ma and Dong Nan. *Collection Management for IDEAL*. Blacksburg, VA, USA. May 2016. URL: http://hdl.handle.net/10919/70930.

[21] Sneha Mehta and Radha Krishnan Vinayagam. *Extracting Topics from Tweets and Webpages for the IDEAL Project*. Blacksburg, VA, USA. May 2016. URL: http://hdl.handle.net/10919/70933.

[22] *MLLib: RDD-based API - Spark 2.10 Documentation*. https://spark.apache.org/docs/2.1.0/mllib-guide.html. Accessed January 2016. The Apache Software Foundation.

[23] *Scala Dataset*. http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset. Accessed March 2017. Scala.

[24] *scala.Tuple2*. http://www.scala-lang.org/api/2.9.3/scala/Tuple2.html. Accessed March 2017. Scala.

[25] *Spark API*. https://spark.apache.org/docs/latest/api/scala/index.html. Accessed January 2016. Apache Software Foundation.

[26] *Spark API - RDD*. https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.RDD. Accessed January 2016. Apache Software Foundation.

[27] *Spark Programming Guide*. http://spark.apache.org/docs/latest/programming-guide.html. Accessed August 2016. Apache Software Foundation.

[28] *Spark programming guide - RDD programming operations*. http://spark.apache.org/docs/latest/programming-guide.html#rdd-operations. Accessed January 2016. The Apache Software Foundation.

[29] *SparkContext (Spark API)*. https://spark.apache.org/docs/2.0.2/api/java/org/apache/spark/SparkContext.html. Accessed September 2016. Apache Software Foundation.

[30] *SQLContext (Spark API)*. https://spark.apache.org/docs/2.0.2/api/java/org/apache/spark/sql/SQLContext.html. Accessed September 2016. Apache Software Foundation.

[31] *Sqoop*. http://sqoop.apache.org/. Accessed March 2017. The Apache Software Foundation.

[32] Lijie Tang, Swapna Thorve, and Saket Vishwasrao. *Clustering and Social Network*. Blacksburg, VA, USA. May 2016. URL: http://hdl.handle.net/10919/70947.

[33] *t.co links - Twitter Developers*. https://dev.twitter.com/basics/tco. Accessed April 2017. Twitter.

[34] *Tweet Archive DB, DLRL, Virginia Tech*. http://hadoop.dlib.vt.edu:82/twitter/index.php. Accessed January 2016. Object Oriented Design.

[35] *Twitter IDs - Twitter Developers*. https://dev.twitter.com/overview/api/twitter-ids-json-and-snowflake. Accessed September 2016. Digital Library Research Laboratory at Virginia Tech.

[36] *Welcome to Apache Avro!* http://avro.apache.org/. Accessed January 2016. The Apache Software Foundation.

[37] *What is Scala? | The Scala programming Language*. https://www.scala-lang.org/what-is-scala.html. Accessed March 2017. Scala.