Measuring the Propagation Speed of Information Spread in Social Networks for Real-Time Fake News Detection

Alexios Polyzos, 2338
Ioannis Polyzos, 2339

Diploma Thesis

Supervisor: Prof. Spyridon Kontogiannis

Ioannina, September 2019

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF IOANNINA

Abstract

Fakes news has seen a tremendous rise in recent years and many have stated that it is a larger problem than racism, climate change, or terrorism. Studies have shown that misinformation might run the risk of making people less well informed. Rather than making them believe false news, the rise of fake news is making it harder for people to see the truth.

There has also been a great rise in the studies trying to address the problem of fake news. They are tackling the problem from different angles, for example, searching for spam/bot accounts or analyzing text from tweets along with replies and retweets on those fake news tweets.

The purpose of this work is to explore novel ways for identifying and flagging Twitter Stories as (potentially) fake as early as possible, and definitely before they get out of hand. Towards this direction, we study the behavior of Twitter Story items in the underlying graph of Twitter users, focusing mainly on their information-theoretic and structural characteristics within the graph (e.g., speed and range of propagation, structure of the involved graph, etc.) without involving the actual content of each story.

The vision of this work is to act complementarily to other, quite successful fake-news detection techniques, which are based on natural-language processing and machine-learning techniques, which nevertheless have difficulties in taking action in real-time because of either prohibitive computational requirements or very large and focused training sets.

Keywords: fake news, social networks, propagation speed, apache spark, neo4j, networkx, simrank, graph analytics

Acknowledgments

To our Mom and Dad.
Thanks for always being there.

September, 2019
Alexios Polyzos
Ioannis Polyzos

Table of Contents

Table of Figures

Table of Tables

Chapter 1. Introduction

1.1. Motivation

Fake news is usually referred to as fabricated news. It can be found in traditional news, like newspapers and on social media or fake news websites. This type of news has no basis in fact but is presented as being factually accurate.

Fake news is written and published with the intent to mislead to damage an entity, or person and/or gain financially or politically. There are, also, clickbait stories and headlines that earn advertising revenue from the activity generated.

There has been a great fuss about fake news and how it affects major events, mainly political, like the US Presidential Elections of 2016 and Great Britain's referendum regarding the EU Brexit. There have been various studies and papers regarding ways that technology can help detect and perhaps diminish the effect of fake news. Some focus on the characteristics of the accounts that share the fake news, some others on analyzing the content, and some on fact-checking.

There are seven types of fake news, as identified by **Claire Wardle of First Draft News**[1]:

Type	Description
Satire or parody	There is no malice behind or intent to harm but rather the potential to fool
False connection	When headlines, visuals and/or captions do not support the content of the article
Misleading content	The use of misleading information to frame an issue or an individual
False Context	When genuine content is shared with false contextual information
Impostor content	Impersonation of genuine sources with the use of false, made-up sources
Manipulated content	The use of "photoshopped" images to deceive
Fabricated content	This content is 100% false and is designed to deceive and do harm

Table 1 Different types of fake news

[1] https://firstdraftnews.org/fake-news-complicated/

1.2. Related Work

There are several approaches out there to tackle the problem of fake news on social media. Some focus on the characteristics of the accounts that share the fake news, some others on analyzing the content, and some others on fact-checking.

Regarding the analysis of social media accounts, the current state of the art has been focusing on trying to identify bot or spammer accounts. Benevuto et al. [5] developed a model to detect spammers by building a manually annotated dataset of spam and non-spam accounts and then extracted attributes regarding the content and user behavior.

A work by Antoniadis et al. [3] tried to identify misinformation on Twitter. The authors annotated a large dataset of tweets and developed a model using the features from the Twitter text, the users, and the social feedback it got (number of retweets, number of favorites, number of replies). On the other hand, the work of Perez-Rosas et al. [4] created a dataset of crowdsourced fake news. The dataset was built based on real news. They used crowd-sourced workers, to whom they provided a real news story and asked them to create a similar fake one.

The fact-checking approach is widely used. However, due to the massive scale of information spread through social networks, there is a need to automate this task. Automated fact-checking aims to verify claims through the extraction of data from various sources. Then, based on the strength and stance of the source regarding the claim, a classification is assigned as shown in Cohen et al. [33].

The detection of bot/spam accounts, the machine learning and deep learning approaches to the detection of fake content or even the network analysis to understand how this type of content can be identified is diffuse and generally yet quite difficult to be understood by the general public.

There have been various researches using machine learning algorithms, like SVM, K-NN and Naïve Bayes that could tackle the Fake News Detection problem. Elmurngi and Gherbi [6]used machine learning algorithms with really promising results, with SVM having the greatest accuracy both in text classification and the detection of fake reviews.

Also, the characterization of content and sentiment analysis have become very interesting methods for fake news detection. Oshikawa et al. [7] used Natural Language Processing (NLP) techniques for automated fake news detection and reviewed several existing NLP techniques.

The missing step that our approach tries to cover: Early detection of potential Fake News, to be consequently analyzed and cross-checked, by fact-checkers or computationally demanding NLP and ML techniques.

1.3. Structure

Comparing the spread of fake news inside a social network as the spread of a virus inside an organism, we can see that many of the aforementioned approaches detect a fake news story after it has spread inside a network and caused damage. With our approach, we are trying to flag Twitter Stories as potentially fake news before they have spread, and isolate them for further analysis.

In our approach, we will not focus on the type (whether it's text, image, etc.) or the content of fake news items, but rather their propagation speed within a social network. Our goal is to verify whether, inside a social network, a fake news story is spread differently (e.g., faster, or exploiting different graph structures) than a legitimate counterpart. This could happen due to various reasons, one of them being the use of strong language and/or dramatic "doctored" imagery, due to the use of bots/spam accounts that help bombard a social network, etc.

As an example, our experimental evaluation of our approach both on legitimate and fake-news stories, on a dataset that we describe in the following chapters, revealed that if we take the vectors returned by the SimRank algorithm on the subgraph that was involved in the propagation of a specific story, then these "similarity" vectors fit to different power-law distributions for legitimate and fake stories.

In Chapter 2 we will explain the tools we use and how they help us build our approach. In Chapter 3 we present in detail our approach for fake news early detection, i.e., we explain what it takes to build a good dataset and how to process it "on-the-fly" with Graph analytics tools, to provide a novel technique for fake news detection. In Chapter 4 we present all the experiments we have conducted on the dataset that we have built. And finally, in Chapter 5 we present some directions for further improving our novel approach.

Chapter 2. Setting up the stage

2.1. State of the Art

There is a lot of ongoing research around the field of Fake News detection. The majority of that research has been targeting fake reviews, biased messages and against-fact information (false news and hoaxes). State of the art concerning types of fake news and the solutions that are being proposed around content analysis, network propagation, fact-checking and fake news analysis, and emerging detection systems, with a heavy focus around the corpus and structure of the data - with classification methods being the primary approach.

Researchers focus on analyzing patterns of propagation, creating supervised systems to classify unreliable content, characteristics of the accounts that share a similar type of content or fact-checking claims. What is common in most approaches is that researchers tend to gather a batch dataset from social media like twitter and try to characterize the validity of the corpus. This approach typically involves researchers with different backgrounds like psychology and linguistics, to better evaluate the content. After labels have been attached to the collected dataset, classification methods are used to classify the label of each post. Methods include Decision Trees like C4.5 [34], Entropy Minimization Discretization [35], Neural Networks like Multilayer Perceptron, Deep Nets and LSTM [36, 37, 38], Bayesian methods like Naive Bayes [39], Instance-based like k-NN [40], Kernel-based like SVM [41] or Rule-based like RIPPER [42].

Earlier approaches include methods like stance detection. Stance detection [44] is the extraction of a subject's reaction to a claim made by a primary user. It can be applied to understand if news, written from a resource with an unknown reputation, is agreeing or disagreeing with the majority of the media outlets. A conceptually similar task to stance detection is textual entailment and methods around the subject include bidirectional LSTM [45], Gradient Boosted classifiers [46]. Feature extraction can play an important role here and the main approaches for that are typically Bag of Words and GloVe vectors.

Our approach focuses on the structural and network-related properties (e.g., propagation speed, the structure of the involved subgraph, etc) of a post in a social network. More specifically instead of using a batch of the data, we focus on real-time feeds using a graph-based model. We try to discover hidden relationships inside the network and determine whether a node in that graph can be a bot spreading fake information, based on the speed and relationships. It should be mentioned though that our main goal is not to characterize whether stories are potentially fake or legitimate, rather than users being bots and/or fake-news producers.

2.2. Graph Data Modeling vs Relational Data Modeling

Relational databases have been the workhorse of software applications since the '80s, and continue as such to this day. They store highly-structured data in tables with predetermined columns of specific types and many rows of those defined types of information. Due to the rigidity of their organization, relational databases require developers and applications to strictly structure the data used in their applications.

In relational databases, references to other rows and tables are indicated by referring to primary key attributes via foreign key columns. Joins are computed at query time by matching primary and foreign keys of all rows in the connected tables. These operations are compute-heavy and memory-intensive and typically have prohibitive costs.

When many-to-many relationships occur in the model, you must introduce a JOIN table (or associative entity table) that holds foreign keys of both the participating tables, further increasing join operation costs. Figure 1 shows this concept of connecting a Person (from Person table) to a Department (in Department table) by creating a Person-Department join table that contains the ID of the person in one column and the ID of the associated department in the next column.

This makes understanding the connections very cumbersome because one should know the person ID and department ID values (performing additional lookups to find them) to know which person connects to which departments. Those types of costly join operations are often addressed by denormalizing the data to reduce the number of joins necessary, therefore breaking the data integrity of a relational database.

Unlike other database management systems, relationships are of equal importance in the graph data model to the data itself. This means we are not required to infer connections between entities using special properties such as foreign keys or out-of-band processing like map-reduce.

By assembling nodes and relationships into connected structures, graph databases enable us to build simple and sophisticated models that map closely to our problem domain. The data stays remarkably similar to its form in the real world – small, normalized, yet richly connected entities. This allows us to query and view the data from any imaginable point of interest, supporting many different use cases.

Each node (entity or attribute) in the graph database model directly and physically contains a list of relationship records that represent the relationships to other nodes. These relationship records are organized by type and direction and may hold additional attributes. Whenever you run the equivalent of a JOIN operation, the graph database uses this list, directly accessing the connected nodes and eliminating the need for expensive search-and-match computations.

This ability to pre-materialize relationships into the database structure allows Graph Databases to demonstrate performances several orders of magnitude faster than others, especially for joining heavy queries, allowing users to leverage a minute to milliseconds advantage.

2.2.1. Structural Difference between Relational and Graph

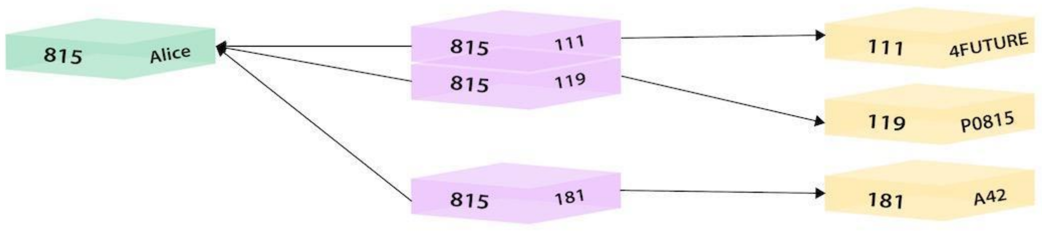


Figure 1 Relational – Person and Department tables. Source: https://neo4j.com

In the relational example of Figure 1, we search the Person table on the left (potentially millions of rows) to find the user Alice and her person ID of 815. Then, we search the Person-Department table (orange middle table) to locate all the rows that reference Alice's person ID (815). Once we retrieve the 3 relevant rows, we go to the Department table on the right to search for the actual values of the department IDs (111, 119, 181). Eventually, we know that Alice is part of the department named.

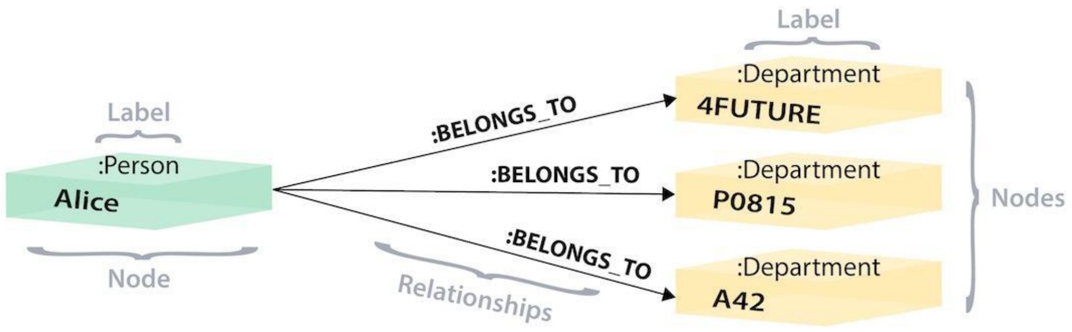


Figure 2 Graph – Alice and 3 Departments as nodes. Source: https://neo4j.com

In the graph version presented in Figure 2, we have a single node for Alice with a label of Person. Alice belongs to 3 different departments, so we create a node for each one and with a label of Department. To find out which departments Alice belongs to, we would search the graph for Alice's node, then traverse all of the BELONGS_TO relationships from Alice to find the Department nodes she is connected to. That's all we need – a single hop with no lookups involved.

2.3. Building Systems that scale in the era of Big Data

2.3.1. The JVM world - Scala & Kotlin

Throughout this Thesis implementation, Scala and Kotlin programming languages are being used, except for the graph analytics part where Python takes all the heavy lifting, as advocates of – the right tool for the job - movement. Both Scala and Kotlin are object oriented-functional languages that are statically typed, running on the Java Virtual Machine (JVM), and offer excellent interoperability with Java that many of the applications and legacy systems already use.

They both have a powerful type system. It can be used to effectively encode some of our domain logic. If used effectively, the type system can help drastically reduce boilerplates and unnecessary test code.

They have first-class language support for functional programming. They support the functional paradigm with higher-order functions and a rich set of combinators, implying that they can collaborate within MapReduce Big Data Model. Using functions as first-class abstractions, you can implement domain behaviors, which can be referentially transparent and thus compositional. Immutability is also a strong aspect of these languages.

Spark is written in Scala and, since Scala runs on the JVM, it is the most prominent language for Big Data Development.

Finally, since they are both statically typed languages, they both support wonderful concurrency based on primitives, they are efficient and fast making them an ideal choice for computationally intensive algorithms and have well-designed libraries for Scientific computing they are the most logical solution

2.3.2. The Akka Toolkit

The Akka toolkit is a JVM based Actor-Model implementation, for building highly concurrent, distributed and resilient, message-driven applications. It provides a rich API for both Java and Scala, but we chose Scala because it's a hybrid language that brings together the best of both worlds - Object-Oriented and Functional Programming - and it is heavily used for building reactive systems, as well as for its elegant approach in error handling.

The actor model is a mathematical model of concurrent computation that treats "actors" as the universal primitives of concurrent computation. In response to a message that is received, an actor can:

- Make local decisions
- Create more actors
- Send more messages
- Determine how to respond to the text message received
- Monitor and supervise the actors he has under his "watch"

Actors may modify their private state, but can only affect each other through messages, avoiding the need for any locks. More information about the actor model can be found at [8].

2.3.3. Apache Spark

Apache Spark is a general-purpose distributed data processing engine that is suitable for use in a wide range of circumstances. On top of the Spark core data processing engine, there are libraries for SQL, machine learning, graph computation and stream processing, which can be used together in an application. Programming languages supported by Spark include: Scala, Java, Python, and R. Application developers and data scientists incorporate Spark into their applications to rapidly query, analyze and transform data at scale. Tasks most frequently associated with Spark include ETL and SQL batch jobs across large data sets, processing of streaming data from sensors, IoT or financial systems and machine learning tasks.

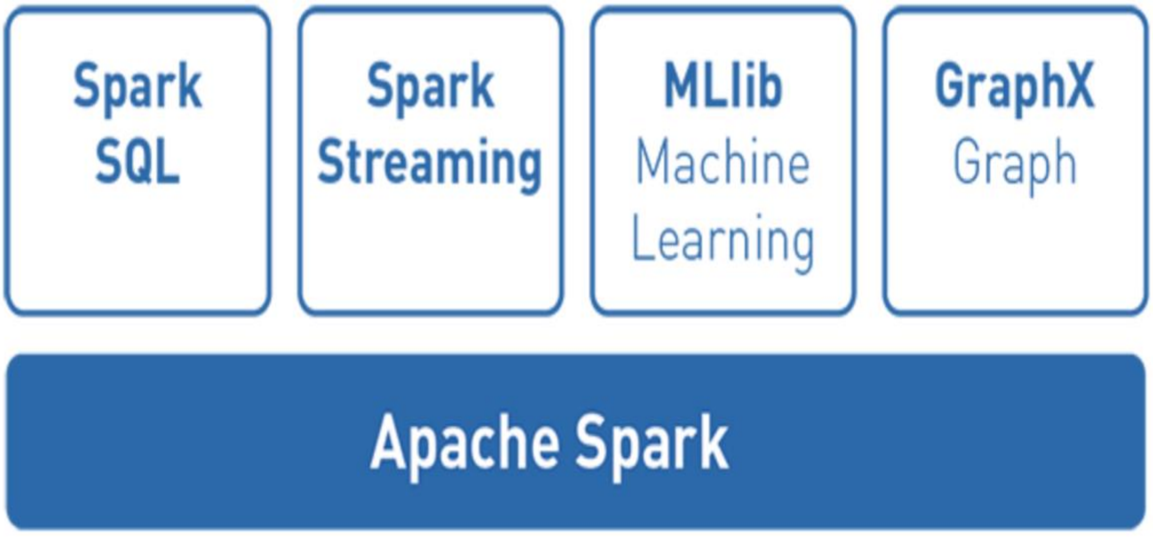


Figure 3 Apache Spark APIs. Source: https://spark.apache.org

Of course, we can't mention Apache Spark and large-scale data processing without bringing MapReduce to the picture. To understand Spark, it helps to understand its history. Before Spark, there was MapReduce, a resilient distributed processing framework, which enabled Google to index the exploding volume of content on the web, across large clusters of commodity servers.

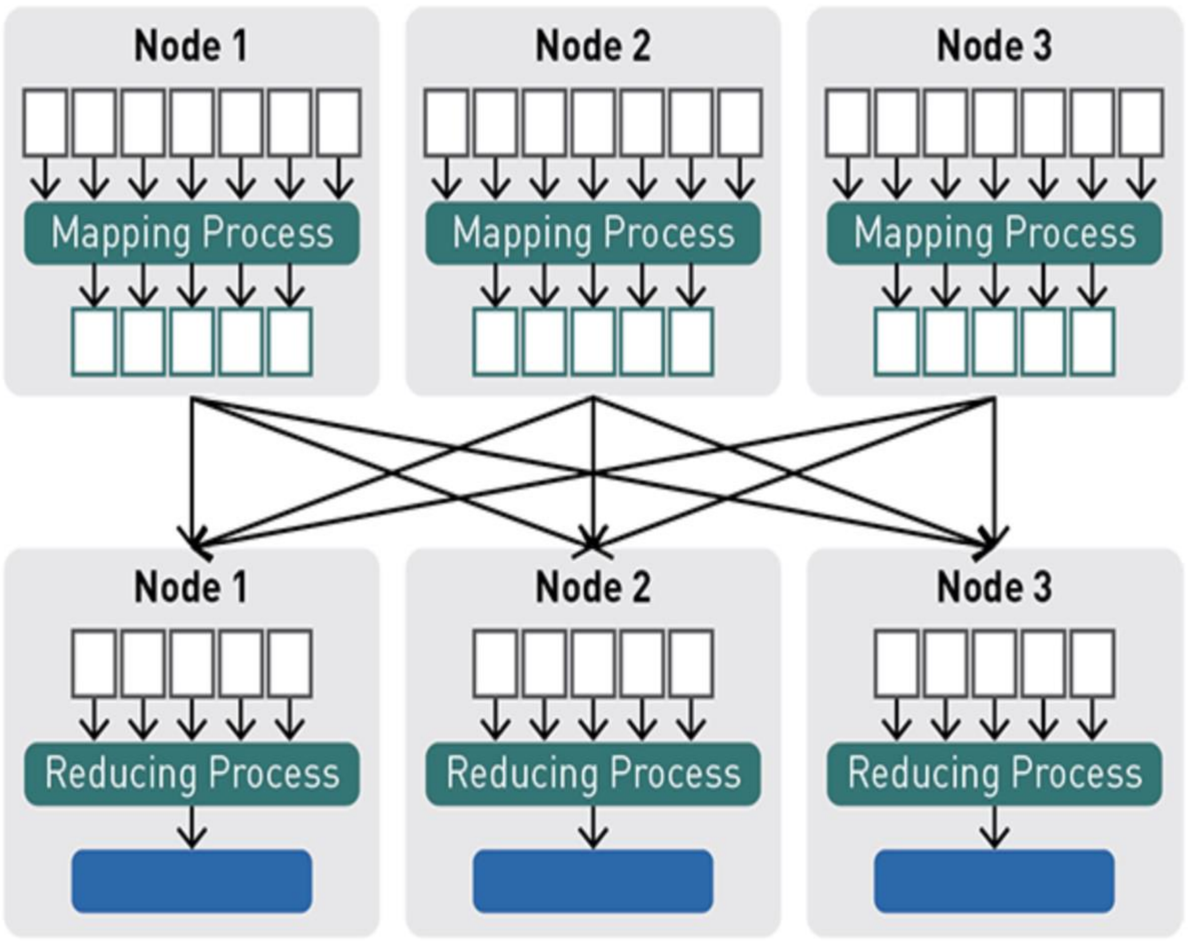


Figure 4 The internals of a Map-Reduce Job. Source: https://mapr.com

There were 3 core concepts to the Google strategy:

1. **Distribute data:** when a data file is uploaded into the cluster, it is split into chunks, called data blocks, and distributed amongst the data nodes and replicated across the cluster.
2. **Distribute computation:** users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs and a reduce function that merges all intermediate values associated with the same intermediate key. Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines in the following way:
    - The mapping process runs on each assigned data node, working only on its block of data from a distributed file.
    - The results from the mapping processes are sent to the reducers in a process called "shuffle and sort": key/value pairs from the mappers are sorted by key, partitioned

by the number of reducers, and then sent across the network and written to key sorted "sequence files" on the reducer nodes.

  - The reducer process executes on its assigned node and works only on its subset of the data (its sequence file). The output from the reducer process is written to an output file.

3. **Tolerate faults:** both data and computation can tolerate failures by failing over to another node for data or processing.

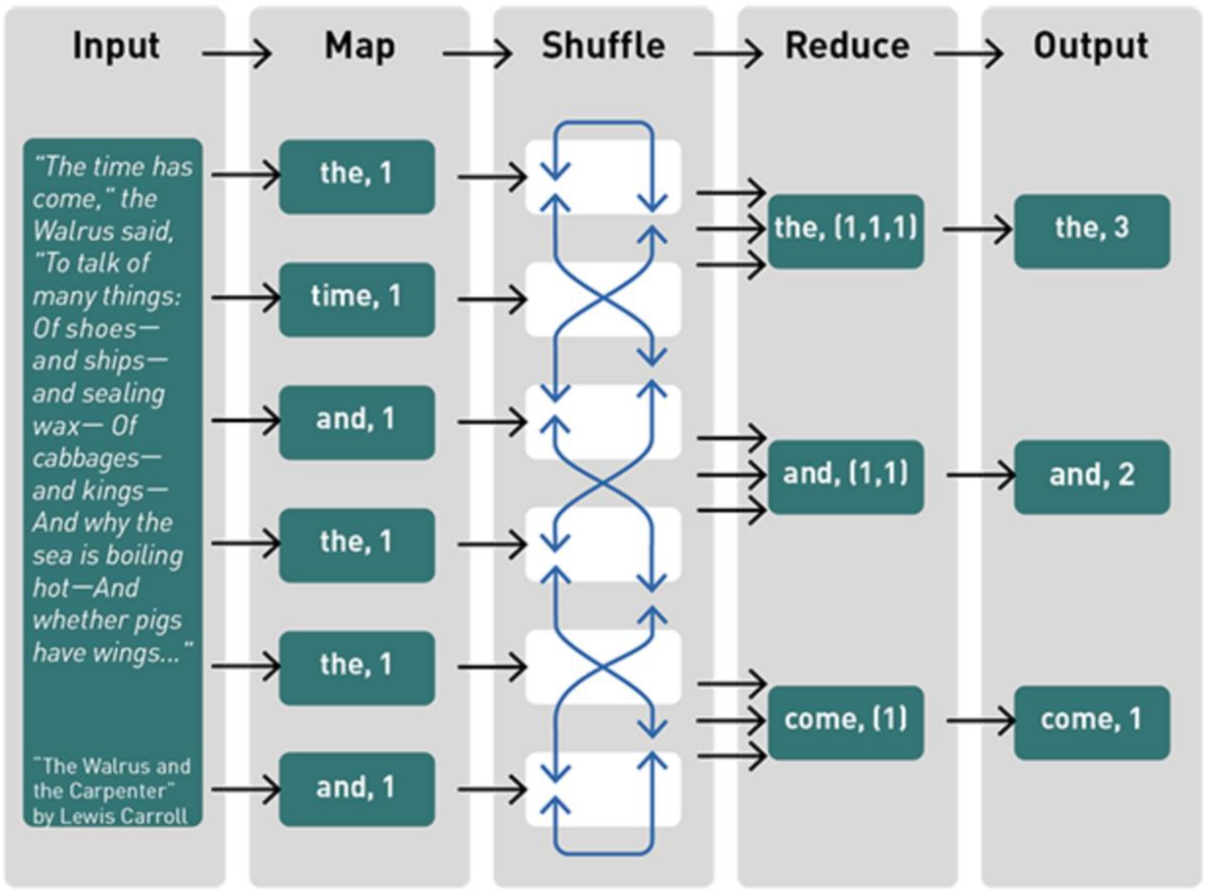

Input
Map
Shuffle
Reduce
Output
"The time has come," the Walrus said, "To talk of many things: Of shoes— and ships— and sealing wax— Of cabbages— and kings— And why the sea is boiling hot—And whether pigs have wings..."
"The Walrus and the Carpenter" by Lewis Carroll
the, 1
time, 1
and, 1
the, 1
the, 1
and, 1
the, (1,1,1)
and, (1,1)
come, (1)
the, 3
and, 2
come, 1

Figure 5 MapReduce word count execution example. Source: https://mapr.com

Some iterative algorithms like PageRank, which Google used to rank websites in their search engine results, require chaining multiple MapReduce jobs together, which causes a lot of reading and writing to disk. When multiple MapReduce jobs are chained together, for each MapReduce job, data is read from a distributed file block into a mapping process, written to and read from a SequenceFile in between, and then written to an output file from a reducer process.

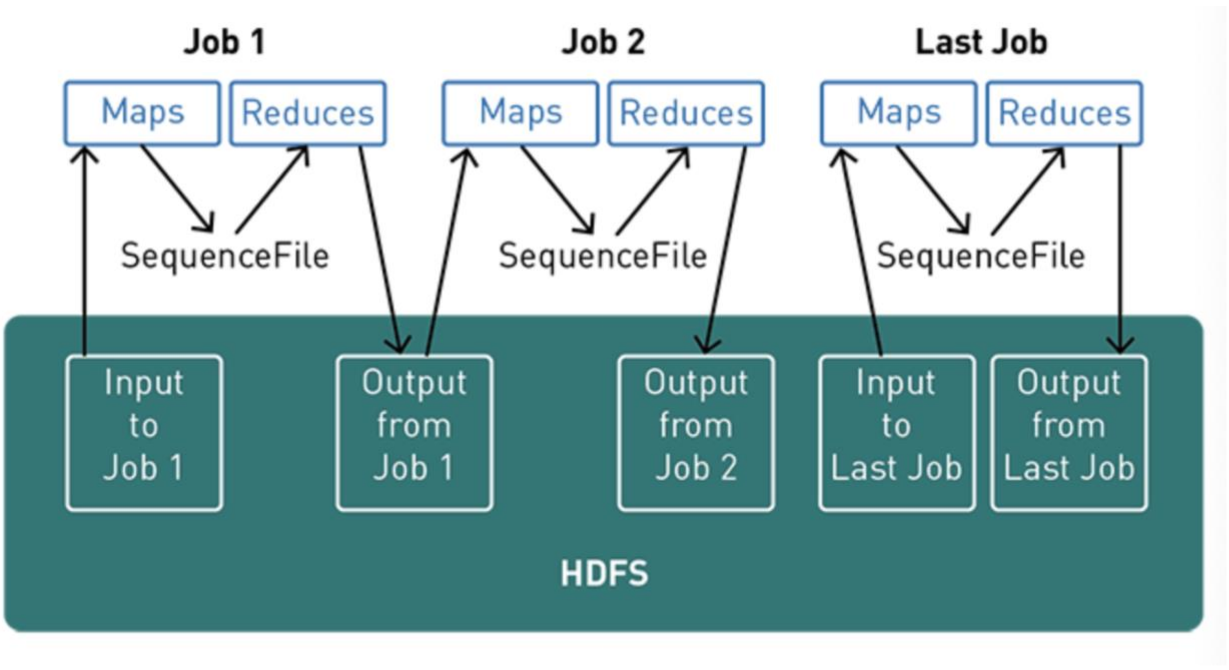


Figure 6 Mappers and reducers in action. Source: https://mapr.com

Apache Spark is one of the best tools available for large scale distributed data processing. It is built on Google's MapReduce, but with some advantages:

- Executes must faster, by caching data in memory across multiple parallel operations, whereas MapReduce involves more reading to and writing from disk.
- Runs multi-threaded tasks inside of JVM processes, whereas MapReduce runs as heavier weight JVM processes. This gives Spark faster startup, better parallelism, and better CPU utilization.
- It provides a richer functional programming model than MapReduce.
- It is especially useful for parallel processing of distributed data with iterative algorithms.

The diagram in Figure 7 shows an Apache Spark application running on a cluster.

- A Spark application runs as independent processes, coordinated by the Spark Session object in the driver program.
- The resource or cluster manager assigns tasks to workers, one task per partition.
- A task applies its unit of work to the dataset in its partition and outputs a new partition dataset. Because iterative algorithms apply operations repeatedly to data, they benefit from caching datasets across iterations.
- Results are sent back to the driver application or can be saved to disk.

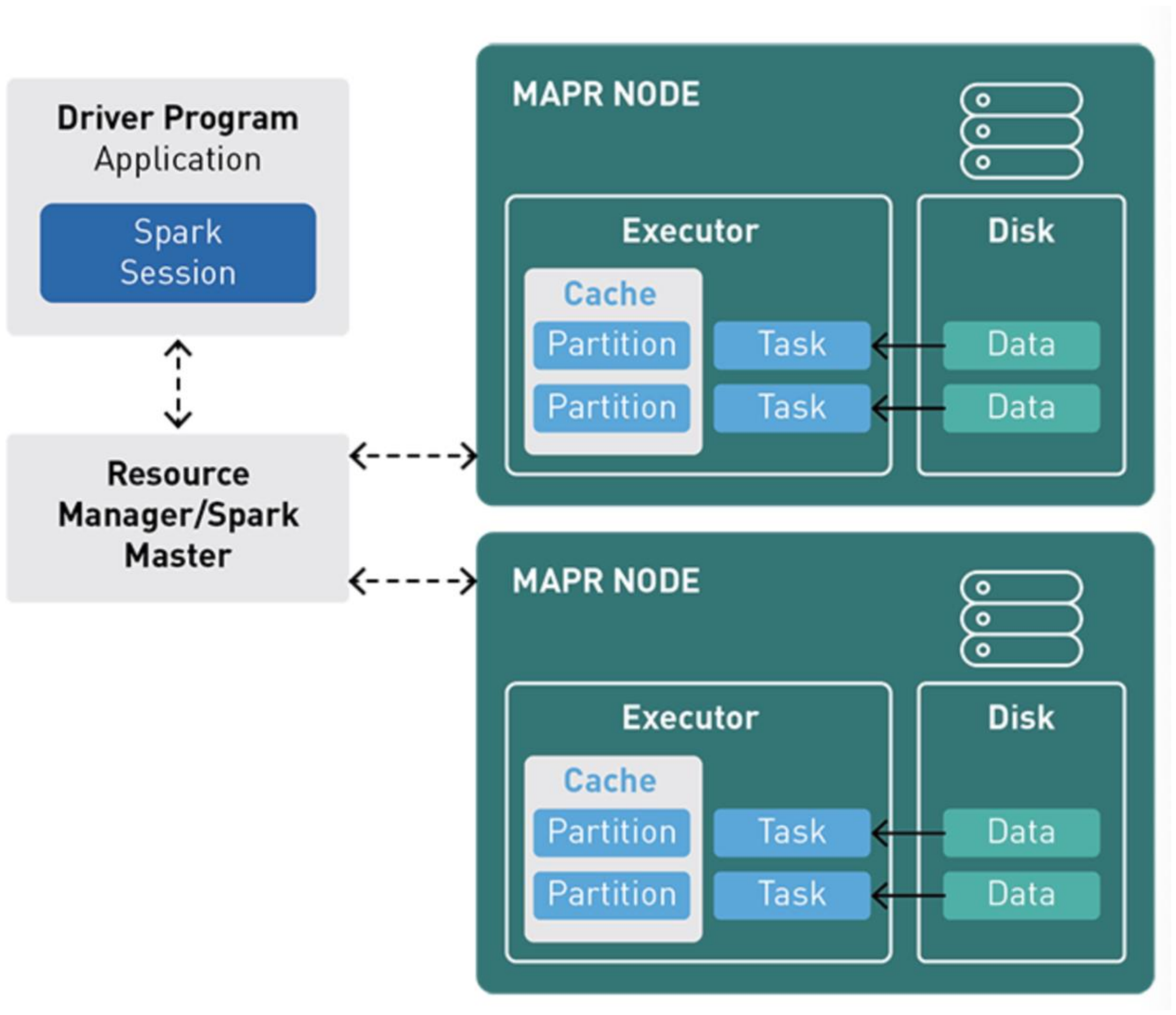


Figure 7 Spark Application running on a cluster. Source: https://mapr.com

Apache Spark is capable of handling several petabytes of data at a time, distributed across a cluster of thousands of cooperating physical or virtual servers. It has an extensive set of developer libraries and APIs along with its support for multiple languages. Thus, its flexibility makes it well-suited for a range of use cases. Spark is often used with distributed data stores such as MapR XD, Hadoop's HDFS, and Amazon's S3, with popular NoSQL databases such as MapR-DB, Apache HBase, Apache Cassandra, and MongoDB, and with distributed messaging stores such as MapR-ES and Apache Kafka.

Typical use cases include:

Stream processing: From log files to sensor data, application developers are increasingly having to cope with "streams" of data. This data arrives in a steady stream, often from multiple sources simultaneously. While it is certainly feasible to store these data streams on disk and analyze them retrospectively, it can sometimes be sensible or important to process and act upon arrivals of new data items. Streams of data related to financial transactions, for example, can be processed in real-time to identify– and refuse– potentially fraudulent transactions.

Machine learning: As data volumes grow, machine learning approaches become feasible and increasingly more accurate. The software can be trained to identify and act upon triggers within well-understood data sets before applying the same solutions to new and unknown data. Spark's ability to store data in memory and run rapidly repeated queries, makes it a good choice for training machine learning algorithms. Running broadly similar queries again and again, at scale, significantly reduces the time required to go through a set of possible algorithms to find the most efficient solutions.

Interactive analytics: Rather than running pre-defined queries to create static dashboards of sales, production line productivity, or stock prices, business analysts and data scientists want to explore their data by asking a question, viewing the result, and then either altering the initial question slightly or drilling deeper into results. This interactive query process requires systems such as Spark that can respond and adapt quickly.

Data integration: Data produced by different systems across a business is rarely clean or consistent enough to be combined simply and easily for reporting or analysis. Extract, transform, and load (ETL) processes are often used to pull data from different systems, clean and standardize it, and then load it into a separate system for analysis. Spark (and Hadoop) are increasingly being used to reduce the cost and time required for this ETL process.

Much of Spark's power lies in its ability to combine very different techniques and processes into a single, coherent whole. Outside Apache Spark, the discrete tasks of selecting data, transforming that data in various ways, and analyzing the transformed results might easily require a series of separate processing frameworks, such as Apache Oozie. Apache Spark, on the other hand, offers the ability to combine these, crossing boundaries between batch, streaming, and interactive workflows in ways that make the user more productive.

Apache Spark jobs perform multiple operations consecutively, in memory, and only spilling to disk when required by memory limitations. Apache Spark simplifies the management of these disparate processes, offering an integrated whole – a data pipeline that is easier to configure, easier to run, and easier to maintain. In use cases such as ETL, these pipelines can become extremely rich and complex, combining large numbers of inputs and a wide range of processing steps into a unified whole that consistently delivers the desired result.

2.3.4. Neo4j

The Neo4j Graph Platform is a tightly integrated graph database and algorithm-centric processing, optimized for graphs. It is popular for building graph-based applications and includes a graph algorithms library tuned for the native graph database.

Neo4j is the right platform when:

- Algorithms are more iterative and require good memory locality.
- Algorithms and results are performance sensitive.
- Graph analysis is on complex graph data and/or requires deep path traversal.
- Analysis/Results are tightly integrated with transactional workloads.
- Results are used to enrich an existing graph.
- Need for integration with graph-based visualization tools.
- Need for prepackaged and supported algorithms.

It also provides transactional processing and analytical processing of graph data. It includes graph storage and computes with data management and analytics tooling. The set of integrated tools sits on top of a common protocol API, and query language (Cypher) to provide effective access for different uses. Neo4j includes the Neo4j Graph Algorithms library, which can be installed as a plugin alongside the database, and provides a set of user-defined procedures that can be executed via the Cypher query language.

The graph algorithm library includes parallel versions of algorithms supporting graph-analytics and machine-learning workflows. The algorithms are executed on top of a task-based parallel computation framework and are optimized for the Neo4j platform. For different graph sizes, there are internal implementations that scale up to tens of billions of nodes and relationships.

Results can be streamed to the client as a tuples stream and tabular results can be used as a driving table for further processing. Results can also be optionally written back to the database efficiently as node-properties or relationship types. It also includes the Neo4j APOC (Awesome Procedures on Cypher) library, which consists of more than 450 procedures and functions to help the common tasks such as data integration, data conversion, and model refactoring.

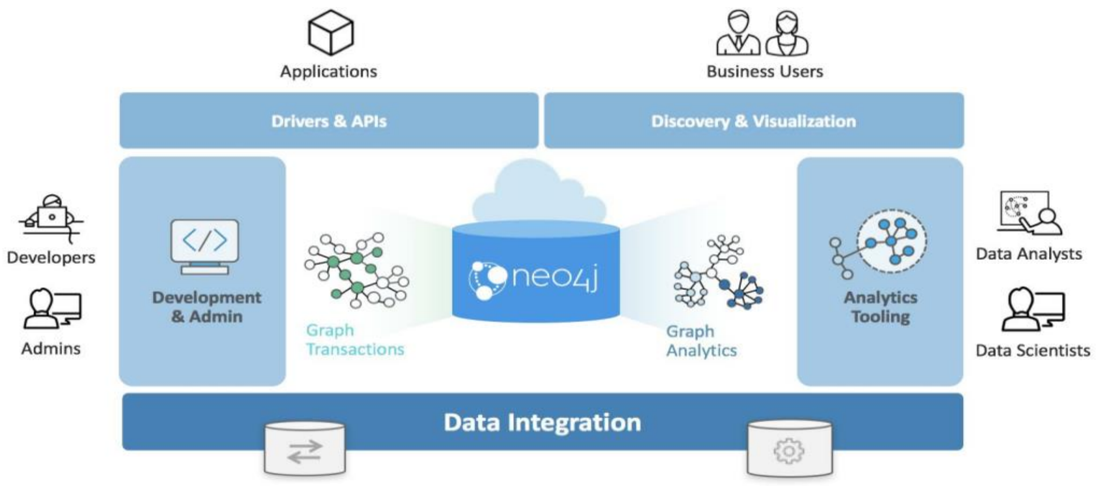


Figure 8 The Neo4j Graph Platform is built around a native database that supports transactional applications and graph analytics. Source: https://neo4j.com

Cypher is Neo4j's graph query language that allows users to store and retrieve data from the graph database. Neo4j wanted to make querying graph data easy to learn, understand, and use for everyone, but also incorporate the power and functionality of other standard data access languages. This is what Cypher aims to accomplish.

Cypher is a vendor-neutral open graph query language employed across the graph ecosystem. Cypher's ASCII-art style syntax provides a familiar, readable way to match patterns of nodes and relationships within graph datasets.

Like SQL, Cypher is a declarative query language that allows users to state what actions they want to be performed (such as match, insert, update or delete) upon their graph data without requiring them to describe (or program) exactly how to do it.

Two years ago, Neo4j, Inc. decided to open source the Cypher language and make the most popular graph query language available to any technology provider with the aim that Cypher becomes the "SQL for graphs." The openCypher project provides an open language specification, technical compatibility kit, and reference implementation of the parser, planner, and runtime for Cypher. It is backed by several companies in the database industry and allows implementers of databases and clients to freely benefit from, use, and contribute to the development of the openCypher language.

2.3.5. Google Cloud Platform

As we mentioned before, we need a lot of computation prowess and lots of fast and reliable storage to research and test our assumptions. One of the options was to use the clusters that the Faculty has available for research purposes. However, it came to us that our Pipeline would need to run continuously on that cluster and it would take us a great amount of network throughput and storage IOPS (refers to Input/output Operations Per Second) and on top of that fellow students would need access to the cluster. So, we chose a Public Cloud to run our experiments so that we did not have to take into consideration all the above-mentioned restrictions.

Google Cloud Platform was our choice for the public cloud since it's one of the top competitors in the cloud market and on top of that, we had enough resources to last us throughout the whole research and testing period. It comes with a lot of offerings and services that helped us focus on the experiments rather than focusing on how we would get the infrastructure to work the way we needed it to work.

Chapter 3. Data Pipelines & Data Insights

3.1. Stage 1: Building the Dataset

3.1.1. Available Datasets and Limitations

Investigating a problem like Fake News Detection is a hard topic. It becomes even harder when the data you need is not available so that we can make sure that our results are representative. Long-running research is being conducted around the topic, but all the institutions that get involved in this topic, struggle a lot to collect this kind of data and they keep it private since there is a lot of work involved around building a representative dataset.

Search for a dataset that would be representative and meet the needs of this research, was a very long ride. Searching the web, all the data available would be either really small in size or wouldn't contain any indication that it could contain spreading fake news posts. There are a lot of social networks out there from where data could be fetched, but since the concern was about the structural and information-theoretic properties (e.g., speed of propagation, structure of the transmission subgraph, etc), the need arises for a source where data could be fetched in real-time and having a robust and well-maintained API.

That's why the Twitter API was used. Twitter offers a robust and well-maintained API, from which data can be fetched in real-time from the live feed, fetch historic data or monitor based on certain hashtags, handles or users. But everything comes at a cost and the Twitter API is no different. It has its limitations since data is the most valuable asset in our times, so there are quite some limitations where someone would hit a wall - more on this later.

3.1.2. Finding the data

After this initial search and choosing twitter as the source of truth for their live-stream API, there was still one major issue - how would a distinction be made between users that could probably propagate fake information on the social network. After further investigation, there is this amazing work of Shao et al [40], who made extensive research around different accounts in the twitter social networks and their results give a wide list of twitter accounts that could be marked as sites that generate fake, satire news and more.

Building on their results, a good approach would be to filter some accounts from their list, that are known to be accounts of fake news propagation, monitor these sites to fetch the data they generate and start building a system that would be able to run for weeks or even months and would be able to build a dataset to meet our needs. For that reason, the system implemented for this task had to be asynchronous and non-blocking to process large amounts of incoming data which would be stored on the disk in high throughput. It also needed to be robust and fault-tolerant, so that no data would be lost and in case of failure, and there would be no need to start the whole process from the beginning.

For that reason, the implementation of this system was based on the Scala programming language and the Akka Toolkit as discussed in Sections 2.3.1 and 2.3.2 respectively. Having these tools at hand, a fault-tolerant system that monitors the twitter feed and fetches the data was created to cover the missing dataset need. More specifically an actor hierarchy was created that consists of the following actors, each having a specific role in the system.

- The **StreamListenerActor** holds a selection list of twitter accounts that are marked as fake news spreading sites from [3]. The role of this actor is to monitor these twitter accounts and catch every post that is being generated, whether it is a tweet post, a retweet or a reply. When a retweet or reply gets fetched, it checks whether or not the last 200 tweet posts retrieved for that retweet/reply could trace back to its original post. If it does, then it stores the entire story chain until that point to the disk and keeps monitoring for more incoming posts.

```
private def monitorStream(ids: Seq[Long]): Future[TwitterStream] = {
    streamingClient.filterStatuses(
      follow = ids,
      stall_warnings = true,
      languages = Seq(Language.English))(cacheTweet)
  }

  private def cacheTweet: PartialFunction[StreamingMessage, Unit] = {
    case tweet: Tweet =>
      if (tweet.retweeted_status.isDefined && !postsCache.contains(tweet.retweeted_stat
us.get.id)) {
        val id = tweet.retweeted_status.get.id
        postsCache += id
        retweetHandlerActor ! FetchRetweets(id)
        if (postsCache.size > 1000) postsCache.clear()
      }

      if (streamCache.size > 200) {
        saveToDisk(streamCache.toList, "fake_tweets.json")(context.system)
        streamCache.clear()
      }
      streamCache += tweet
    case _ =>
      log.info("Unknown object received from twitter stream.")
  }
```

Figure 9 Sample code of the StreamListenerActor that monitors Twitter Stream API for actions from specific Twitter Accounts.

- The **RetweetHandlerActor** actor collaborates with the **StreamListenerActor.** When the latter receives retweets or replies it communicates with the former through a message and the **RetweetHandlerActor** fetches the 200 most recent posts for that tweet. Then it checks if we can trace back to their original post and if yes it stores the story on the disk.

```
private def retrieveRetweets(id: Long): Unit = {
   implicit val ec: ExecutionContextExecutor = ExecutionContext.global
   client.retweets(id = id) onComplete {
     case Success(result) =>
       log.info(s"Fetched '${result.data.size}' retweets for tweet $id.")
       val retweets = result.data.toList
       saveToDisk(retweets, "retweets_batch.json")(context.system)
       log.info(s"Rate Limit Remaining: ${result.rate_limit.remaining}")
       if (result.rate_limit.remaining == 0) Thread.sleep(60*15 + 5)

     case Failure(exception) => log.error("Failed to retrieve retweets for '$id': ", exception.printStackTrace())
   }
 }
```

Figure 10 Sample code of the RetweetHandlerActor that caters for the retrieval of retweets.

- The **SampleStreamListenerActor** is responsible for monitoring the twitter live sampling stream and consumes all the generated streaming data, storing them after regular time intervals on the disk. It also makes sure to clear its caching data after it gets stored, so that we don't run out of memory since the amount of data is quite large.

```
private def sampleTweetHandler: PartialFunction[StreamingMessage, Unit] = {
    case tweet: Tweet =>
      if (tweet.user.get.followers_count > followersCountThreshold) {
        log.info(s"Received tweet: ${tweet.text.toString}")
        if (sampleStreamCache.size > 100) {
          log.info(s"Total tweets in cache '${sampleStreamCache.size}' - saving data to disk.")
          saveToDisk(sampleStreamCache.toList, "sample_tweets_stream.json")(context.system)
          sampleStreamCache.clear()
        }
        sampleStreamCache += tweet
      }
    case _ =>
      log.info("Unknown object received from twitter stream.")
  }
```

Figure 11 Sample code of the SampleStreamListenerActor that monitors the Twitter Stream API.

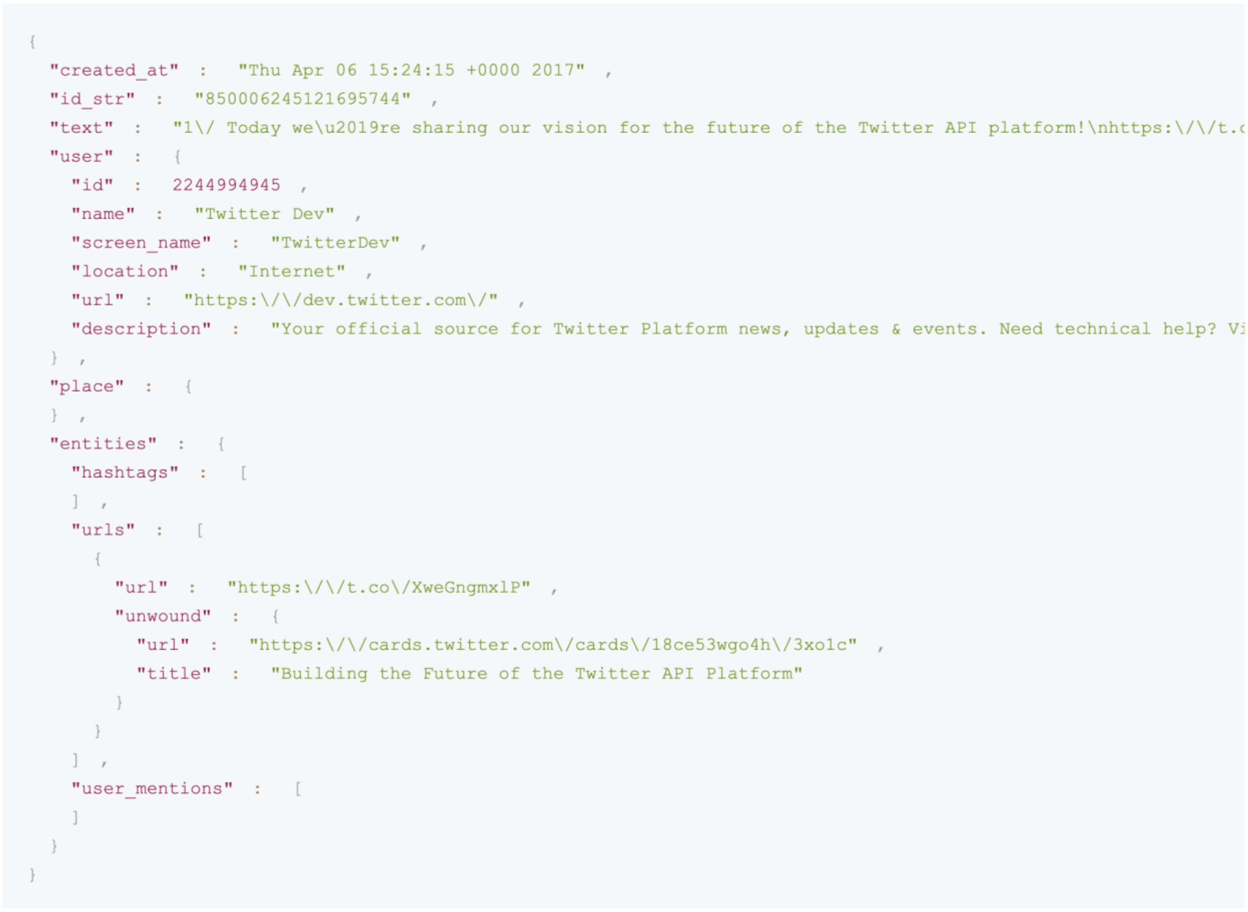
Running the application on Google Cloud servers for almost a week, ingests about 35GB of data, from the Twitter Stream API as well as from the sources that are marked as Fake News Generators. So the combination of twitter's live stream and suspicious twitter accounts creates an output result that contains both legitimate stories (from the live stream) as well as fake stories (from these suspicious accounts).

3.1.3. Data lakes

As mentioned in the previous section, all these data being collected from the application was stored on the file system. Typically, when we talk about Data Pipelines, the part of the pipeline where the ingested data gets dumped it's called a Data Lake.
To quote from Amazon Web Service's page - A data lake is a centralized repository that allows you to store all your structured and unstructured data at any scale. You can store your data as-is, without having to first structure the data, and run different types of analytics—from dashboards and visualizations to big data processing, real-time analytics, and machine learning to guide better decisions.

In most cases, it is a distributed file system, like Hadoop's HDFS or Amazon's S3 - systems that are distributed and fault-tolerant that can provide guarantees that your large volumes data can be safely stored, without the risk of losing them. A discussion about Hadoop's HDFS or Amazon's S3 is out of the scope in this thesis. For this use case even though the generated data was not that big, Google's Cloud Storage was used since the infrastructure was based on Google Cloud.

The output of the crawler stores three new files in the data lake - one containing the tweets from the monitored fake news propagator sites, one file containing all the history of that tweet that was probably missed and one containing the data from twitter's live stream feed. All data was stored for further processing and each file is a JSON file that contains tweet posts. Figure 12 shows the structure of a typical tweet object in JSON format.



```
{
  "created_at"  :  "Thu Apr 06 15:24:15 +0000 2017" ,
  "id_str"  :  "850006245121695744" ,
  "text"  :  "1\/ Today we\u2019re sharing our vision for the future of the Twitter API platform!\nhttps:\/\/t.
  "user"  :  {
    "id"  :  2244994945 ,
    "name"  :  "Twitter Dev" ,
    "screen_name"  :  "TwitterDev" ,
    "location"  :  "Internet" ,
    "url"  :  "https:\/\/dev.twitter.com\/" ,
    "description"  :  "Your official source for Twitter Platform news, updates & events. Need technical help? Vi
  } ,
  "place"  :  {
  } ,
  "entities"  :  {
    "hashtags"  :  [
    ] ,
    "urls"  :  [
      {
        "url"  :  "https:\/\/t.co\/XweGngmxlP" ,
        "unwound"  :  {
          "url"  :  "https:\/\/cards.twitter.com\/cards\/18ce53wgo4h\/3xo1c" ,
          "title"  :  "Building the Future of the Twitter API Platform"
        }
      }
    ] ,
    "user_mentions"  :  [
    ]
  }
}
```

Figure 12 The Tweet Object JSON Format as seen on the Twitter API Documentation

It is a JSON representation, containing many nested fields, for which more information can be found about its structure and data types at the Twitter Developer Portal https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/tweet-object.html.

After having dumped all this unstructured data in the data lake, the next step is to further process it, to find out more information about its quality and probably more hidden insights.

3.2. Stage 2: Data Preprocessing

3.2.1 ETL with Spark SQL

The next layer in the pipeline is the ETL preprocessing layer. All the data living in the data lake needs to be preprocessed, to bring it in a clean and structured format so it can later be processed further to start extracting some meaningful insights. Apache Spark, mentioned in Section 2.3.3, is a lightning-fast distributed data processing framework and therefore is a perfect fit for our preprocessing phase since we need to combine large amounts of multiple data sources and execute a series of transformations on them.

The data is already inside the system, but why is it necessary to run transformations on and not use them as they are? Information coming in from different sources contains metadata that is not trivial to these experiments. Data might also be corrupted or malformed, it might have outliers that cause noise, as well as for this case there are relationships between the fetched tweets, retweets, replies than need to be unraveled. For example, there is a need to explore and find the whole history of a tweet story (because its important to look for a tweet post and its sequence of retweets to investigate its propagation) and on the other hand discard any entries for which the original tweet post might not exist (after all the streaming API is monitored and all the incoming data is consumed - whether it is relevant or not).

For this need a Spark cluster was set up on the Google Cloud Platform, to handle this volume of data. Since data contains a schema Spark SQL can be leveraged and perform the processing using plain SQL queries. The ETL job can be broken down to the following series of steps:

- **Step 1** - The first thing in any Spark application is to create a SparkSession to ingest the data from our data lake and start the processing job. The SparkSession provides a single point of entry to interact with underlying Spark functionality and allows programming Spark with DataFrame and Dataset APIs. Most importantly, it curbs the number of concepts and constructs a developer has to juggle while interacting with Spark.

```
val spark = SparkSession
                .builder()
                .appName("tweets-etl")
                .master("local[*]")
                .orCreate
```

Figure 13 Sample code of initializing a Spark Session.

- **Step 2** - After the SparkSession is created there are three files containing the data stored in JSON format. Those files are **fake_tweets.json**, **retweets_batch.json**, and **sample_tweets_stream.json**. Using Spark data can be ingested and using the JSON representation of an underlying tweet object it can infer it's schema. Spark uses the inferred schema for the catalyst optimizer to optimize the queries executed when using Spark SQL. While loading the data filtering on the columns should be performed, since not all fields are relevant for the investigation (and the size of the data will be reduced dramatically), as well as some flattening since some fields of interest are nested in columns.

```
private fun flattenRetweetStatus(data: Dataset<Row>): Dataset<Row> {
    return data.withColumn("retweeted_status_id",functions.col("retweeted_status.id"))
        .withColumn("retweeted_status_text", functions.col("retweeted_status.text"))
        .withColumn("retweeted_status_user_followers_count", functions.col("retweeted_status.user.followers_count"))
        .withColumn("retweeted_status_user_friends_count", functions.col("retweeted_status.user.friends_count"))
        .withColumn("retweeted_status_user_id", functions.col("retweeted_status.user.id"))
        .withColumn("retweeted_status_user_screen_name", functions.col("retweeted_status.user.screen_name"))
}
```

Figure 14 Sample code of field flattening process for retweet posts.

- **Step 3** - Moving forward and having cleaned the data, so that only fields and columns that of interest remain, a series of transformations gets executed to extract the following:
    - Tweets for which the sequence of their retweet posts exists.
    - Retweets for which traceback to the original Tweet post can be performed.
    - Replies for which traceback to the original Tweet post can be performed.

  To perform these actions Spark will cache the results of the previous steps as in-memory tables, as shown below:

```
tweets?.createOrReplaceTempView("tweet_posts")
retweets?.createOrReplaceTempView("retweet_posts")
replies?.createOrReplaceTempView("reply_posts")
```

Figure 15 Sample code of registering in-memory tables.

Using the above code there are three tables registered, on which queries can be performed. For brevity the code with the series of transformation will be skipped, but can be found on the GitHub repository https://github.com/polyzos/thesis and below we can see the queries that give the results from those transformed tables:

```
val tweetsWithRetweets = spark.sql(
   """
   SELECT *
   FROM tweet_posts
   WHERE id IN (
               SELECT retweeted_status_id
               FROM retweet_posts)
   """
)

val retweetPostsWithTweet = spark.sql(
   """
   SELECT *
   FROM retweet_posts
   WHERE retweeted_status_id IN (
               SELECT id
               FROM tweet_posts)
""")


val repliesWithTweets = spark.sql(
   """
   SELECT *
   FROM reply_posts
   WHERE in_reply_to_status_id IN (
                SELECT id
                FROM tweet_posts)
   """)
```

Figure 16 Sample code of Spark SQL queries.

- **Step 4** - The output results contain tweets objects, retweets object, and reply objects. All of them contain different fields and after the transformations taking place, there are more fields that need to be flattened and dropped. More specifically, fields kept are:

	Fields
1	Created_at
2	Id
3	In_reply_to_screen_name
4	In_reply_to_status_id
5	In_reply_to_user_id
6	Retweeted_status
7	Text
8	User

Table 2 Fields that we keep for the Tweet posts.

	Fields
1	Created_at
2	Id
3	Retweeted_status
4	Text
5	User

Table 3 Fields that we keep for the Retweet posts.

	Fields
1	Created_at
2	Id
3	In_reply_to_screen_name
4	In_reply_to_status_id
5	In_reply_to_user_id
6	Text
7	User

Table 4 Fields that we keep for the Reply posts.

- **Step 6 -** Lastly and after the preprocessing of the posts ends the final data output will be stored on the data lake once again to proceed to the next layer for further modeling and analysis.

One thing to note here is that the need is to uncover **relationships** inside a network and although there is data for tweet posts and their sequence of propagation, there is one key thing that's still missing. Social networks are all about user interaction, but there is no indication at the moment for the relationships between the users in the collected data. For that reason, while storing the cleaned data in the data lake, users that participate in those stories need to be extracted, so that later can be used to find the relationships between them.

3.3. Stage 3: The Rise of Graphs

3.3.1 Data Enrichment

As mentioned in the last section the relationships between the users in the social network data sample are missing. So, the first action that needs to take place, is to try and collect more data about those relationships. Initially, this seems like a simple task. Just query the Twitter API for the user relationship, i.e. if user x follows user y and store the result.

Unfortunately, it is not that simple. The Twitter API rate enforces a limit on the number of requests that can be performed. For example, to retrieve relationships between users, you can make 180 requests every 15 minutes. Now, imagine that we have 10k users in our graph. To find the relationships among them, we need 10k * 10k requests. With that rate limit, the amount of time needed to fetch all this data is way too much.

This thesis approach to tackle this problem is to select a subset of the stories, to decrease the number of users being involved. From the original **240** tweet posts, the research was conducted on a subset of **44** user stories. Ten different Twitter API credentials were used (this is the maximum number of credentials allowed by Twitter per user) to fetch this kind of information using multiple threads. The sample code for using multiple connections on different threads is shown below:

```
val threads = originalPostIds.chunked(2)
    .mapIndexed { index, stories ->
        Thread {
            processStories(stories,twitterConnections[index],connection)
        }
    }

threads.forEach { it.start() }
threads.forEach { it.join() }
```

Figure 17 Functional way of querying the Twitter API with multiple access tokens.

Even though there are 10 different threads doing the work and only a sample of the data, the process takes about ten days to retrieve the relationships between all of the users. Once again, all the fetched data gets stored in the data lake.

3.3.2. Modeling for Graph Databases

Finally, the process of collecting all the data needed is completed and now the modeling of all these relationships inside the social network can take place. One key point here is that social networks are all about user interactions and relationships. These interactions and relationships can be easily represented by what is known as a Social Graph. To quote from WhatIs.com, a Social Graph is a diagram that illustrates interconnections among people, groups and organizations in a social network. The term is also used to describe an individual's social network. When portrayed as a map, a Social Graph appears as a set of nodes that are connected by lines. Each node, which may also be referred to as an actor, is a data record. The connecting lines used to map relationships between data records are referred to as edges, ties or interdependencies.

An illustration of a Social Graph can be found below:

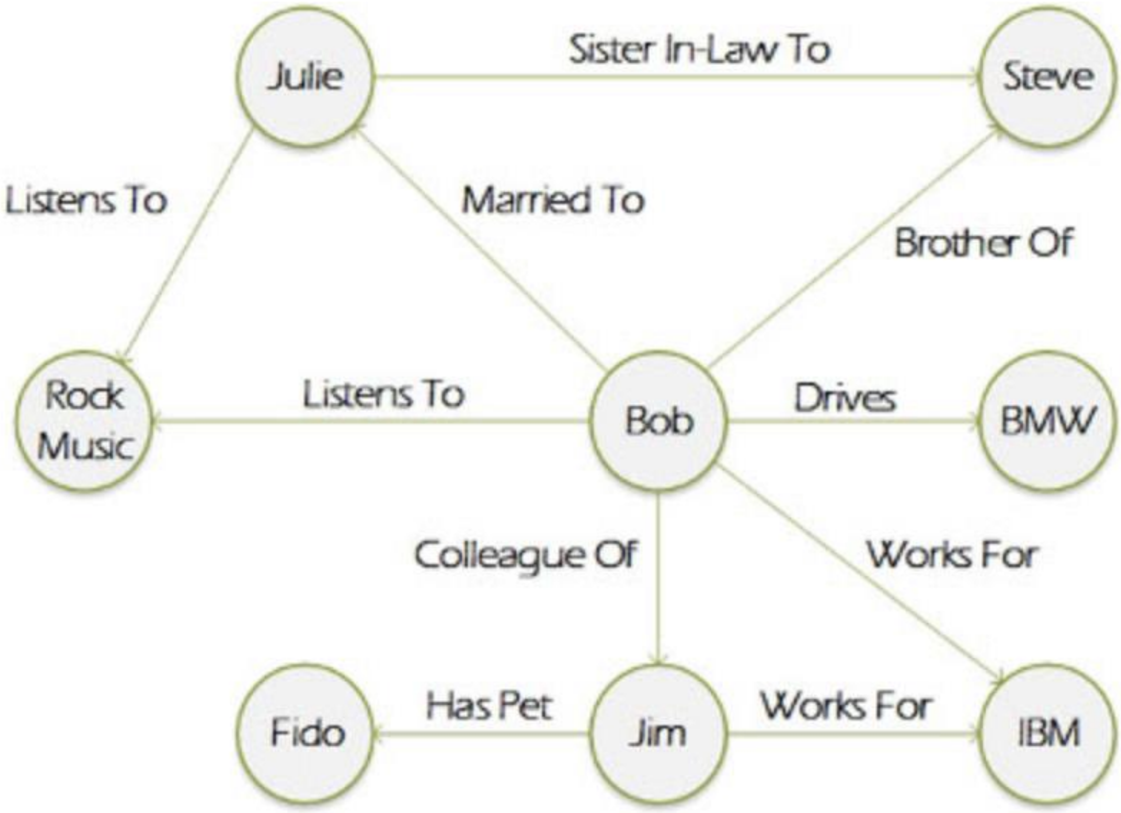
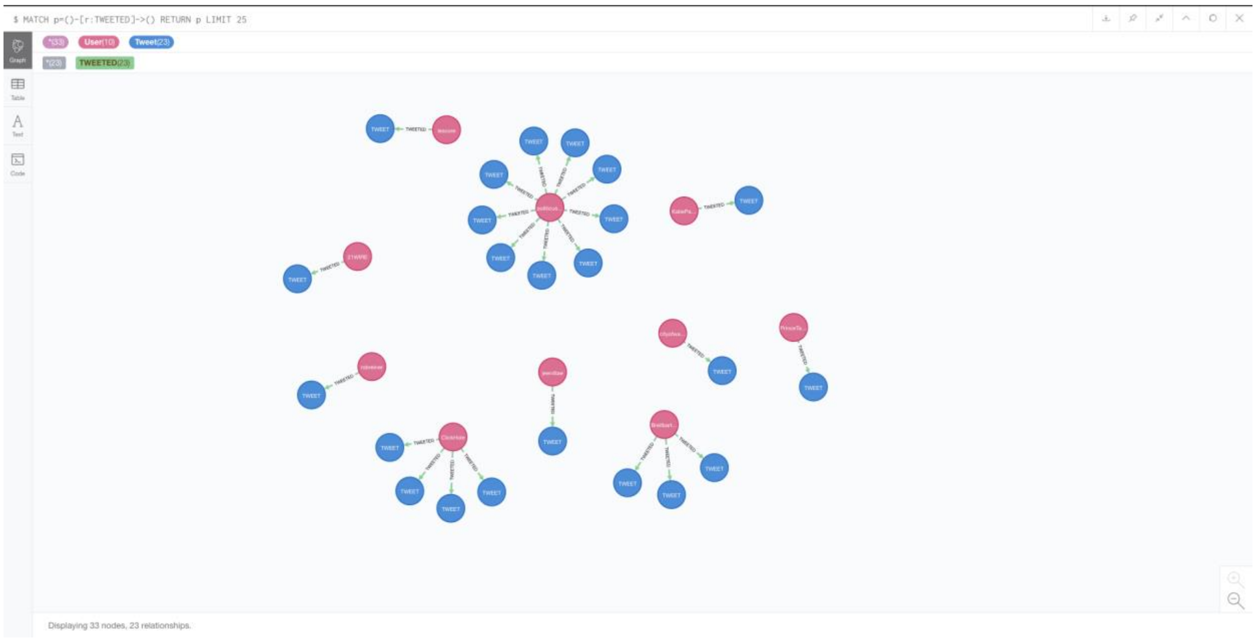


Figure 18 An illustration of a Social Graph where each node is an Entity or something that can be modeled and the edges are the relationships between two nodes. For example, node "Bob" which is a Person and node "BMW" which is a car model have a relationship called "Drives". This means that Bob drives a BMW car.

What is needed next is to take the data and build a Social Graph. The way the modeling of the graph takes place is by combining all the different objects (actors). A tweet story is going to be a graph node that needs to be connected with the user that made that post, with some kind of relationship. The tool of choice for this task is going to be the Neo4j Graph Database platform as discussed in section 2.3.4. This first relationship will be in the form of <Node: User x> -> [Relationship: Tweeted] -> <Node: Tweet Post>.

The next step is to build on that and construct a "chain" of user interactions, which includes retweets of that post and replies, based on the time those events took place. A retweet post node can have an incoming edge from the user node that made the retweet and an outgoing edge pointing back to the previous post that in a timely manner happened before that (if it's the original tweet post, then it points back to the parent node) and the same approach will take place for the replies as well. The edges for the retweet nodes in the chain will have a RETWEETED relationship holding a label indicating the timestamp that the event took place; the replies will have a REPLIED relationship with a timestamp label as well.

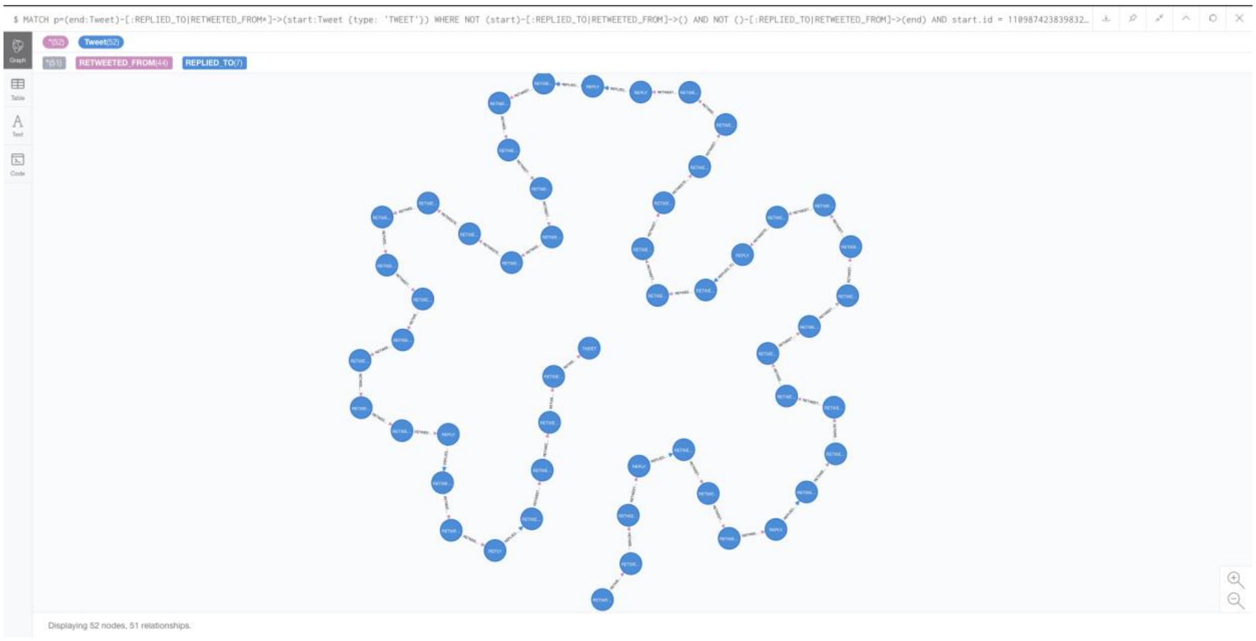


```
$ MATCH p=()-[r:TWEETED]->() RETURN p LIMIT 25
```

Displaying 33 nodes, 23 relationships.

Figure 19 The result of a neo4j query using the Cypher language, that shows the Tweeted relationship.

What can be seen here is different tweet stories that exist inside the graph database, that have been limited for performance reasons. Specifically, the red nodes are users, the blue nodes are the tweets and the edge between a red and a blue node is the tweeted relationship showing that a user posted a tweet.



```
$ MATCH p=(end:Tweet)-[:REPLIED_TO|RETWEETED_FROM*]->(start:Tweet {type: 'TWEET'}) WHERE NOT (start)-[:REPLIED_TO|RETWEETED_FROM]->() AND NOT ()-[:REPLIED_TO|RETWEETED_FROM]->(end) AND start.id = 110987423839832…
```

Displaying 52 nodes, 51 relationships.

Figure 20 The result of another neo4j query using the Cypher language, that shows a Tweet Chain.

And here we can see all the tweets from a tweet story. There are two types of relationships that exist between two Tweet (blue) nodes. The first one is the RETWEETED_FROM denoted with a pink color and the second one is the REPLIED_TO denoted with a blue color. In Figure 20, a chain occurs that starts from a tweet and ends to the origin tweet of the tweet story. Tweets are ordered by time meaning that the edge between two Tweet nodes has its arrow to the Tweet node that happened first in order of time.

```kotlin
tweetStories
    .forEach {
        // foreach post find its retweets
        val fetchedRetweets = GraphUtils.findPostRetweets(it.id, spark)
        val fetchedReplies = GraphUtils.findPostReplies(it.id, spark)
        val user = ParsedUser(it.user_id, it.user_screen_name, it.user_followers_count, it.user_friends_count)
        println("Tweet ${it.id} has ${fetchedRetweets.count()} retweets and ${fetchedReplies.count()} replies.")

        nodeRepository.createUserNode(user)
        nodeRepository.createTweetNode(it)
        relationshipRepository.createTweetedRelationship(user, it)
        val fetchedReactions: MutableList<Row> = mutableListOf()
        fetchedRetweets.collectAsList()
            .forEach { fr -> fetchedReactions.add(fr) }
        fetchedReplies.collectAsList()
            .forEach { fr -> fetchedReactions.add(fr) }

        fetchedReactions.sortBy { fr -> Utilities.parseDate(fr.getString(0).split(".")[0]) }
        fetchedReactions.forEachIndexed { index, fr ->
            if (fr.size() == 10) {
                val reply = Utilities.rowToParsedReply(fr)
                val replyUser = ParsedUser(
                    reply.user_id,
                    reply.user_screen_name,
                    reply.user_followers_count,
                    reply.user_friends_count
                )
                nodeRepository.createUserNode(replyUser)
                nodeRepository.createReplyNode(reply)

                relationshipRepository.createRepliedRelationship(replyUser, reply)
                if (index == 0) {
                    relationshipRepository.createRepliedToRelationship(it.id, reply)
                } else {
                    if (fetchedReactions[index - 1].size() == 10) {
                        val previous = Utilities.rowToParsedReply(fetchedReactions[index - 1])
                        relationshipRepository.createRepliedToRelationship(previous.id, reply)
                    } else {
                        val previous = Utilities.rowToParsedRetweet(fetchedReactions[index - 1])
                        relationshipRepository.createRepliedToRelationship(previous.id, reply)
                    }
                }
            } else {
                val retweet = Utilities.rowToParsedRetweet(fr)
                val retweetUser = ParsedUser(
                    retweet.id,
                    retweet.user_screen_name,
                    retweet.user_followers_count,
                    retweet.user_friends_count
                )

                nodeRepository.createUserNode(retweetUser)
                nodeRepository.createRetweetNode(retweet)

                relationshipRepository.createRetweetedRelationship(retweetUser, retweet)

                if (index == 0) {
                    relationshipRepository.createRetweetedFromRelationship(it.id, retweet)
                } else {
                    if (fetchedReactions[index - 1].size() == 10) {
                        val previous = Utilities.rowToParsedReply(fetchedReactions[index - 1])
                        relationshipRepository.createRetweetedFromRelationship(previous.id, retweet)
                    } else {
                        val previous = Utilities.rowToParsedRetweet(fetchedReactions[index - 1])
                        relationshipRepository.createRetweetedFromRelationship(previous.id, retweet)
                    }
                }
            }

        }
    }
```

Figure 21 The sample code that constructs the graph in the neo4j database.

For each tweet story, all of the retweets made are retrieved as well as all the replies. Using the formula described above the social graph is constructed and stored in the Neo4j database. All this graph construction takes place using the Cypher query language. Cypher is Neo4j's graph query language https://neo4j.com/docs/cypher-manual/current/ that allows users to store and retrieve data from the graph database. Neo4j wanted to make querying graph data easy to learn, understand, and use for everyone, but also incorporate the power and functionality of other standard data access languages. This is what Cypher aims to accomplish.

Cypher is a vendor-neutral open graph query language employed across the graph ecosystem. Cypher's ASCII-art style syntax provides a familiar, readable way to match patterns of nodes and relationships within graph datasets. Like SQL, Cypher is a declarative query language that allows users to state what actions they want to be performed (such as match, insert, update or delete) upon their graph data without requiring them to describe (or program) exactly how to do it.

In 2017, Neo4j, Inc. decided to open source the Cypher language and make the most popular graph query language available to any technology provider with the aim that Cypher becomes the "SQL for graphs." The openCypher project provides an open language specification, technical compatibility kit, and reference implementation of the parser, planner, and runtime for Cypher. It is backed by several companies in the database industry and allows the implementers of databases and clients to freely benefit from, use, and contribute to the development of the openCypher language.

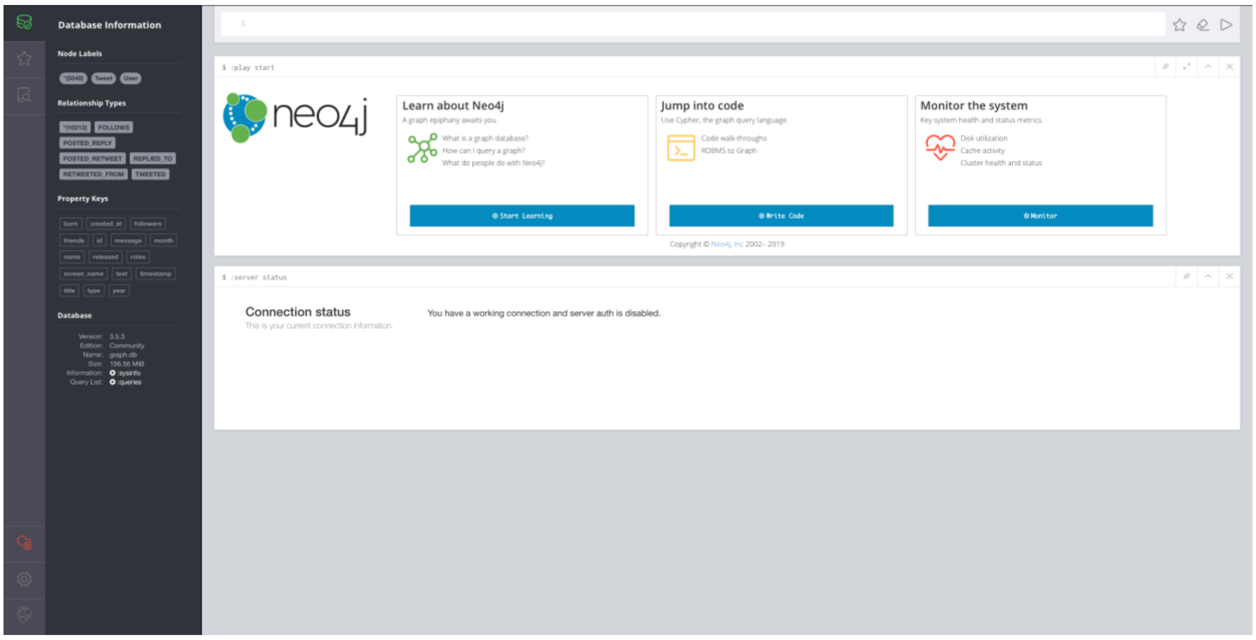
```kotlin
override fun createUserNode(id: Long, screenName: String) {
    try {
        driver.session()
            .writeTransaction {
                it.run(
                    """
                        MERGE (user:User {screen_name: '$screenName', id : $id})
                        RETURN user.id""",
                    Values.parameters("id", id, "screen_name", screenName)
                )
                    .single().get(0)
            }
    } catch (e: Throwable) {
        println("Failed txn in createUserNode: $e")
    }
}
```

Figure 22 Sample code for creating a user node in neo4j.

```kotlin
/**
* Follows Relationship with Long Values MATCH Lookup
*/
override fun createFollowsRelationship(follower: Long, followee: Long) {
    try {
        driver.session()
            .writeTransaction {
                it.run(
                    """
                        MATCH (follower:User {id: $follower})
                        MATCH (followee:User {id: $followee})
                        MERGE (follower)-[:FOLLOWS]->(followee)
                        RETURN follower.id, followee.id""",
                    Values.parameters("id", follower, "id", followee)
                )
                    .single().get(0)
            }
    } catch (e: Throwable) {
        println("Failed txn in createFollowsRelationship: $e")
    }
}
```

Figure 23 Sample code for creating a FOLLOWS relationship between two users in neo4j.

Finally, after running the code and all the data has been stored in the database the social Graph can be visualized using the Neo4j Browser. A visual representation of the graph gives a better perspective of how the data looks, reveals the relationships between them and makes it easier for the eye to work with the data at hand.



Version: 3.5.3
Edition: Community
Name: graph.db
Size: 156.56 MiB
Information: :sysinfo
Query List: :queries

```
$ :play start
```

Learn about Neo4j
A graph epiphany awaits you.
What is a graph database?
How can I query a graph?
What do people do with Neo4j?

Jump into code
Use Cypher, the graph query language.
Code walk-throughs
RDBMS to Graph

Monitor the system
Key system health and status metrics.
Disk utilization
Cache activity
Cluster health and status

Copyright © Neo4j, Inc 2002- 2019

```
$ :server status
```

Connection status
This is your current connection information.
You have a working connection and server auth is disabled.

Figure 24 This is the neo4j browser. Source: https://neo4j.com

On the left panel of the neo4j browser details about the graph database can be seen, like the version, the size of the database and the number of nodes and relationships in the graph database. On the top side, there is a box where users can input queries to run on the graph database and below the results of those queries are demonstrated.

3.4. Stage 4: Gaining Insights

3.4.1. Finding Potential Bots

The three previous stages would eventually lead to this point. Hereby taking all the data being modeled, processed and saved in the graph database further analysis can be conducted to find potential bots.

The culprit here is a tweet story that has the properties of a fake news story, and an attempt is being conducted to try and acknowledge if a tweet story is indeed a bot and to flag it for further analysis. The main focus will be on the flow of information inside the network and how the nodes inside it look like with respect to the rest of the nodes.

3.4.2. Information Propagation

Let's try to explain how the propagation of stories evolves over time within the twitter graph. For starters, let's consider an origin node, someone who starts/posts a tweet story, and then various users that react on that post, we consider retweets and replies as a reaction to a tweet story. Users who can react to it are followers of the origin node and/or followers of a user that already reacted to the original post or one of its retweets and/or replies.

The first task is to infer how a user could have been informed about the specific tweet story. We consider that a user "b" could have been informed about the story from another user "a" if, user "b" follows user "a" AND user "a" reacted to the story before user "b". This way, for the given user "b" and a given story "s" which the user is involved, we mark all the potential ancestors of "b" with respect to "s", i.e., other users who also reacted to "s" and are connected to "b" via some path from "b" to "a" in the twitter subgraph. The sequence of reaction times when one moves from "b" to "a", and along that path, must be strictly decreasing, meaning that each potential ancestor of "b" must have reacted to "s" earlier than "b".

To put it in perspective, this creates a temporal graph from the Twitter subgraph and activates the edges if and only if the timestamps increase if one were to traverse the graph as a tree where the origin node is the root and the nodes with no incoming edges are the leaves.

Next plot a graph, where nodes on the graph are the users that reacted to the story and the edges are directed from some user "a" (a parent-ancestor) to another user "b" (a child-descendant) when the previous condition holds. The directed edges also have the timestamp of the reaction.

Another structural feature that is being explored in the subgraph of a story is the discovery of Nodes of Interest (NOI). A NOI is a node inside the story subgraph that has more incoming edges than the average node of the graph.

In the figures below, you can see the difference in the topologies of two-story subgraphs. Figure 25 shows the subgraph of a legitimate story, with only one Node of Interest (NOI), i.e., it looks like a star graph. Figure 26, on the other hand, show the subgraph corresponding to s fake story; this subgraph has multiple Nodes of Interest (NOIs) and does not look like a star graph. What is interesting about those two different stories, one being flagged as fake and the other one being legitimate, is that they have almost the same number of users reacted and almost the same count of relationships.

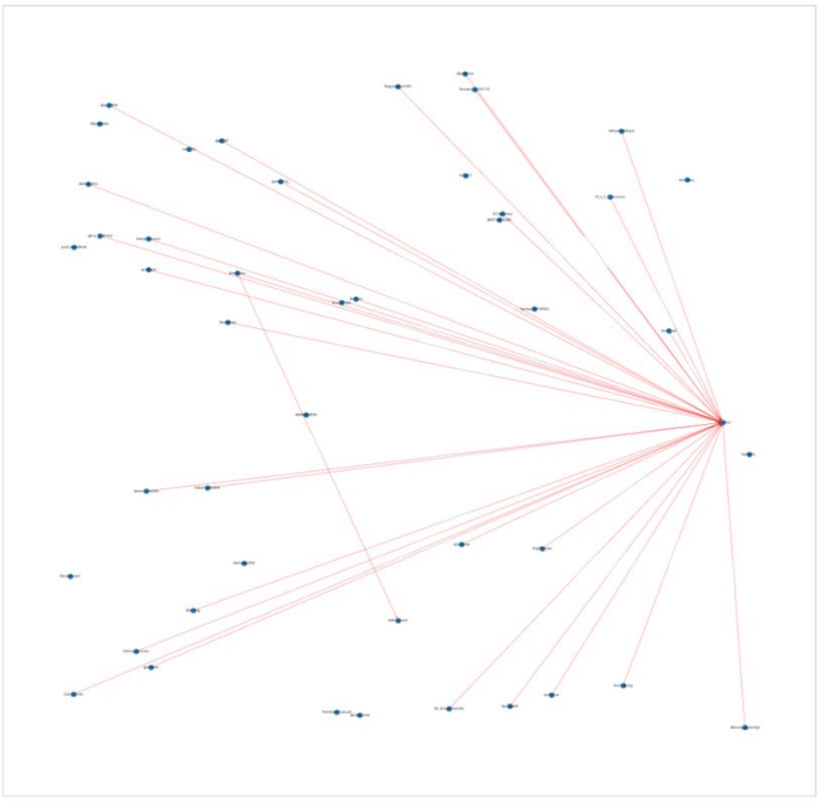


Figure 25 A legitimate story, with one Node of Interest (NOI) resulting in a star graph.

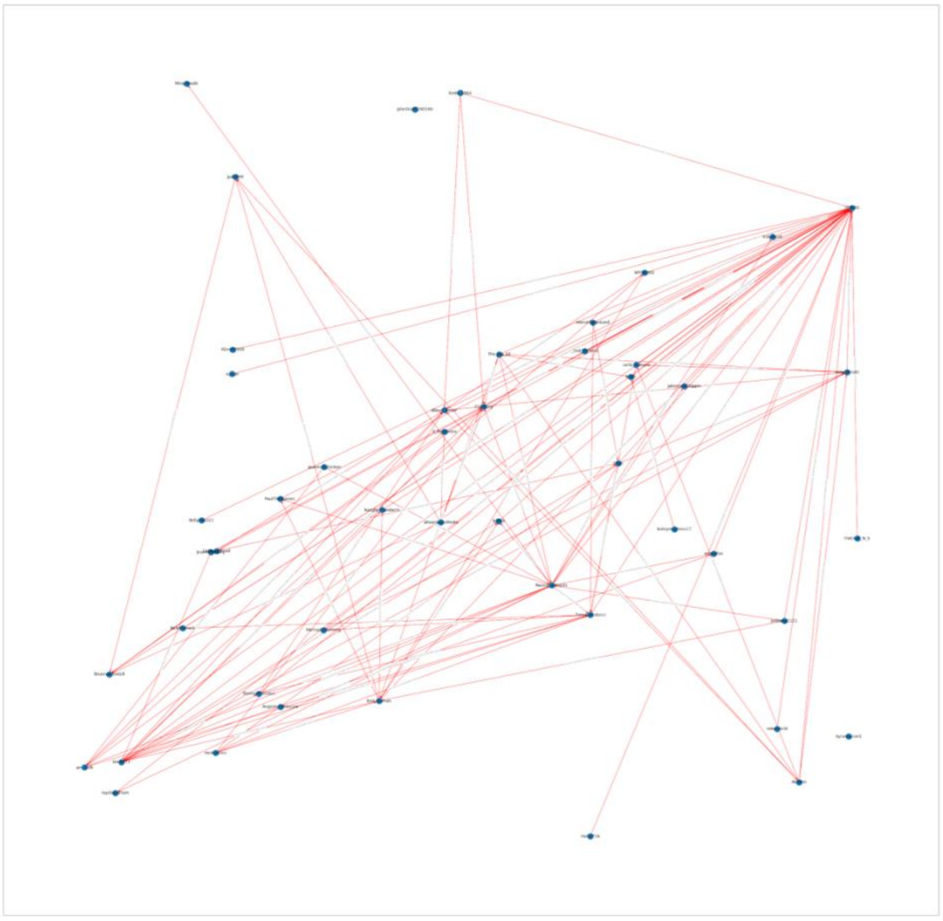


Figure 26 A fake story, with multiple Nodes of Interest (NOIs). It does not form a star graph.

Now with the graphs in place, we can see how a fake news story looks like in the dataset. However, considering that the graph takes this form after a few days, looking at it allows us to flag the story as fake after it has been spread across a social network and impacted users in a bad way.

3.4.3. Graph Analytics

Gaining insights from the structural properties of story subgraphs about the nature of the story itself is the prime goal of this work. We can start understanding how people react to fake or legit news stories by extracting information found in the graphs that we create through the process explained earlier.

Let's proceed now with the consideration of algorithms which provide graph analytics, e.g., characteristic vectors quantifying the distances of the involved users in a story from its originator, that can help create some structural fingerprint that will allow the early recognition of an emergent

story’s nature. In this thesis, the SimRank similarity algorithm is being used to provide such a structural (and information-theoretic) footprint.
SimRank similarity is the algorithm of choice to provide similarity of the nodes inside a network. Various aspects of objects can be used to determine similarity, usually depending on the domain and the appropriate definition of similarity for that domain. In a document corpus, matching text may be used, and for collaborative filtering, similar users may be identified by common preferences.

The SimRank algorithm is a general approach that exploits the object-to-object relationships found in many domains of interest. Our principal assumption is that, for a particular story, the most important node is its originator. Consequently, we apply the SimRank algorithm to quantify the importance of the remaining nodes in the story’s subgraph. On the Web, two pages are related if there are hyperlinks between them. A similar approach can be applied to scientific papers and their citations, or any other document corpus with cross-reference information. In the case of recommender systems, a user’s preference for an item constitutes a relationship between the user and the item. Such domains are naturally modeled as graphs, with nodes representing objects and edges representing relationships.

The intuition behind the SimRank algorithm is that, in many domains, similar objects are referenced by similar objects. More precisely, objects A and B are considered to be similar if they are pointed from objects c and d, respectively, and c and d are themselves similar. The base case is that objects are maximally similar to themselves.

It is important to note that SimRank is a general algorithm that determines only the similarity of structural context. SimRank applies to any domain where there are enough relevant relationships between objects to base at least some notion of similarity on relationships. Obviously, the similarity of other domain-specific aspects is important as well; these can — and should be combined with relational structural-context similarity for an overall similarity measure. However, in this approach, the similarity of the structural context is enough to provide the information needed to plot a tweet story graph.

Chapter 4. Experiments

The most resource-demanding part of our Thesis was clearly the data collection and preprocessing part. Everything was run on our infrastructure on Google Cloud Platform as stated in Section 2.3.5. Our infrastructure consisted of a cluster with multiple nodes of 4 vCPUs and 24GB of RAM where we run our Spark Jobs, and a highly available neo4j database with 2 nodes having 2 vCPUs and 12GM of RAM and 250GB of persistent solid-state disk storage (SSD). All the parts of infrastructure ran on a virtual private cloud (VPC) with private subnets inside Google Cloud with high bandwidth network access (10Gbps).

The experiments that we run for the graph analytics part of our Thesis, were run on our personal computers of 6 cores and 16GB of RAM.

4.1. Digging into the data

Now is the time to guide you through some experiments conducted on the collected data. To gain a better insight into the dataset that curated, we searched for the number of appearances for each user that reacted to the tweet stories. This will allow for future-flagging of accounts that seem like spam or bots. Something like this could further increase the reliability and accuracy of the application.

Screen Name	# of Appearances
manyfeathers514	2
Mike_Padgett	3
kungfu_mandarin	3
monkey201905	2
Iconoclastttt	2
Mike_Charles	2
dazgood500	2
miniacmarcus	2
ClarkReidSF	2
brscheaffer	2
snekmeseht	2

BoriquaGoddess	3
politicususa	7
lavenderblue27	6
RSmith1935	3
Alobhin	2
jolady42	2
bndsnoopy61	5
Xantl	2
Atomizer_X	2
GGodLove77	2
MJGarciaKCMO	2
lauracgilleslil	3
cat17534	3
catsformede	5
donniedingdong	2
PaulTheNurse1	2
Dragon17xrt	2
souljahsingh	4
KatGodspell	3
lance_pool	2
Burnouts3s3	2
JohnHawkwood62	2
Maureen95410753	3
Georgia17753006	2
BreitbartNews	5
tonto_1964	2

GeostompX	4
Willeamon	3
obiekezie_b	2
forsurftoo	2
CRG_CRM	2
alliomack	2
am_mirk	3
michael05833466	2
enrirosasdiaz	2
ClickHole	4
hala_at_my_girl	2
santo_aol	2
agimediata	2
roentgens9	2
ROOSTERTLC	4
WhobbaBobba	3
JieWang1111	2
Thatsworrisome	2
uciccolella	4
GaryC23619095	2
Gitmoite	2
silverthornn264	4
PolitJunkieM	5
MiaTorres8	2
FCriticalThink	2
mcmounes	3

maxrafaelwaller	2
sylc5star	2
LadyGloriousjax	2
KnucklesMal0ne	2
BobJavorcic	2
Bravery05089991	2
malama00	2
ArtisticEye1	2
NanaDavis_46	2
infowarrior2019	2
dhillos318	2
Towelprazolam	2
MarcoPo15951142	2
yylow02	2
My2centz_1	3
marios_brother	2
darmagirl24	3
LovToRideMyTrek	4
TheCeeside	2
KarlSanchez13	2
AnttiHarjula	2
Bellynda321	2
Harry71A	2
jpwilloughby	2
kevskewl	2
BigBrotherQandA	2

SylviaZ1913	2
dougmac0063	2
charlesazorn	2
calling12001	3
maritzasolito	2
mryfrtsn	2
JoyOfDachshunds	2
TinyFingerTrump	2
21WIRE	3
jamiescottwrite	2

Table 5 Number of appearances for each user that appears in our dataset. Only users with at least 2 appearances show up on this table.

To have a better look at the flow of information from a random node of the graph to the origin node, let's create a function that shows only the subgraph from a node "a" to the origin node.

```python
def subgraph_from_node(node):
    nodes = [node]
    queue = [neighbor for neighbor in G.neighbors(node)]
    while queue:
        next_one = queue.pop()
        if next_one not in nodes:
            nodes.append(next_one)
        queue.extend(neighbor for neighbor in G.neighbors(next_one))
    return nodes
```

Figure 27 Sample code that produces a subgraph from a user node to the origin node.

Then plot the subgraphs and take a look at how random nodes, in the two previous stories, were informed about the tweet story.

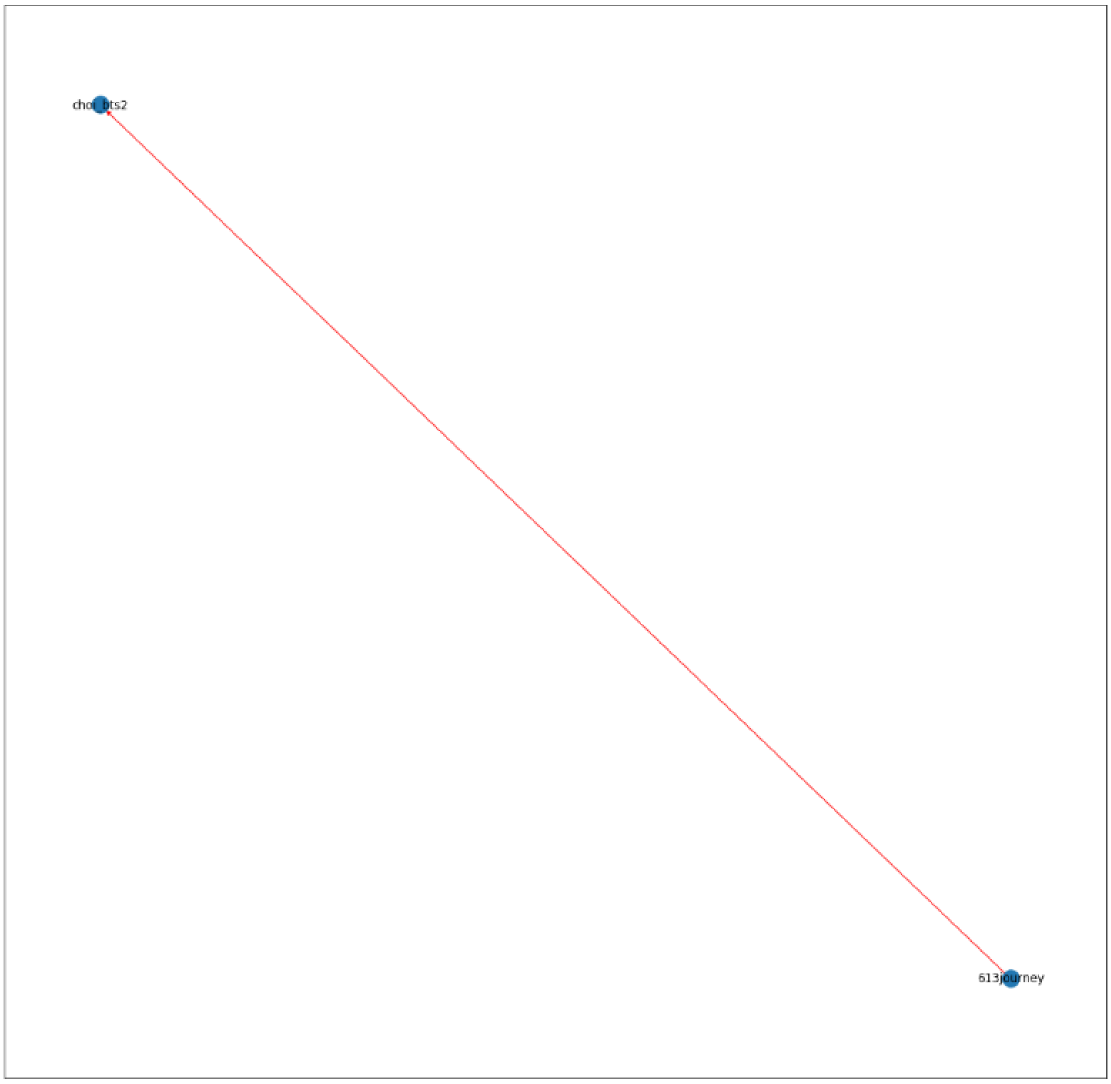


Figure 28 A subgraph of the legitimate tweet story from Figure 25. The propagation of the origin node "choi_bts2" to the user "613journey". The arrow shows that the user "613journey" follows "choi_bts2".

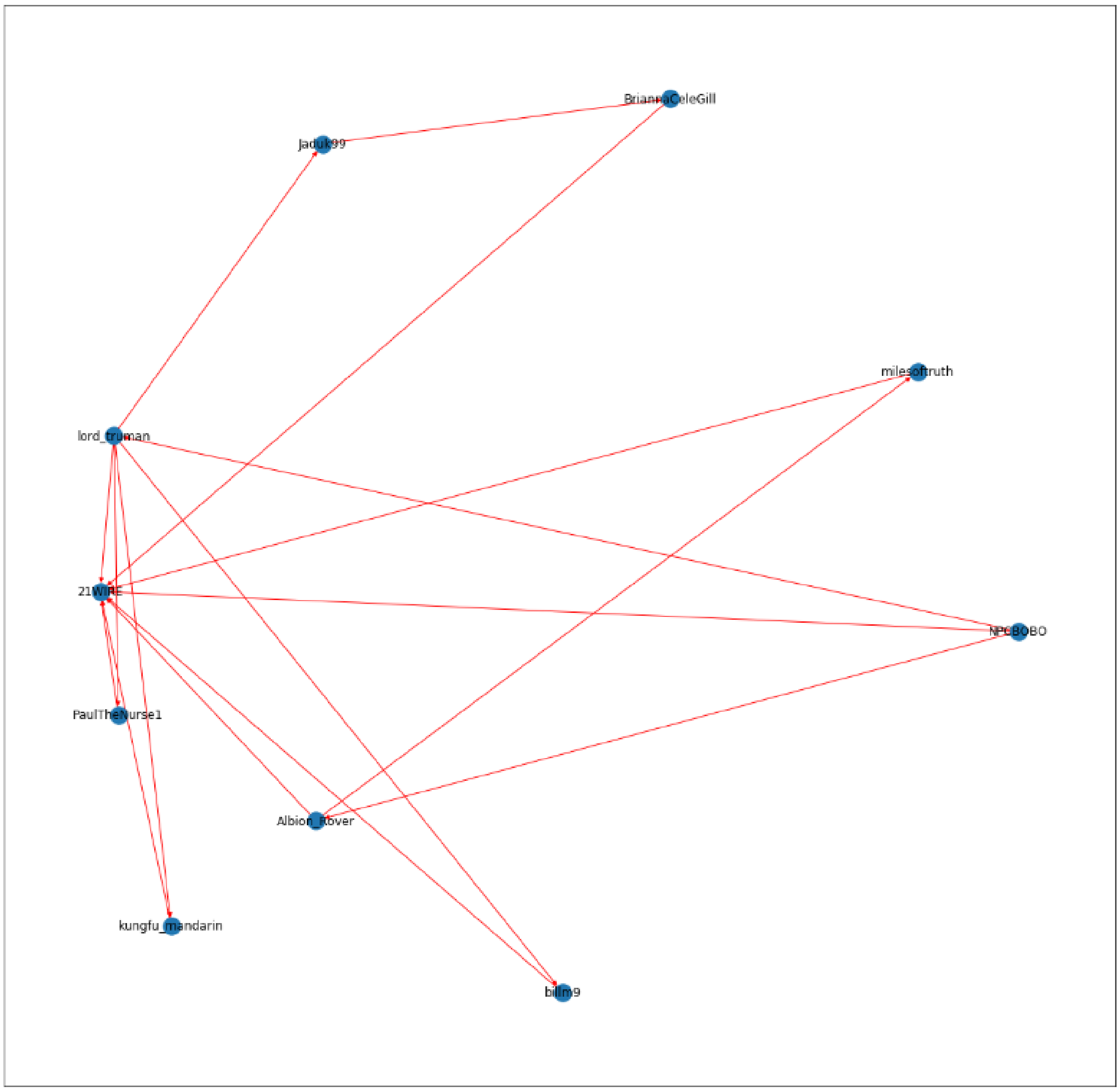


Figure 29 A subgraph of the fake tweet story from Figure 26. The propagation of the origin node "21WIRE" to the user "NPCBOBO". We can see multiple paths that lead to the origin node.

From the above figures, we can see that there is a huge difference between the two stories. However, no assumptions can be made if what happens here is due to the one being a fake and the other one being a legitimate news story. So, there is a need to dig a little bit deeper into our data.

Let's collect some statistics about the tweet stories, by querying the Twitter API for every pair of users in each tweet story to find their relationship. The API would, then, return true if user "a" follows user "b" and false if user "a" did not follow user "b" and it would do so for the reversed relationship as well. True if user "b" follows user "a" and false otherwise.

Afterward, let's count the number of true, false and null relationships for every tweet story. Again, true means that user "a" follows user "b", false means that user "a" does not follow user "b" and null if the API could not return a result. Null exists to state that we cannot find the relationship between two users, and that is because at the point of the query either user did not exist, due to getting banned by Twitter Staff or due to them deleting their account, or because either of them turned their account private.

After that, calculate the percentage of how many of the true relationships were users that follow the origin user of the tweet story. This can help understand the propagation of the story inside the social network. If the percentage is 100% or close to it, this would result in a star-type graph; meaning that all users have probably heard the story from the origin node making it the Node of Interest (NOI) inside the graph. However, a lower percentage may result in more than one Nodes of Interest (NOIs).

As you can see in the tables below, the existence of multiple Nodes of Interest (NOIs) is closely related to the tweet story being flagged as fake inside our dataset. While the existence of one Node of Interest (NOI) does not flag the tweet story as fake.

When some attributes of the tweet story are the same in two counterpart stories (a legitimate and a fake) we can assume that, indeed, the propagation speed of the tweet stories can help us flag the story as fake or not. Some of these attributes are the number of users reacted, the count of relationships in a tweet story.

Origin Tweet ID	**True Count**	**False Count**	**Null Count**	**Relationships Count**	**% of users following origin**
1139836584998637569	1	701	0	702	100
1139849809634791424	1	1189	0	1190	0
1139876779001298946	0	462	0	462	0
1139885238912348161	4	1476	2	1482	25
1139893971461857280	4	502	0	506	50
1139969808680857601	5	547	98	650	100
1139971037620310016	2	460	44	506	100
113997723250544897	9	747	0	756	77.78
1140035697019183105	10	1180	70	1260	100
1140037496367009792	5	1401	0	1406	100
1140089744799358978	31	1949	90	2070	96.77
1140199564273573889	0	600	0	600	0
1140210192631701504	0	2862	0	2862	0
1140223773804658688	3	503	0	506	66.67
1140304883242942465	7	863	0	870	100
1140351859464404992	23	627	0	650	100
1140385690695245824	0	1422	3874	5296	0
1140410386753003520	1	1481	0	1482	100

1140486425286082560	9	1113	0	1122	88.89
1140550497486561280	22	1384	0	1406	100
1140553848743841798	21	735	0	756	100

Table 6 Tweet stories that are considered as legitimate in our dataset.

Origin Tweet ID	True Count	False Count	Null Count	Relationships Count	% of users following origin
1139865460953026560	51	2705	0	2756	100
1139880274962059264	55	3025	0	3080	85.45
1139887625609895936	276	46598	0	46874	75.72
1139906468315095040	50	502	0	552	46
1139911776659787776	97	2353	0	2450	28.87
1139914664685580288	76	1730	0	1806	46.05
1139923398132424705	28	524	0	552	82.14
1139925597583532032	35	1087	0	1122	74.29
1139927656403746821	82	3458	0	3540	64.63
1139935778182717440	50	2020	0	2070	88
1139948716608167937	44	948	0	992	56.82
1139980218293215233	185	2071	0	2256	20.54
1139985715570384897	75	1407	0	1482	42.67
1140001077691133959	34	1688	0	1722	94.12
1140061456060297217	53	3027	112	3192	96.23
1140078466764853248	88	6074	158	6320	84.09
1140170052374859776	36	834	0	870	75
1140171260477018112	52	2018	0	2070	73.08
1140240721619955714	75	1047	0	1122	41.33

1140242652480376832	25	575	0	600	92
1140306312942800898	68	2002	0	2070	60.29
1140392133825835008	44	2026	0	2070	93.18

Table 7 Tweet stories that are flagged fake in our dataset.

One can see from the last column of Table 6 that legitimate stories take the form of star graphs, while the fake stories of Table 7 have a different structure from the legitimate stories resulting in non-star graphs.

Finally, taking all the intermediate nodes, namely those that are not origin nodes but have incoming edges and querying the Twitter API produced interesting results. The query returned that 87% of those users existed, even after two months' period. The rest of the intermediate nodes were deleted, probably from the Twitter staff. This shows that there could be bots that are scaling the spread of fakes news, namely the deleted users the query did not return. However, the rest of the users also help the spread of fake news and there could be two reasons for that. It could be that Twitter has not found those accounts yet because the process of making twitter bots got a lot more sophisticated, or the users that reproduce the content do so in a manner that look like bots because they don't take the time to cross-check and validate the news story.

4.2. Analyzing with SimRank

The framework being used is to make various computations on the graphs, including the calculation of SimRank similarity vectors, is called networkx. Networkx is a vastly used python package which is open source and comes with a great community.

While using the SimRank similarity algorithm of networkx we came across an error that occurred when some nodes of the graph had no in degrees or no out degrees. After spending some time with the community, the source code, and the documentation we managed to find a fix.

We, then, contributed our fix back to the open-source project which was approved and merged into the official networkx package to the soon-to-be-released 2.4 version of the package.

A better representation of the graphs introduced in Section 3.4.2, can be achieved using the SimRank vectors to calculate the size of each node. This can show how the average node differs from the Nodes of Interest (NOIs). So, the next step is to run the SimRank similarity algorithm on those stories and use vectors to calculate the size of the nodes.

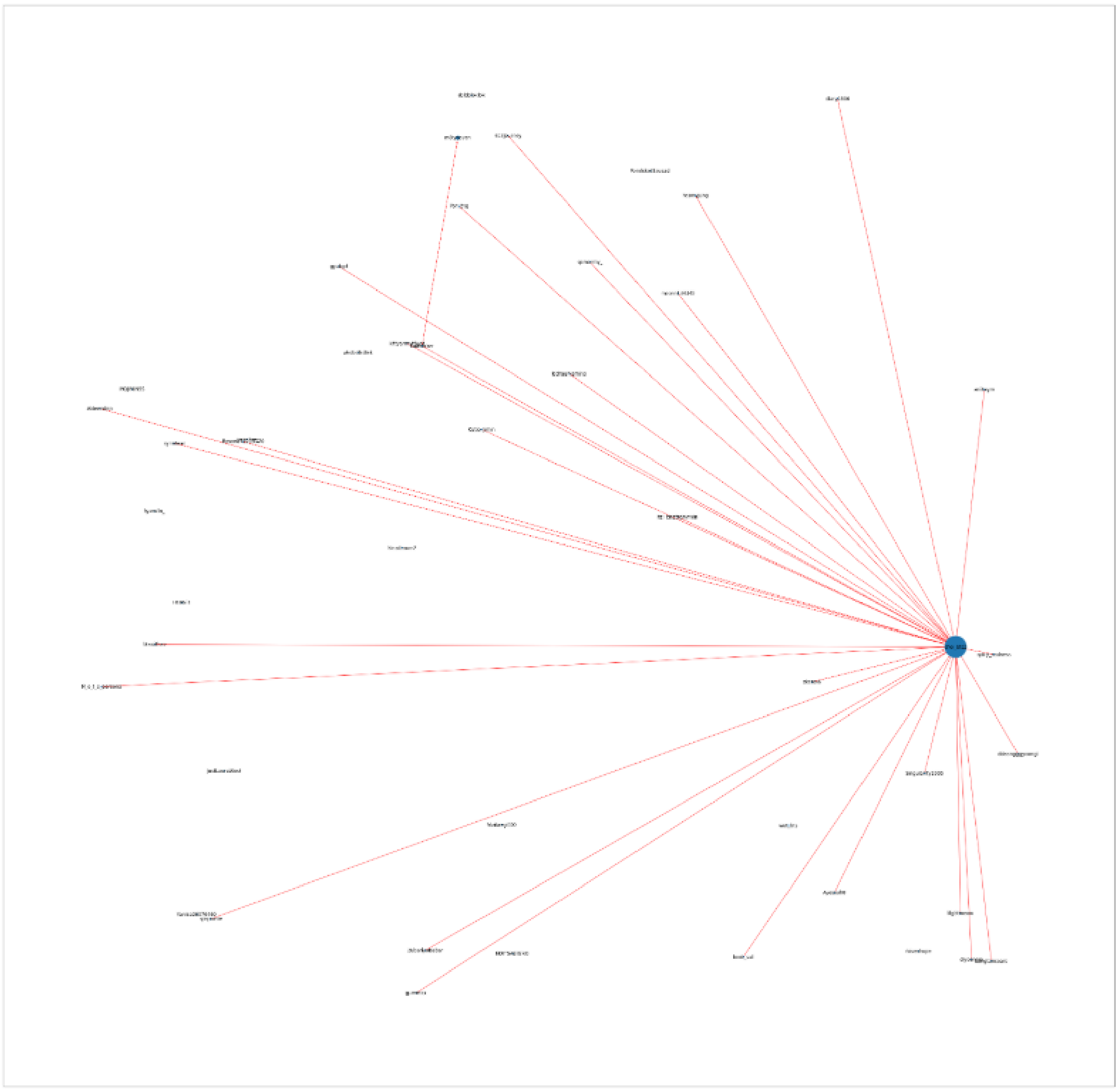


Figure 30 A legitimate story, with node sizes based on the SimRank vectors.

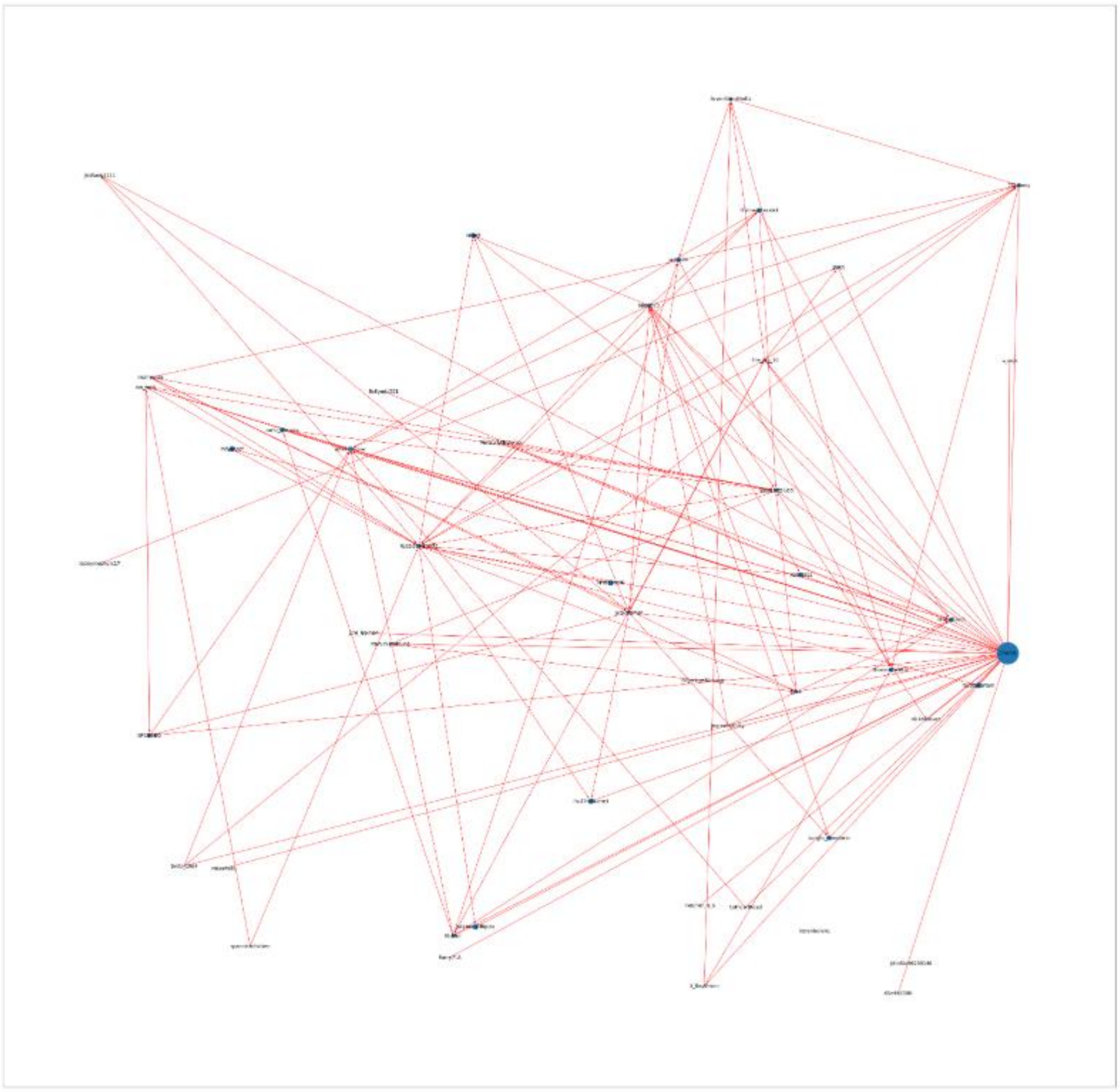


Figure 31 A fake story, with node sizes based on the SimRank vectors.

But this is still not a good way to tackle the fake news flagging problem. What we need is a way to separate legitimate news from fake news and flag them for further analysis, before they have done damage in the network they are spreading.

We could make better use of the already computed SimRank similarity vectors. We can take those vectors from every tweet story that we have and fit them to a power-law function and, then, plot the results.

We are using the powerlaw package for python. The more non-zero values we have produced by the SimRank similarity algorithm the more accurate our distribution will be. Below, we can see two figures, one for the legitimate tweet stories and one for the fake tweet stories.

Due to the low-to-no similarity of the nodes of a legitimate tweet story the vectors produced by the SimRank similarity algorithm are zero for the most part. This means that the power-law function -after finding the min and max values- finds less than two values and therefore cannot plot the distribution as shown in Figure 32. And that's the case for almost all the legitimate tweet stories.

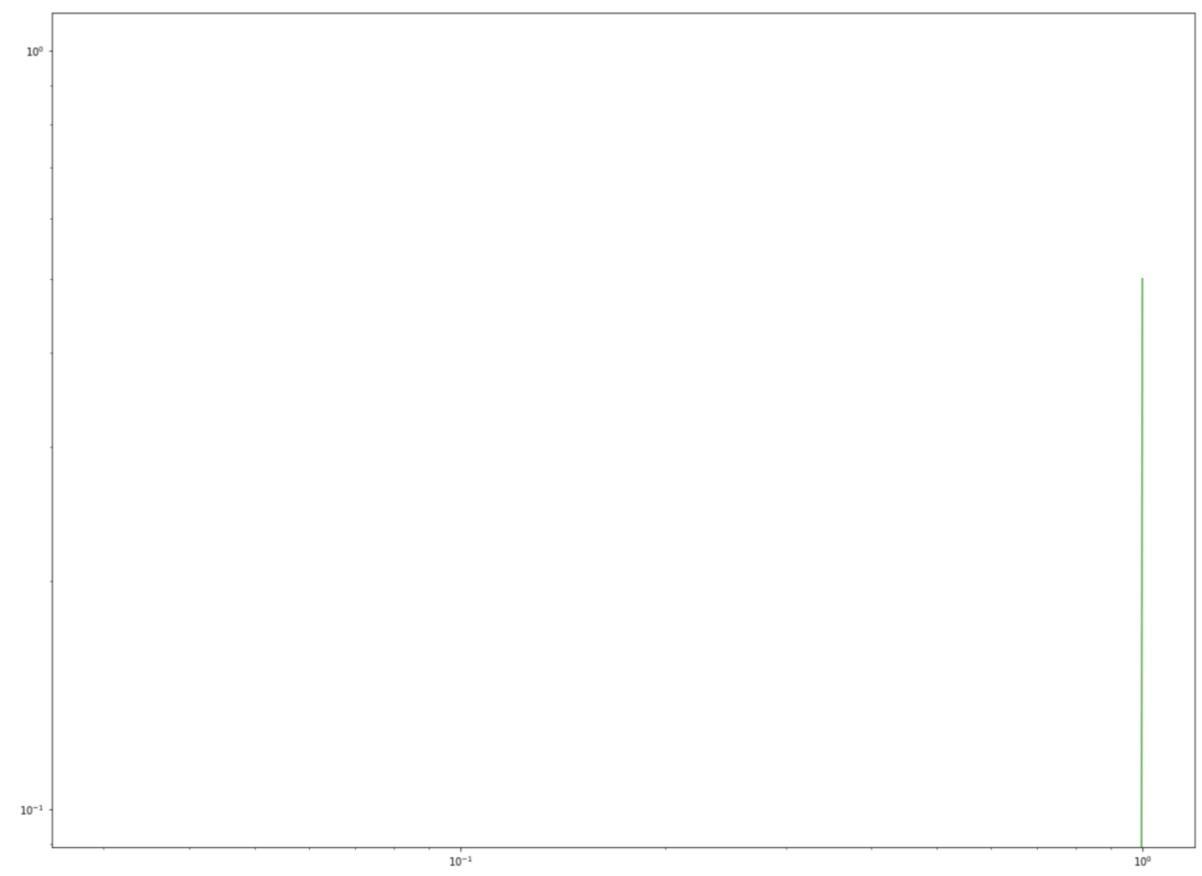


Figure 32 Power-law distribution for legitimate tweet stories.

However, the SimRank vectors from Fake News stories produce better results when fitted in a power-law function due to the existence of more than two non-zero values as shown in Figure 33.

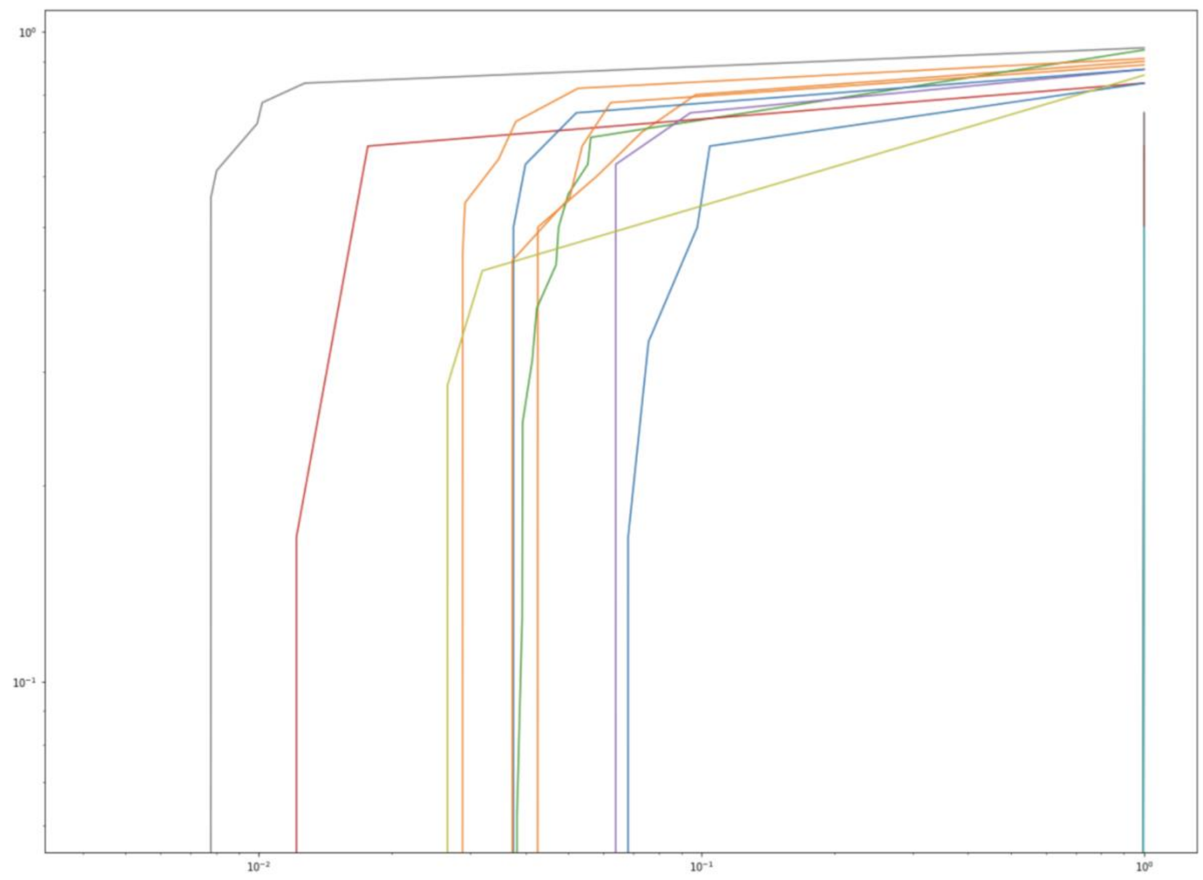


Figure 33 Power-law distribution for fake tweet stories.

Having a way to distinguish fake news from legitimate we can, now, continuously monitor the Twitter Stream API and look for tweet stories that their power-law distribution starts looking like the ones in Figure 33, by computing the SimRank similarity vectors and fitting them to a power-law function.

From the dataset that we have gathered throughout the whole period of our Thesis, we found out that the SimRank algorithm and the vectors produced using the algorithm show promising results on this dataset. The legitimate stories of our dataset had, on average, 87.81% of the edges going to the origin node, while the fake ones had a 61.36% of the edges end up to the origin node.

This means that legitimate stories had a star-like structure, but the fake stories deviated a serious amount from that structure. We had more than one Nodes of Interest (NOIs) in the fake stories, which shows that it would take more than one hop for a random Node in the story graph to reach the origin node. This could show the existence of spam/bot accounts, that are trying to scale out the reach for those types of stories and eventually lead to misinformation.

Chapter 5. Future Improvements

5.1. Data Quality

One of the problems is data quality. Having top quality data is paramount in every research, and that is the case here. Specifically, in our research, the limitations of the Twitter API as well as the timeframe that we had for collecting our data did not help us curate the best dataset.

To further develop this approach, one would need to curate a proper dataset. For this to happen it would take time and it would need hot topics to emerge that could result in fake news spread. Some topics of that kind would be elections in a country that highly impacts the whole world, like the USA, or public concerning events like the recent environmental disaster from the fires in the Amazon rainforest.

Then, it would only take some time to take data from the Twitter Stream API and process those data as explained in Chapter 3.

5.2. Going near real-time

When we talk about real-time stream processing systems, the first approach or iteration if you will always start with a batch-based approach. In that way, you can better work out the problems or limitations that might occur, run some experiments and eventually validate your hypothesis. That's our case as well, so moving forward as a next step we would like to upgrade our pipeline to a real-time one. By having data coming in continually, we can use windows of time and evaluate how information is spread for different posts on an hourly basis, daily basis or even weekly bases. With that information at hand, we can then look for patterns, apply different algorithms and eventually try and flag potential nodes in the graph as bots - all that in real-time, so that anyone reading these posts can give a second thought at the validity of what he or she is reading.

Of course, nothing comes without a cost. A huge consideration here, is the amount of resources that are needed to support such a system, that we will run continually, process large amounts of data in real-time and more. More data systems might need to be integrated like Apache Kafka to store the data, handle retention as well as provide fault-tolerance.

5.3. Graph Analytics

The graph analytics is another part of our approach that can be improved. Here, we chose SimRank Similarity as the algorithm for our computations and then a power-law function to fit the vectors from the SimRank algorithm. Through a more extensive study, we can find different algorithms that can yield better results or use multiple of those algorithms to provide a more secure conjecture regarding the legitimacy of the content in question.

Bibliography

1. Wardle, Claire (February 16, 2017). "Fake news. It's complicated". firstdraftnews.org. Retrieved April 22, 2017.
2. Xinyi Zhou and Reza Zafarani. 2018. Fake News: A Survey of Research, Detection Methods, and Opportunities. ACM Comput. Surv. 1, 1 (December 2018), 40 pages.
3. Antoniadis, S., Litou, I., and Kalogeraki, V. (2015). A Model for Identifying Misinformation in Online Social Networks. 9415:473–482.
4. Perez-Rosas, V., Kleinberg, B., Lefevre, A., and Mihalcea, ´ R. (2017). Automatic Detection of Fake News.
5. Benevenuto, F., Magno, G., Rodrigues, T., and Almeida, V. (2010). Detecting spammers on twitter. Collaboration, electronic messaging, anti-abuse and spam conference (CEAS), 6:12.
6. Elshrif Elmurngi, Abdelouahed Gherbi. Detecting Fake Reviews through Sentiment Analysis Using Machine Learning Technique. https://www.researchgate.net/publication/325973731_Detecting_Fake_Reviews_through_Sentiment_Analysis_Using_Machine_Learning_Techniques
7. Ray Oshikawa, Jing Qian, William Yang Wang. A Survey on Natural Language Processing for Fake News Detection. https://arxiv.org/pdf/1811.00770.pdf
8. SimRank Similarity on Wikipedia https://en.wikipedia.org/wiki/SimRank
9. Jeff Alstott, Ed Bullmore, Dietmar Plenz. Powerlaw: a Python package for analysis of heavy-tailed distributions https://arxiv.org/abs/1305.0215
10. Glen Jeh, Jennifer Widom. SimRank: A Measure of Structural-Context Similarity http://ilpubs.stanford.edu:8090/508/1/2001-41.pdf
11. Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! but at what COST?. In Proceedings of the 15th USENIX conference on Hot Topics in Operating Systems (HOTOS'15), George Candea (Ed.). USENIX Association, Berkeley, CA, USA, 14-14.
12. Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD '10). ACM, New York, NY, USA, 135-146. DOI: https://doi.org/10.1145/1807167.1807184
13. Chengcheng Shao, Giovanni Luca Ciampaglia, Onur Varol, Kaicheng Yang, Alessandro Flammini, and Filippo Menczer. 2018. The spread of low-credibility content by social bots. Indiana University, Bloomington, IN, USA
14. Apache Spark - Unified Analytics Engine for Big Data. Apache Spark - Unified Analytics Engine for Big Data, http://spark.apache.org/.
15. Neo4j Graph Platform – The Leader in Graph Databases. Neo4j Graph Database Platform, https://neo4j.com/.
16. CAPS: Cypher for Apache Spark - Extension of Apache Spark with Cypher Query Language, https://github.com/opencypher/cypher-for-apache-spark
17. The openCypher project aims to deliver a full and open specification of the industry's most widely adopted graph database query language: Cypher. openCypher, http://www.opencypher.org/
18. Actor Model – Wikipedia. https://en.wikipedia.org/wiki/Actor_model
19. Carl Hewitt. Actor Model of Computation: Scalable Robust Information Systems https://arxiv.org/pdf/1008.1459.pdf

20. Lightbend. Scala Programming Language https://www.scala-lang.org/
21. Jetbrains. Kotlin Programming Language https://kotlinlang.org/
22. Python Software Foundation. Python Programming Language https://www.python.org
23. Networkx, Software for complex networks. https://networkx.github.io/
24. Twitter API Documentation. https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/tweet-object.html.
25. Apache Foundation. Apache Hadoop. https://hadoop.apache.org/
26. Amazon Web Services. Amazon Simple Storage Service (S3). https://aws.amazon.com/s3/
27. Google Cloud Platform. Google Cloud Storage. https://cloud.google.com/storage/
28. Google Cloud Platform. https://cloud.google.com
29. Detecting spammers on social networks.
30. https://www.researchgate.net/publication/312181260_Feature_engineering_for_detecting_spammers_on_Twitter_Modelling_and_analysis
31. Current State of the Art to Detect Fake News in Social Media: Global Trendings and Next Challenges. http://www.insticc.org/Primoris/Resources/PaperPdf.ashx?idPaper=71885
32. Automatic Detection of Fake News. https://aclweb.org/anthology/C18-1287
33. Fake News vs Satire: A Dataset and Analysis. http://web.stanford.edu/~mattm401/docs/2018-Golbeck-WebSci-FakeNewsVsSatire.pdf
34. A comparative study of decision tree ID3 and C4.5. https://www.researchgate.net/publication/265162251_A_comparative_study_of_decision_tree_ID3_and_C45
35. CSI: A Hybrid Deep Model for Fake News Detection. https://arxiv.org/pdf/1703.06959.pdf
36. Discretization Techniques: A recent survey. https://pdfs.semanticscholar.org/a5c6/4077637d57aaef1845f84d9f9c282ab96af5.pdf
37. Fake News, Real Consequences: Recruiting Neural Networks for the Fight Against Fake News. https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1174/reports/2761239.pdf
38. Fake News Identification on Twitter with Hybrid CNN and RNN Models. https://arxiv.org/pdf/1806.11316.pdf
39. Fake News Accuracy using Naive Bayes Classifier. https://www.ijrte.org/wp-content/uploads/papers/v8i1C2/A11660581C219.pdf
40. Detection of Maliciously Authored News Articles. https://cooper.edu/sites/default/files/uploads/assets/MirajPatel_Masters_Thesis.pdf
41. Detection of Online Fake News Using N-Gram Analysis and Machine Learning Techniques. https://www.researchgate.net/publication/320300831_Detection_of_Online_Fake_News_Using_N-Gram_Analysis_and_Machine_Learning_Techniques
42. A Rule-Based Classification Algorithm for Uncertain Data. https://www.researchgate.net/publication/220966055_A_Rule-Based_Classification_Algorithm_for_Uncertain_Data
43. The spread of fake news by social bots. https://www.researchgate.net/publication/318671211_The_spread_of_fake_news_by_social_bots
44. Stance Detection in Fake News: A Combined Feature Representation. https://aclweb.org/anthology/W18-5510
45. Towards Automatic Identification of Fake News: Headline-Article Stance Detection with LSTM Attention Models. https://johnsholar.com/pdf/CS224NPaper.pdf

46. A Gradient Tree Boosting based Approach to Rumor Detecting on Sina Weibo. https://arxiv.org/pdf/1806.06326.pdf