Using Storm for Real-Time First Story Detection

Michail Vogiatzis

Master of Science

School of Informatics

University of Edinburgh

2012

Abstract

Twitter has been an excellent tool for extracting information in real-time, such as news and events. This dissertation uses Twitter to address the problem of real-time new events detection on the Storm distributed platform such that the system benefits from the scalability, efficiency and robustness this framework can offer. Towards this direction, three different implementations have been deployed, each of which having a different configuration. The first and simplest distributed implementation was the baseline approach. Two implementations followed, in an attempt to achieve faster data processing without loss in accuracy. The rest two implementations demonstrated significant improvements in both performance and scalability. Specifically, they achieved a 1357.16 % and 1213.15 % speed-up over the single-threaded baseline version, correspondingly. Moreover, the accuracy and robustness of the scalable approaches comparing to the baseline version were retained.

Acknowledgements

I would like to express my gratitude towards my supervisor Miles Osborne for being an outstanding advisor and a great intellectual support. His guidance and his humour made this dissertation not only possible but also a rewarding enjoyable journey.

I would also like to thank Sasa Petrovic for the valuable insights he gave me throughout the implementation of this work.

I am obliged to Storm project leader Nathan Marz and further Storm users for their helpful and practical comments through the on-line Storm user group.

Last but not least, I cannot but express my sincere appreciation to my friend Yannis Konstas whose constant encouragement and influential discussions added a remarkable value to this work.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Michail Vogiatzis*)

Table of Contents

Chapter 1

Introduction

Twitter is one of the most widely spread social media in recent years. People can publish short text-messages which can be viewed by their followers. The usability and ease of using this tool contributed to its wide growth. It can be claimed, that although Twitter began as a social media application, users nowadays take advantage of its news media aspect. Every day, thousands of posts share information about news, events, automatic updates (weather, songs) and personal information.

The information published can be retrieved and analyzed in a news detection approach. The immediate spread of events in Twitter combined with the large number of Twitter users prove it suitable for first stories extraction. Towards this direction, this project deals with a distributed real-time first story detection (FSD) using Twitter. Specifically, we try to identify the first document in a stream of documents, which discusses about a specific event. Furthermore, this is implemented in the *Storm* platform. Storm is a distributed platform which provides an abstract way of dealing with underlying hardware and low level network configurations and allows the development of real-time applications on a distributed system.

Storm applications benefit from the scalability of its abstraction layer and also present high fault-tolerance. This leads to both efficient and high throughput rate deployments that utilize the compute capacity provided by the network nodes. The real-time computation needs of this project has defined Storm as the most suitable development platform against its competitors.

While the new events detection task has been previously addressed, it is not real-time and cannot

run in a distributed fashion. Using Storm to deal with the FSD task should retain any previous accuracy in such a way that first stories are successfully spotted. Storm applications require the design of multiple logical steps that are handled by different machines as well as the manual configuration of the load distribution across the cluster.

All above problems have been addressed by this work. At the beginning, the FSD task was implemented on Storm platform. Three different versions were developed, which mainly differ in the design and the resources required to run efficiently. The first, *Baseline Topology* (BLT) has been deployed on Storm and is equivalent to the sequential version running on a single processor. The second, called *Ad-Hoc Topology* (AHT) refers to a distributed FSD implementation which benefits from limited available resources. *Maximum Scalability Topology* (MST) is the third implemented version, which aims to benefit from a higher number of nodes available and thus presents worse performance than AHT if limited resources are available. Comparing to the single-threaded baseline version, AHT presented a speed-up of 1357.16 % whereas MST that of 1213.15 % on a 100,000 tweets data input. AHT and MST presented a speed-up of 365.12 % and 326.37 % correspondingly, when the multi-threaded BLT was used. An important aspect which affects negatively the time performance was found to be the overhead from message communication between machines. Storm serializes data in order to send or receive them, a process which adds significant cost to the overall performance. In the final analysis, our main contributions included three robust Storm versions two of which can scale up easily when additional resources are added to the system. Additionally, all of the Storm implementations presented high fault tolerance.

Moreover, the project contributes in finding the bottlenecks inside each topology. Changing the allocated resources of different areas inside each program has shown that the most demanding parts are those that include the computation of the dot product function which demands the highest compute capacity. Finally, the work demonstrated a trivial deviation in the accuracy of the AHT and MST results comparing to those of the baseline implementation. In terms of mean absolute error, AHT presented an error of 0.054 whereas MST a value of 0.055. A minimal deviation is however acceptable as the FSD task uses a randomized algorithm to compute the final results.

The remaining chapters of this dissertation are structured as follows. Chapter 2 provides a detailed description of the existing background work in the area of FSD. It also provides a rationale

for selecting Storm as the primary development platform. The dissertation continues with Chapter 3, which presents an extensive explanation of the Storm processing system. Chapter 4 details the methodology followed by this study, as well as the metrics used to evaluate the experiments. Moreover, the experiments on different Storm configurations along with their results are demonstrated in Chapter 5. This Chapter also presents the FSD algorithm used in all experiments and the results discussion. Finally, Chapter 6 concludes this work and outlines areas for future improvements in the field of distributed real-time FSD.

Chapter 2

Background

This section aims to present the background work on the problem addressed by this project. Recent work in the field of first story detection is discussed and a comparison between different approaches is provided. The chapter concludes with the motivation for pursuing this research and argues about the selection of the most appropriate distributed real-time platform to work on.

2.1 Twitter

The micro-blogging service Twitter is used by millions of people, as a social network to broadcast news and connect with friends. Over 300 million of text-based posts are generated every day including significant events, global breaking news and daily communication (personal life, spam). These are known as *tweets* and have a maximum length of 140 characters. Tweets can be posted in any language and they are easily created by anyone registered.

Since messages have short length, URLs are allowed to provide additional information. Additionally, keywords can be used to add special functionality on the messages. Twitter users add hash (#) tagged words to provide a topic labeling on each tweet. Then similar tweets can be easily found by asking queries over the Twitter search engine. It is worth mentioning that Twitter handles approximately 1.6 billion search queries per day[1]. Also, the at-sign (@) is a special

[1] Retrieved from `http://engineering.Twitter.com/2011/05/engineering-behind-twitters-new-search.html` on 05 May 2012.

symbol which refers to a personal account or to a reply.

Twitter has become very popular lately and certainly contributes to the *Big Data* [Jacobs, 2009] era. Such a huge load of information defines it as an excellent source of events which can be discovered and analyzed. As a result, conclusions can be derived about aspects of social behavior. Specifically, we can detect the significance of the events and the way that people reacted to them.

2.2 First Story Detection

As Twitter is an excellent source of information, we can detect bursts of tweets that refer to a particular topic. Given a series of documents[2], first story is defined as the first document to discuss a specific event, which occurred at a particular time and place. First story detection (FSD) was firstly defined by [Allan, 2002] in terms of topic detection and tracking. Representative first story examples can be the 8.8 magnitude earthquake which occurred in Chile on 27 February 2010 or a death of a famous person such as *Michael Jackson* on 25th June 2009. Events such as these are spread widely over Twitter showing the noticeable impact they have on the real world.

The rise in mobile devices usage along with the numerous applications available for Twitter [3] allow users to publish updates easily and frequently. Due to the voluminous amount of incoming information, the FSD task cannot be performed manually by humans. Therefore, systems have been implemented in order to cope against this explosive amount of load. Regarding Twitter, these systems face problems such as dealing with mostly irrelevant information - *spam* - and very high input rates. In our context, we define *spam* as a tweet that refers to advertisements, automatically generated messages such as weather updates etc. A description of relevant systems and a comparison between them is provided on the following section.

2.2.1 Related Work

New event detection has attracted much interest in the recent years. This section aims to describe and analyze how previous systems address the first story detection task and provides a

[2]For the rest of this dissertation, the terms documents and tweets will be used interchangeably.

[3]Retrieved from `http://techcrunch.com/2009/02/19/the-top-20-Twitter-applications/` on 23 July 2012.

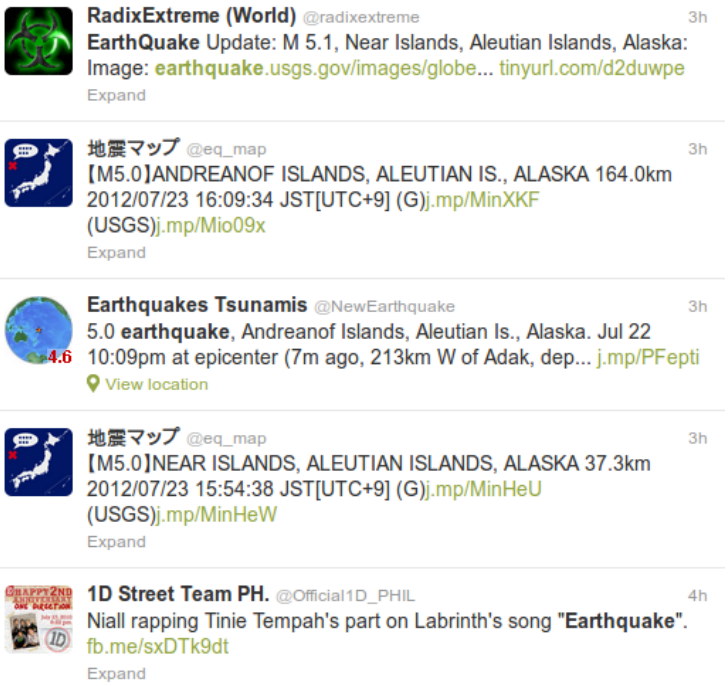

Figure 2.1: Sample of tweets speaking about an earthquake which occurred on 23/07/2012 near Aleutian Islands, Alaska.

comparison between them.

Traditional approach: Initial approaches to the FSD task have been made by [Allan et al., 2000] (UMass) and [Yang et al., 1998] (CMU system). They both use a similar approach which will be called the *traditional approach* from now on. More specifically, documents are represented using the vector space model by splitting each document into IDF (Inverted Document Frequency) - weights. Given a term t and a total number of documents D:

$$\text{idf}(t,D) = log\frac{D}{\{d \in D : t \in d\}} \tag{2.1}$$

where $\{d \in D : t \in d\}$ is the total number of documents that the term t appears in. TF (Term Frequency) - IDF weighting aims to produce a representative weight for the terms found in each document depending on the number of times this term occurs. Thus, words that appear more frequently are expected to have lower weight in contrast to the rarer ones. A nearest-neighbor search follows the TF-IDF representation, by comparing each tweet to all tweets previously seen. The nearest neighbor is defined as the tweet with the closest distance. For the rest of this study, we will be using the word distance to refer to the comparison between two tweets. After all

comparisons have been made, the closest distance is compared to a given threshold. If it is above that value, the document is defined as a first story. It is also worth mentioning that in order to reduce the computational complexity, document vectors may only keep words with the highest weight. The algorithm followed by the traditional approach is described in Algorithm 1.

Algorithm 1: Traditional approach to FSD

Data: corpus

```
1 foreach document d in corpus do
2     foreach term t in d do
3         foreach previously seen document d' that contains t do
4             update distance(d, d')
5         end
6     end
7     dis_min(d) = min_d' {distance(d, d')}
8     add d to inverted index
9 end
```

However, for large datasets this method is ineffective in terms of computational complexity. In case of unbounded input, e.g. a Twitter streaming system, the comparisons would grow proportionally as the incoming data increase. Consequently, assumptions on storing the data in main memory are no longer valid. As a result, following the traditional approach on a streaming system is certainly infeasible.

This leads to an approach firstly defined by [Indyk and Motwani, 1998], called *locality sensitive hashing* (LSH) . LSH method aims to benefit from a reduced number of comparisons required to find the approximate $(1+\varepsilon)$-near neighbors among a number of documents. The idea behind it is to use hash-tables called buckets for similar points (tweets). According to this approach, whenever a new point (tweet) arrives, its hash value will be computed and it will be stored into several buckets. Inside the same bucket, the probability of collision with similar documents is much higher. In other words, tweets that have an identical hash with the arriving tweet are nearest neighbour candidates. Figure 2.2 helps to understand the core LSH mechanism.

There are numerous ways to construct a family of hash functions [Paulevé et al., 2010]. An interesting approach is to use the random projection method which is based on the cosine similarity

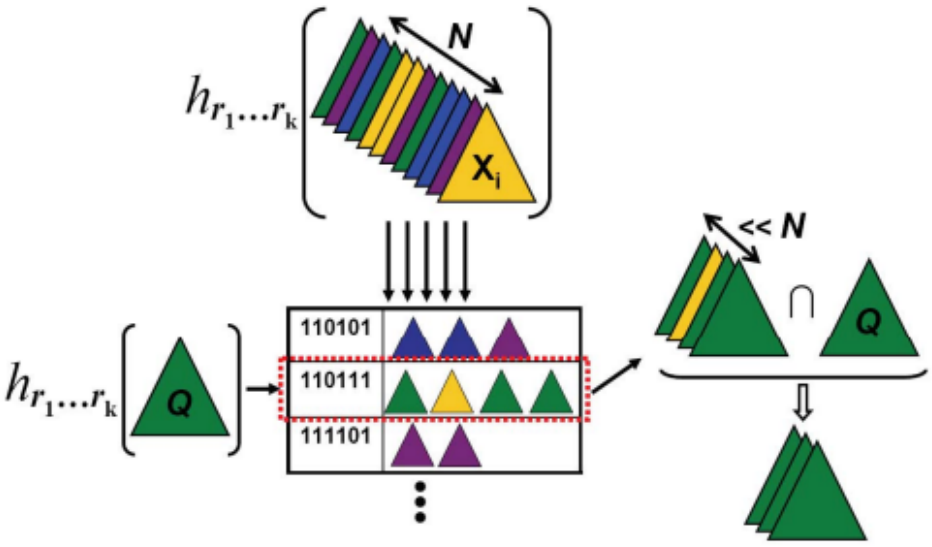

Figure 2.2: LSH [Grauman and Darrell, 2007] - Probability of collisions is much higher for similar documents

measure. The cosine similarity between two vectors $\vec{u}$ and $\vec{v}$ is defined as

$$cos(\theta) = \frac{u \cdot v}{\|u\| \cdot \|v\|} = \frac{\sum_{k=1}^{n} u_i \times v_i}{\sqrt{\sum_{k=1}^{n} (u_i)^2} \times \sqrt{\sum_{k=1}^{n} (v_i)^2}} \tag{2.2}$$

According to [Charikar, 2002], constructing random hyperplanes will give a family of hash functions such that the probability of collision between two vectors is proportional to the cosine of the angle between them. Specifically, the hash bit is computed by the following function:

$$h_{\vec{r}}(\vec{u}) = \begin{cases} 1 & \text{if } \vec{r} \cdot \vec{u} \geq 0, \\ 0 & \text{if } \vec{r} \cdot \vec{u} < 0. \end{cases} \tag{2.3}$$

Then given a family of hash functions F and two vectors $\vec{u}$, $\vec{v}$

$$\mathbf{Pr}_{h \in F}[h(\vec{u}) = h(\vec{v})] = 1 - \frac{\theta(\vec{u}, \vec{v})}{\pi} \tag{2.4}$$

The interest lies in the fact that while hash function produces a single bit, we can concatenate n bits to construct an n-bit vector in order to reduce the probability of collisions with documents that are not close enough.

Applying the LSH algorithm on the first story detection context means that documents in the same bucket are very likely to refer to the same subject. Therefore, the number of comparisons to

find the nearest neighbour is significantly decreased and [Petrović et al., 2010] have shown it can be limited to a constant number of 3L, where L is the number of hash-tables used. Specifically, the colliding tweets are sorted according to the frequency they collide, and the tweet is compared only to the top 3L documents it collides more frequently with. The true advantage of the LSH algorithm is that searching requires $O(n \cdot (\frac{1}{1+\varepsilon}))$ comparisons comparing to the linear search that the traditional approach achieves.

It is not hard to conclude that in a system which receives large amount of data, memory and computational power problems will arise. Therefore a different approach is required. In order to deal with such problems and to process tweets in real-time, streaming approaches have been recently implemented [Petrović et al., 2010][Muthukrishnan, 2005]. While documents are processed in chronological order, decisions have to be made in bounded space and time.

Recent work of [Petrović et al., 2010] performs a first story detection on streaming data with application on Twitter. Apart from LSH, they have proposed a *variance reduction* strategy. This is required since LSH fails to find the true neighbor if the nearest point lies in high divergence. A variance reduction strategy imposes that the query is compared with only a given number of previous points. Furthermore, they compare the new tweet with a fixed number of most recently seen ones. If the new minimum distance is lower than the one found from LSH, then this distance is assigned to the tweet. In a later system, [Petrovic et al., 2012] have improved the system's FSD performance, using paraphrases to address the problem of lexical variation in different documents. In terms of speed, [Petrović et al., 2010] significantly outperforms UMass system on the TDT5 dataset[4]. The UMass system presents linear growth on processing time as the data increase, whereas [Petrović et al., 2010] system presents constant time. As previously mentioned, the traditional approach would never terminate in reasonable time. As a result, trying to compare the two systems using large-scale Twitter data would be impossible.

Another approach to new event detection has been presented by [Luo et al., 2007] in an attempt to address efficiency issues on limited resources. They have followed a traditional approach that detects first stories in real-time by applying indexing and compression mechanisms to the incoming documents. In addition, to make the system capable of handling the high rate of the incoming documents, a number of limitations have been applied. Although the system has been evaluated using the standard TDT5 dataset, it has not been tested on a large scale basis.

[4] `http://www.itl.nist.gov/iad/mig/tests/tdt/2004/workshop.html`

In the final analysis, all of the previous approaches focused more on the accuracy they achieved rather than the scalability and throughput rate of the system. Although building a system that successfully process Twitter posts and spots first stories is an essential feature, it is not our primary focus in this study. Nevertheless, there is a high motivation to achieve scalability and speed and therefore we assume that a distributed approach needs to be applied. Following this hypothesis, this project aims to benefit from the capabilities of a distributed processing system such as the Storm [5]. We consider such an environment to be ideal for the first story detection task which uses an incredible amount of input data (millions of tweets) and demands high computational power to process them fast.

2.3 Storm

Storm is a distributed real-time computation system which can guarantee data processing, high fault-tolerance and horizontal scaling to significantly large amounts of input data. It simplifies the implementation of parallel tasks by providing a programming interface suitable for stream processing and continuous computation. Storm became an open-source project on 19th September 2011 and it consists of half Java and half Clojure source code. Many companies use storm including Twitter[6], Groupon[7] and Alibaba[8].

Similarly, S4 [Neumeyer et al., 2010] can be described as distributed processing system which spreads computation over a network of machines. It is a decentralized, distributed, scalable and partially fault-tolerant platform written in Java, which can process unbounded data streams as an input. Yahoo! Inc. was the first that used it to process a large number of search queries in a few seconds. It is now open-sourced and can be used under the Open Source Apache 2.0 license.

These two platforms may share common purposes but differ at many points. Comparing to S4, Storm was released after S4 but has an active and very fast growing community. Many inquiries can be answered using Storm's on-line user group[9]. Concerning the configuration, S4 machine topology is defined through a complex XML file in contrast to the easier in-code definition of

[5] `http://storm-project.net/`
[6] `https://Twitter.com/`
[7] `http://www.groupon.co.uk/`
[8] `http://www.alibaba.com/`
[9] `http://groups.google.com/group/storm-user/`

Storm system. In addition, Storm offers a suitable interface for cluster monitoring managed by a master node daemon called *Nimbus*. Storm system is characterized by the fact that its users have more control and thus responsibility for the choices they make. Since Storm's load balancing is left to the user, this is a critical development task which the developer has to consider. Although S4 project is more mature than Storm which is still under development, Storm possesses better reliability capabilities by guaranteeing that all messages will be fully processed. S4 on the other hand offers less fault tolerance and can lose data through communication between machines. As opposed to traditional approaches, none of these platforms use local storage extensively. All things considered, Storm was selected to be the main system on this research work and S4 was considered as an alternative solution.

As shown by this chapter, the FSD task has been addressed in recent years. There is yet high motivation for development of such a solution towards scalability, efficiency and robustness. It is believed that there is no previous work we are aware of that uses a distributed real-time processing system to spot first stories on Twitter.

Chapter 3

Storm Processing System

Having examined the FSD background as well as a few concepts of S4 and Storm distributed streaming platforms, this chapter aims to provide a deep understanding of Storm's fundamental components. Aspects such as streams, topologies, message processing, task distribution, scalability and robustness are covered below. The chapter also gives an overview of how high fault-tolerance is achieved on the Storm cluster and concludes with a description of the distributed RPC Storm use case.

3.1 Definition

Storm is a distributed real-time framework suitable for stream processing. Similarly to how Hadoop [White, 2009] performs batch processing, Storm is designed to perform real-time computations. A number of key properties prove its suitability for implementing the FSD task on a distributed system.

First of all, it provides robustness and high fault tolerance. It offers an environment that applications can easily scale up when more resources are available. Furthermore, Storm offers multi-language capabilities, therefore developers are capable of developing applications in any programming language of their preference. Additionally, parts of the same system can be written in different programming languages and still be able to communicate with each other. Also, recent integration with Mesos cluster resource manager [Hindman et al., 2011] allows resource

isolation between topologies. In particular, it is possible to configure CPU and memory allocations per topology or even share resources with various computing frameworks such as Hadoop and Spark [Zaharia et al., 2010].

3.2 Purpose

The way real-time computation used to work before Storm was by implementing a network of queues and workers connected to Hadoop and Cassandra, a distributed database management system [1]. In case that more workers need to be added to the system, a series of re-configurations and re-deployments need to apply such that the cascading dependencies are satisfied. Additionally, batch processing updates imply high latencies. On the whole, scaling was a difficult task, fault-tolerance was poor and the development was laborious.

Storm aims to overcome the above problems by guaranteeing that data are fully processed, horizontal scalability can be easily achieved and a high level of abstraction is provided to the developer. Therefore, more time can be spent on actual programming rather than on the system's configuration phase. Moreover, speaking of a real-time system, high latencies and downtimes are no longer allowed. The implementation of Storm system successfully covers these requirements and offers a suitable environment for a number of use cases. The first and most regular use case is the real-time stream processing in which streams are processed and an action is performed on the passing messages i.e. updating databases. The second one is the distributed remote procedure call (RPC). According to this, Storm can be used to compute a complex function on-the-fly, that is really intense to compute on one machine. For this reason, the function is parallelized on top of Storm which acts as an RPC mechanism. See section 3.6 for a detailed description of this particular use case. Last but not least, continuous computation represents the case in which a continuous query provides results that should be known at any time i.e. computing trending topics on Twitter.

[1] `http://cassandra.apache.org/`

3.3 Main Concepts

This section describes the essential Storm components that allow application configuration and development. These components provide Storm an abstraction layer suitable for development regardless of the underlying hardware.

3.3.1 Streams

Storm is able to manipulate infinite streams of data. A *stream* is an abstraction of Storm which consists of an unbounded sequence of tuples. Tuples emission is the message passing mechanism of Storm and contain a sequence of values of any type. The default values supported by Storm are strings, integers, longs, shorts, booleans, bytes, doubles, floats and byte arrays integers. As for custom objects, these can be passed as tuple values by using custom serialization mechanisms which are easy to implement.

3.3.2 Spouts

Streams are emitted in the topology using *spouts*. Spouts are responsible for reading data from external sources and pass them as input to the application for further processing. An interest example is that a spout can be connected to the Twitter Streaming API[2] and read tweets in real-time. Spouts can be implemented in such a way that emitting tuples are replayed if Storm processing units fail to receive them. This is part of Storm's reliability API which is further analyzed in section 3.5. Finally, spouts are capable of emitting multiple streams if declared in the configuration.

3.3.3 Bolts

Bolts are the logical units of the application which are able to process input streams and then produce new output streams to subsequent application steps. Moreover, bolts can perform operations such as running functions, filtering, streaming aggregations, streaming joins and updating

[2] `https://stream.twitter.com/`

databases. As done by spouts, bolts can emit multiple streams. Although the bolts are the smallest logical processing unit, the developer is allowed to run asynchronous threads inside each bolt without worrying about thread concurrency issues.

3.3.4 Topologies

Each application must define a network of spouts and bolts before it starts running. This network is called a *topology* and contains the whole application logic. As MapReduce runs *jobs* [Dean and Ghemawat, 2004], Storm runs corresponding topologies. One major difference between those two is that a MapReduce job eventually finishes, whereas a Storm topology needs explicit termination to stop running.

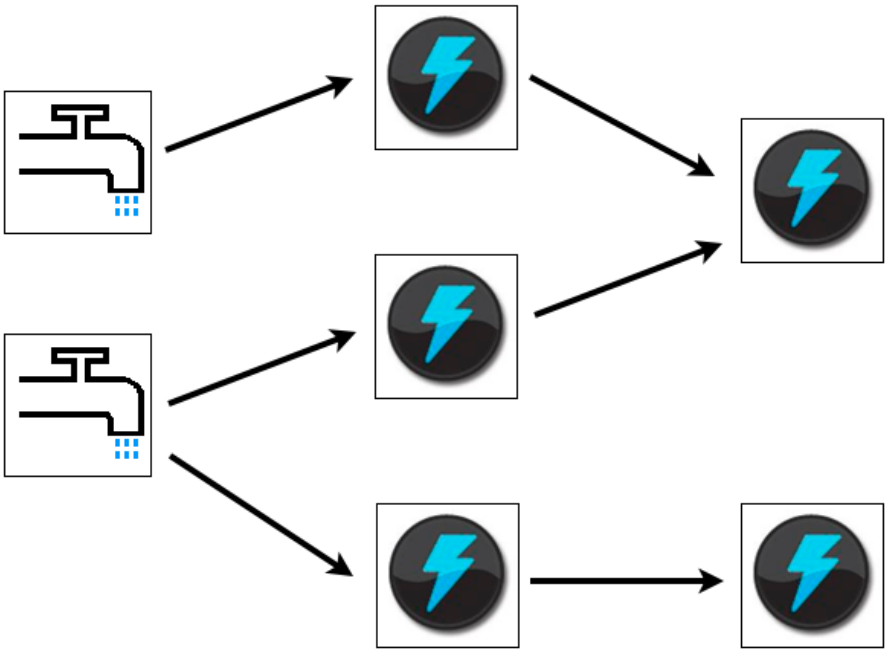

Figure 3.1: Topology example.[3]

As seen in Figure 3.1, every edge in the network indicates a bolt subscribing to the output stream of another spout or bolt. In a Storm topology, bolts and spouts run in parallel and *tasks* are responsible for the level of parallelism in each component. Tasks in different bolts can communicate between each other using the tuple message passing mechanism. Concerning the level of

[3]Retrieved from `http://storm-project.net/` on 31 July 2012.

parallelism of each component, this will be adjustable on-the-fly using the *rebalance* command of the upcoming Storm version (0.8.0).

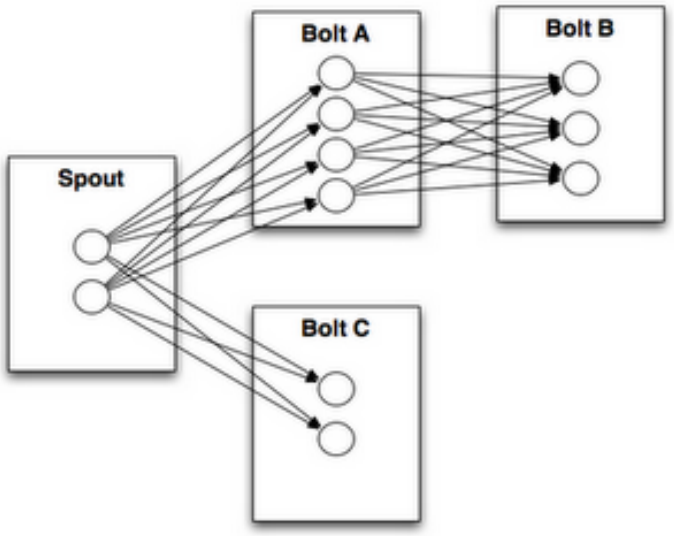

Figure 3.2: Tasks communication between components [4]

Tasks distribution to worker processes is performed by the master node daemon *Nimbus*. Nimbus also performs other operations such as code distribution into the cluster and failure monitoring. A machine may be responsible for more than one tasks in different components. This allows the workload to be evenly distributed across the cluster. The coordination between Nimbus and the worker nodes is performed by Zookeeper [Hunt et al., 2010], a service suitable for cluster coordination. Each worker machine runs a *Supervisor* service which starts and stops worker processes according to Nimbus commands.

Finally, topologies can be terminated, otherwise they run forever [5]. Nonetheless, terminating a topology does not instantly kill the topology, but prevents spouts from emitting new messages. It also awaits before workers emit any tuples they were processing at the time they received the kill command. The number of seconds that Storm awaits can be defined in the topology configuration[6].

[4] Retrieved from `https://github.com/nathanmarz/storm/` on 31 July 2012.
[5] A topology can be killed using the command: *storm kill topologyname*
[6] Config.TOPOLOGY_MESSAGE_TIMEOUT_SECS property is used to define the awaiting time.

3.4 Communication between Components

Storm provides various stream groupings for communication between two components. If the default stream groupings do not satisfy the needs, custom stream groupings can be implemented using the *CustomStreamGrouping* interface. Unlike MapReduce, Storm provides numerous different groupings which are listed below.

- **Shuffle Grouping:** This is the simplest grouping provided by Storm, where tasks randomly select the destination of the emitting tuple. A random round robin algorithm is used to randomize the order of the tasks. Moreover, Storm guarantees that tuples are evenly distributed across the tasks.
- **Fields Grouping:** This is similar to a MapReduce grouping, where keys emitted from different Mappers arrive sorted to the Reducers. Likewise, Storm performs a mod hashing on a subset of a tuple fields such that these tuples are guaranteed to arrive at the same task. As an illustration, in a word count example, same words emitted from different tasks will end up at the same task.
- **All Grouping:** The tuple is replicated to all tasks.
- **Global Grouping:** This kind of grouping ensures that all tasks will send the tuple to the task with the lowest id. As a result, all tuples go to the same consumer task.
- **None Grouping:** At the most recent stable version of Storm (0.7.4), this grouping is identical to shuffle grouping. Nevertheless, future releases will benefit from using this kind of grouping, as Storm will try to direct tuples to be processed by the same thread as the component they subscribe from.
- **Direct Grouping:** According to this particular type of grouping, each task emits tuples directly to a specific consumer task.
- **Local / Shuffle Grouping:** Tuples here are emitted to tasks that are located in the same worker process if possible. If not, the grouping is similar to shuffle grouping.

3.5 Fault Tolerance

Storm offers reliability capabilities to ensure that all tuples will be processed by the consumers and guarantees that the system will continue to operate normally in the presence of failures. The guaranteed message processing capability is an exceptional advantage against similar systems which operate in "lossy" behavior.

As mentioned in section 3.3.4, Zookeeper uses a set of nodes responsible for the coordination between Nimbus master node and the supervisors. This intermediate coordination service allows Storm system to be extremely robust. Since Nimbus and supervisors do not maintain state, they can fail and recover with no loss of data. As a reminder, supervisors are daemon services that run on each machine and run or stop worker processes according to Nimbus orders. To put it differently, Storm ensures that even if a node fails the application will continue to operate normally.

The message processing guarantee is achieved using Storm's reliability API. To begin with, at the time a spout emits a tuple, it also assigns a unique id. As the tuple moves to the next set of components, it triggers bolts which emit new tuples that depend on the original root tuple. The new tuples similarly trigger the production of other new tuples and eventually a *tuple tree* of dependencies is built. In order for Storm to replay messages that are not processed by every node in the tuple tree, Storm provides the *anchoring* mechanism. According to this, the developer should notify Storm about creating a new edge in the tuple tree by passing the tuple it processes as an argument along with the new tuple values. Also, the developer has to notify Storm when it has fully processed the tuple by calling the *ack* method. The whole "acking" procedure can be automatically performed using an instance of the BaseComponent class in Java.

Storm can now replay any messages that should but were not processed by every node in the tuple tree within a specified amount of time. In case of failure, the *fail* method forces a message replay from the initial spout. Otherwise, if the tuple is successfully processed then the *ack* method will be called on the spout, specifying the message id that was initially assigned to it. The reliable mechanism is optional and the developer may deactivate it if necessary. Then a spout will always consider that the message has been fully processed at the time it emits it.

Even if all tuple trees need to be tracked, Storm tracks them in an breakthrough way. A fully distributed key algorithm ensures that each tuple tree will take approximately 20 bytes of mem-

ory regardless of how many tuples are pending in the tree. This prevents from facing memory problems in case that millions of tuples are active in the topology.

Following the above procedure, Storm provides at-least-once semantics to the application. It seems that Storm would end up with wrong results in case that failures occur and operations are not *idempotent*. Idempotent is the operation that can be invoked more than once, without changing the initial result. This problem however has been solved thanks to the transactional topologies that Storm offers. Transactional topologies are provided as a higher level of abstraction and can achieve exactly-once semantics. Nevertheless, their design details are out of the scope of this document and will not be analyzed further.

3.6 Distributed RPC

Distributed remote procedure call (DRPC) is a way to use Storm in order to benefit from its scalability and speed. A complex function which is really intense to compute on a single machine can be computed using a Storm topology on-the-fly. The usefulness of such a feature in Storm leaded Storm developers to embed it as a main functionality. This is the main environment that our FSD task runs into and its logic is described in this section.

DRPC uses a DRPC Server to handle requests which is already implemented in Storm. As shown in Figure 3.3, the client passes a function and a number of arguments to the DRPC Server and the latter acts as the source for a spout for the topology. Then the DRPC Server emits a stream of function indications which is a simple tuple. Each tuple contains a request id, the actual arguments given by the client and the information of the host and port that the results will be returned to.

The topology reads the stream coming from the spout, computes the intense function and emits the results to a special bolt along with the request id. The bolt then connects to the DRPC Server and sends the result for this particular request. Finally, the DRPC Server sends the results to the host and port defined previously and unblocks the waiting client. A powerful feature of DRPC is that it allows the implementation of a particular type of bolt interface called *IBatchBolt*. This allows tuples to be emitted in batches and it has the ability to wait before a tuple is fully

[7]Retrieved from `https://github.com/nathanmarz/storm/` on 31 July 2012.

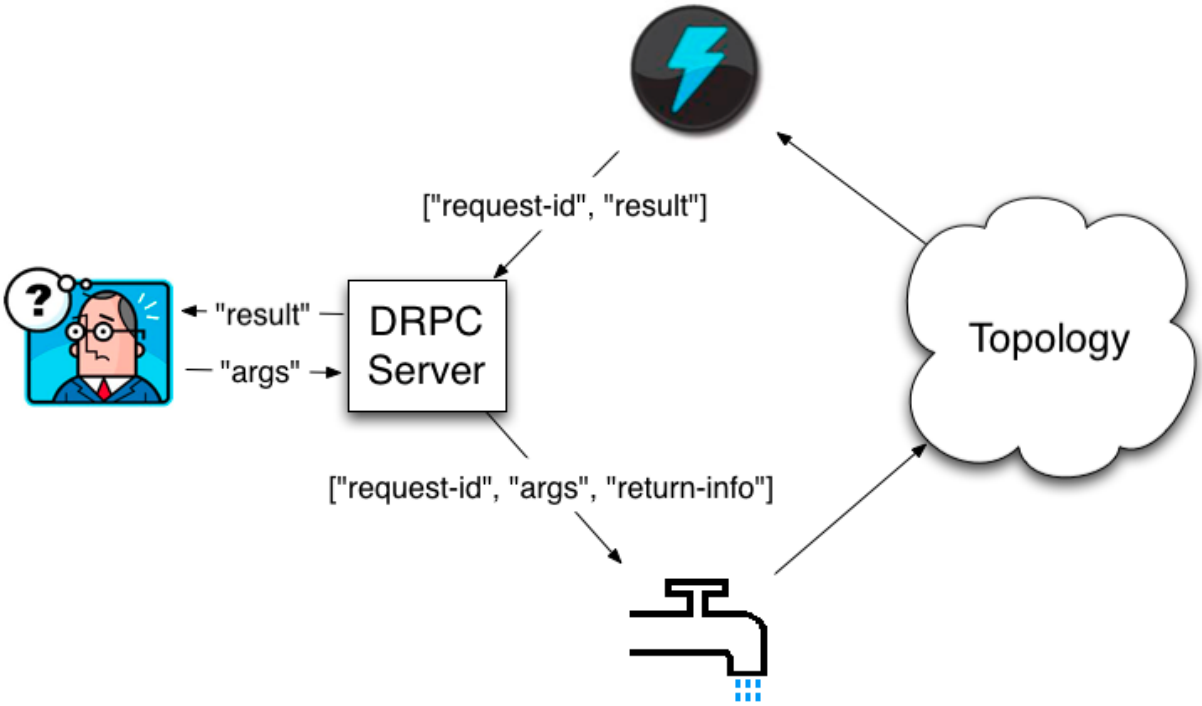

Figure 3.3: Distributed RPC data flow [7]

processed by all tasks inside a bolt before emission. This is crucial to our FSD task application where incoming tweets should be processed strictly by the order they arrive.

Chapter 3 has demonstrated the main concepts of Storm as well as the purpose of using such a system to solve the FSD problem. As previously described, Storm's abstraction level provides an efficient way of developing easily scalable applications that are also robust. It is believed that it is the most suitable environment for real-time streaming processing to date which allows ad-hoc analysis with low latency and minimal maintenance.

Chapter 4

Evaluation - Methodology

This chapter aims to provide the methodology followed by this study and the questions that it is called to answer. Three different metrics will be described which evaluated the experiments listed in chapter 5.

4.1 Setup

All experiments followed by this work were conducted on the same machine of which specifications are listed in Table 4.1.

The input used for this project was the dataset used by [Petrovic et al., 2012]. It consists of 51 million tweets crawled from 30 June 2011 to 15 Sept 2011 from the Twitter streaming API[1]. The corpus format is in JSON[2] where each tweet is accompanied with various meta-data. A field example is the *in_reply_to_status_id* field which contains the original tweet ID that this tweet replies to, if any. Additional meta-data include the number of times this tweet was re-tweeted, the user who posted the tweet, the geographic location as published by the user's application etc. For the purpose of this project, only three fields were used per tweet as shown in Table 4.2.

From the dataset used, two evaluation sets were derived. The first evaluation set is called TWT100 and consists of 100,000 tweets. The second one is called TWT500 and contains

[1] https://stream.twitter.com/
[2] http://www.json.org/

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
CPU(s):	24
On-line CPU(s) list:	0-23
Thread(s) per core:	2
Core(s) per socket:	6
Socket(s):	2
CPU MHz:	2261.010
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	12288K

Table 4.1: Machine specifications

text	String	The tweet text
id_str	String	The unique representation of this tweet
created_at	String	The UTC time the tweet was created

Table 4.2: Tweet fields used

500,000 tweets. TWT100 was used considering that the simplest topology can finish in reasonable time. TWT500 was the one used by the two Storm versions, excluding the Storm baseline version.

Two interesting research questions are called to be answered. Regarding the efficiency of the implementations, an interesting question would be: "Which is the fastest topology to run my FSD task?" To answer to this question we will proceed to the strategies described in section 4.2. The second is to measure how accurate each topology is, by defining an accuracy metric as described in section 4.3.

4.2 Workload

The FSD task has numerous ways to be run on Storm. Storm components can split and merge such that the topologies differ both in speed and resources needed. In an attempt to answer to the efficiency question, the following methods are used. The first changes the topology's structure, and the second finds the optimal parallelization distribution inside each topology. All time performance comparisons between topologies used the *speed-up* metric in percentage. Given two topologies and their execution times T_1, T_2, the speed-up S of the second topology over the first, is defined as:

$$S = \frac{T_1}{T_2} \tag{4.1}$$

4.2.1 Changing the Topology Structure

The FSD task will be tested on different topologies in order to compare how fast the same workload is consumed by each topology. Three different topologies will be used for this purpose. The first topology, called the *Baseline Topology* (BLT) represents the simplest implementation of an FSD system on Storm. It consists of a single bolt which contains the whole locality sensitive hashing logic. The second implementation, *Ad-Hoc Topology* (AHT) aims to cover the case of limited resources, without losing the benefit of a distributed system's advantages. The most demanding approach, both in development and in resources needed, is the *Maximum Scalability Topology* (MST). As the name denotes, it is the most scalable approach which demands a bigger number of nodes to be efficient. The three configurations are further described in section 5.2.

4.2.2 Load balancing

In order to find the fastest configuration in each topology, a number of experiments will be conducted. As previously mentioned, bolts consist of tasks that define its parallelism. Each task is handled by a different machine, although the same machine can handle more than one tasks inside the same component. By changing the number of tasks inside a bolt, the level of its parallelism is changed. Consequently, we can boost the points where the most demanding work is carried out and limit the resources of other areas. As a result, we can achieve an overall

greater benefit. In other words, this strategy aims to find the bottlenecks in each topology and thus provide the optimal solution for each.

4.3 Accuracy Metric

The purpose of this metric will be to estimate the most accurate topology. The BLT will be used as the gold standard since it provides the same results as the sequential version of the program, running on a single machine. The *Mean Absolute Error* (MAE) will be used as a measure of the deviation between the results of the systems. Given a prediction pair $<p_i, q_i>$MAE is defined as:

$$MAE = \frac{\sum_{k=1}^{N} |p_i - q_i|}{N} \tag{4.2}$$

We expect the systems to show a minimal deviation because they run an identical task on the same data. However, since LSH is a randomized algorithm, the MAE will not be zero.

In conclusion, the chapter enlisted three methods which will be used to evaluate the FSD task on Storm. The first is concerned with changes in the topology design in an attempt to find the most efficient design for specific machine specifications. The second focuses on identifying the bottlenecks inside a topology, and finally the third metric measures the accuracy of the results comparing to the baseline implementation based on the mean absolute error.

Chapter 5

Experiments

The Experiments chapter begins with a detailed description of the FSD algorithm. The chapter continues with the experiments carried out on different topologies that implement the same FSD task. Results are also presented and analyzed in depth such that the research questions of chapter 4 are fully covered.

5.1 FSD Implementation

Implementing the FSD system on Storm requires clear understanding of the problem addressed such that the algorithm can be split into logical steps of which the topology will be consisted. For this reason, a first story detection system similar to [Petrović et al., 2010] was initially developed. The algorithm was implemented in single and multi-threaded versions on a single machine. Due to thread construction and destruction operations, no speed-up has been detected when using more than four threads. However, this is an implementation-specific observation rather than a general rule. The FSD algorithm used by all three Storm topologies is described below in brief.

Upon a new tweet arrival, the short text is split into words such that a partial counting can be applied. In an attempt to reduce the lexical variation that occurs in different documents, a small normalization set of *out-of-vocabulary* (OOV) words has been used [Creutz et al., 2007]. Words like “coz” are replaced with “because” and “sureeeeeeeeeeeeee” with “sure”. It has been noticed that such a step improves the accuracy of the results without adding a significant overhead

[Brody and Diakopoulos, 2011]. In addition to the OOV approach, the Porter Stemming algorithm [Porter, 1980] was considered as an extra step in order to improve the algorithm's accuracy and also reduce the number of unique words seen in the corpus. According to [Porter, 1980], word suffixes can be removed such that words with similar meaning are treated as identical. This method aims to improve the performance of the Information Retrieval (IR) systems by decreasing the number of distinct words. However, as the chart in Figure 5.1 shows, the number of unique words were minimally reduced[1] without improving the accuracy of the final results.

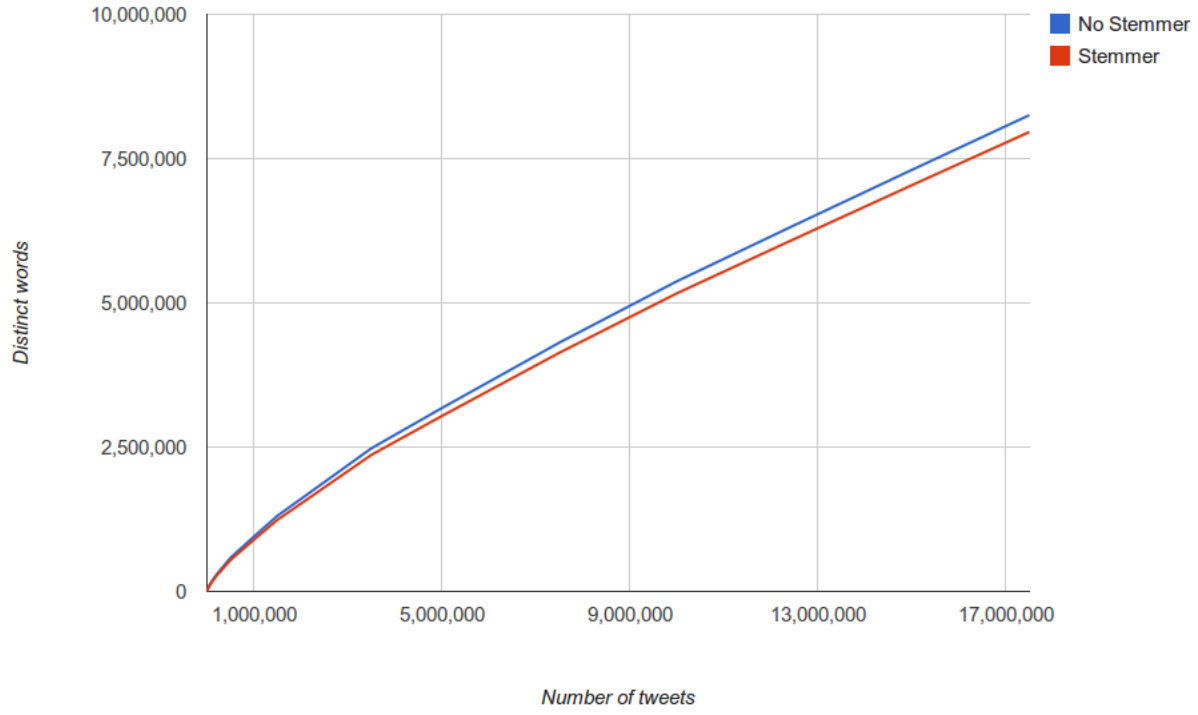

Figure 5.1: Number of distinct words applying Stemming algorithm.

Having replaced the OOV words, the URLs and mentions (@) are removed from the tweet text. Twitter users use the mention symbol to refer to a personal account or to a reply to a specific person. Moreover, it has been observed that URLs add linearity to the number of distinct words seen in the corpus. Storing and processing linear growing unigrams would be catastrophic and therefore have to be removed. Figure 5.2 proves the above remark by showing a comparison between the number of distinct words and the number of links seen in a 30 million tweets corpus.

The algorithm continues by representing each tweet in the vector space using the TF-IDF weight-

[1]Note that URLs are still counted as words.

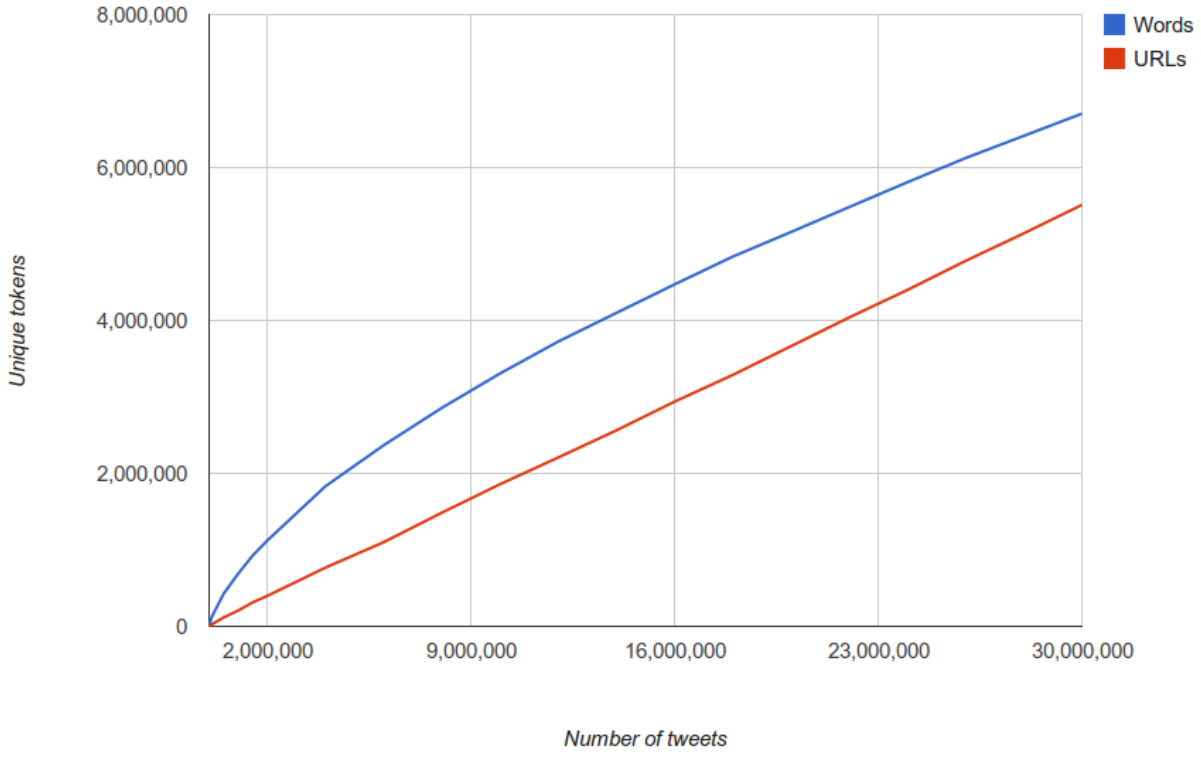

Figure 5.2: Comparison between number of distinct words and number of URLs.

ing as mentioned in section 2.2.1. Each vector is normalized using the Euclidean norm. Given a n-dimension vector $\vec{x}$ the Euclidean norm is defined as

$$\|x\| = \sqrt{x_1^2 + \cdots + x_n^2} \tag{5.1}$$

Further to the algorithm description, the tweet vector is inserted in each bucket - hash-table to apply the LSH algorithm as described in 2.2.1. The hash value is created using the random projection method. According to this, k random vectors are assigned to each bucket. Each random vector consists of values derived from a Gaussian distribution $N(0,1)$. Then the binary hash value consists of k bits each of which is calculated using the Function 2.3. In all implementations, 13 bits were used to define the length of the hash value.

The benefit of the LSH algorithm relies on the fact that similar documents are more likely to hash to the same value. Having found all approximate neighbours, the tweet is compared to the top-3L documents that collides more frequently with, where L is the number of buckets - hash-tables. A tweet can collide more than once with the same neighbour as documents are hashed

to all L buckets. In both sequential and Storm versions, 36 buckets were used. Further to the comparison, the cosine similarity measure is being used, as defined by Function 2.2. As an extra step, the tweet is also compared to a fixed number of most recently seen tweets if the distance from the nearest neighbour is above a given threshold of 0.4. The threshold value was calculated heuristically. Finally, the neighbour with the closest distance is assigned as the nearest.

The above procedure would require infinite memory for hashing of all previously seen tweets into the buckets. To prevent this from happening, we keep up to 20 tweets with the same hash per bucket. Thus in our case, given the maximum number of distinct hash values r a bucket can store, and queue size q the maximum number of colliding tweets the bucket allows, the bucket capacity c using 13 bits for the hash value representation is:

$$c = r * q = 2^{13} * 20 = 163840 \tag{5.2}$$

In case the queue size of a hash value has been reached, the oldest entry is removed.

5.2 Deploying on Storm

This section lists the experiments conducted on Storm processing system together with their results. All experiments were conducted on Storm 0.7.4 version, which was Storm's most recent stable version to the date this project was implemented. Also, Storm offers two ways of running topologies, the local and the remote mode. The remote mode requires a cluster to run whereas local mode is useful for development and testing. Local mode was the one used by this study, on the machine specified in section 4.1.

5.2.1 Baseline Topology (BLT)

5.2.1.1 Topology description

This implementation represents the simplest FSD system on Storm. The whole algorithmic logic is contained in a single bolt. The bolt receives tweets from the input and processes them sequentially. The process is identical to the one described in section 5.1.

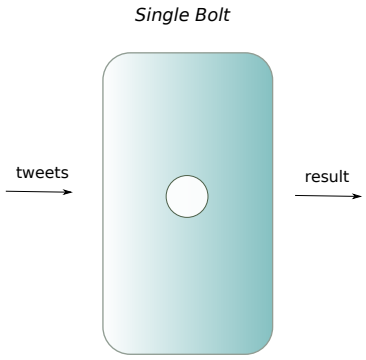

Figure 5.3: Baseline Topology

As the Figure 5.3 shows, the single bolt receives a stream of tweets as input and emits the result for each tweet. Each result consists of the tweet id and the score assigned to the tweet according to the distance from the nearest neighbour. As described in section 3.5, Storm uses the anchoring mechanism to ensure that tuples which fail to process will be replayed. For this reason, the input carries an additional unique id, the incoming tuple's id. This field of type *long* is unique and represents the tuple inside the topology, such that the tuple tree can be constructed and thus reliability can be satisfied.

5.2.1.2 Results

Running the baseline topology on the TWT100 evaluation set (100,000 tweets) produced the results shown in table 5.1. The larger evaluation set TWT500, consisted of 500,000 tweets could not finish in reasonable time and therefore was not used for this particular implementation. The baseline topology was implemented in both single-threaded and multi-threaded versions. The latter uses 4 threads.

The baseline version has been implemented using one task in the single bolt. As a result, it is obvious that the idea of different parallelization between bolts does not apply to the current context.

Regarding the accuracy, the results present no deviation compared to the sequential version, and

Version	Evaluation Set	Execution Time (minutes)
Single-Threaded	TWT100	9252.06
Multi-Threaded	TWT100	2489.10
–	TWT500	not applicable

Table 5.1: BLT execution times

this implementation will be used as a baseline for comparison of the next 2 topologies.

5.2.2 Ad-Hoc Topology (AHT)

5.2.2.1 Topology description

As the name describes, the Ad-Hoc topology was designed to work with optimal performance for a particular set of machine specifications. In our case, machine's specifications as described in section 4.1 can handle a limited number of threads. It is assumed that a topology which benefits from scalability but also considers the cost raised by the transfer of messages between components is supposed to achieve the best results. Following this hypothesis, the Ad-Hoc topology was designed, which is presented in Figure 5.4. It is worth mentioning that the topology was executed as a distributed RPC (DRPC) case. The current version of Storm allows a DRPC method to include a particular type of bolt, an instance of *IBatchBolt* interface. According to this, tuples arrive at the bolts in batches and each bolt is able to wait until all tuples that come from different tasks but belong to the same batch are processed before emission. Our Storm implementations took advantage of this type, as the nature of the problem imposes a strict synchronization in the processing order of the tweets.

To explain further, let there be a case where a partial counting is handled by 4 tasks in a bolt and the next bolt computes the Term Frequency map (UpgraderBolt). We assume that two tweets arrive simultaneously. Each partial counter task computes the count for some tweet words and emits to the UpgraderBolt. Before $task_1$ finishes its $tweet_1$ words processing, $task_2$ has emitted the partial count for both $tweet_1$ and $tweet_2$ that arrive at its input. Due to the fact that tasks are handled by different machines, this can actually happen. Then UpgraderBolt will count terms from both $tweet_1$ and $tweet_2$ and $tweet_1$ vector will be constructed with wrong values. To prevent

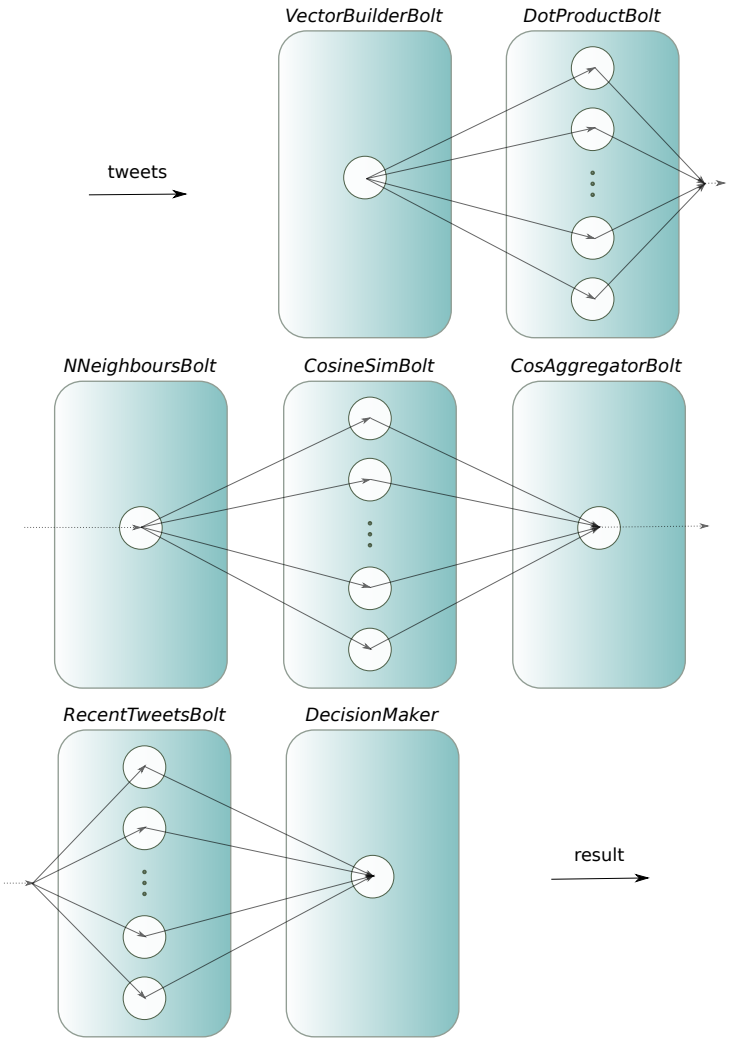

Figure 5.4: Ad-Hoc Topology

this from happening, the BatchBolt class provides a *finishBatch* method which is called only after all tuples in the batch have been processed. Our Storm versions benefit from this class by simulating batches with tweets (each batch contains a single tweet).

Each component's role is described below in brief. The different groupings occurring during the emissions are also enlisted. In case that the next bolt contains only a single task the grouping method is skipped.

VectorBuilderBolt (*BaseBasicBolt*): The VectorBuilderBolt receives the tweet id along with the tweet text, and emits a tweet object and the number of new words seen in this tweet. This bolt consists of a single task. Specifically, it computes the term frequencies, creates

the vector using TF-IDF weighting and updates the inverse document frequencies for the words seen. The URLs as well as the mentions are not included in the tweet vector, for the reasons explained in section 5.1. The vectors are 1-*d* sparse matrices and were implemented using the CERN Colt software package[2]. The vectors are also normalized using the Euclidean norm. Finally, the VectorBuilderBolt emits the tweet object which contains the tweet vector and the number of new words. An *All Grouping* is used such that the tweet arrives at all tasks of DotProductBolt.

DotProductBolt (*BaseBatchBolt*): The DotProductBolt contains the hash-tables (buckets) where previous tweets have been inserted, as described in section 5.1. To achieve parallelization, the total number of buckets is equally distributed to more than one tasks of this bolt. The DotProductBolt begins with an update to the dimensions of the random vectors that belong to each hash-table (bucket), according to the number of new words seen in VectorBuilderBolt. The random vectors are now ready to be multiplied with the tweet vector in order to produce the dot product. Each task then inserts the tweet into the buckets assigned to it, and finds the tweets that collide with it (possible neighbours). As shown by the LSH algorithm, similar tweets have high probability to hash to the same value. Finally, each task emits the tweet along with the possible neighbours to the next bolt.

NNeighboursBolt (*BaseBatchBolt*): This bolt's purpose is to collect the possible neighbours emitted by the DotProductBolt tasks and sort them according to the frequency they collide with the input tweet. Collisions can occur more than once for the same tweet, since each tweet is hashed to all buckets. As stated in section 5.1, only the top-3L tweets need to compare to the input tweet, where L is the number of buckets. Having gathered the top-3L tweets, these are then distributed to the CosineSimBolt tasks using a *Shuffle Grouping*.

CosineSimBolt (*BaseBatchBolt*): Each task of CosineSimBolt receives a part of the top-3L tweets to make a comparison using the cosine similarity measure. Finally, each task emits the nearest neighbour found along with the input tweet to the single task of CosAggregatorBolt.

CosAggregatorBolt (*BaseBatchBolt*): A single task is used to compare the closest neighbours received from the CosineSimBolt's tasks. The task also keeps the 500 most recently

[2] `http://acs.lbl.gov/software/colt/`

seen tweets in memory. Having found the neighbour with the closest distance, it then decides if a comparison with the 500 most recently seen tweets is required. This is estimated by comparing the closest neighbour distance d to a fixed threshold t of 0.4. If the closest neighbour distance is longer than a given threshold, then the task distributes the $<tweet, recent\ tweet>$ pair to the tasks of RecentTweetsBolt. A *Shuffle Grouping* is used to ensure that tasks receive an equal amount of load. Otherwise, the task emits the tweet and a *null* value as an indicator that no comparison is required.

RecentTweetsBolt (*BaseBatchBolt*): According to CosAggregatorBolt's decision, the tasks of this bolt compare the cosine similarity between the input tweet and the recent ones. The closest neighbour from the partial pairs received is emitted to the single task of DecisionMaker that takes the final decision.

DecisionMaker (*BaseBatchBolt*): This bolt consists of a single task which compares the incoming closest neighbours and emits the final result. The emitted result consists of the tweet and a score. Given a minimum distance d, the score s is computed as

$$s = 1 - d \tag{5.3}$$

An example output is summarized in Table 5.2.

Tweet ID	Score
1	0.0
2	0.6364
3	0.6706
4	0.6359
5	0.6253
6	0.4423
7	0.5294

Table 5.2: AHT example output

All classes seen in the topology are instances of BaseComponent class, which means that the "acking" procedure is automated by Storm. Otherwise, the programmer would have to manually acknowledge to the previous bolt upon finishing a tuple processing.

5.2.2.2 Results

In order to run the Ad-Hoc topology, a number of configurations is required, such as the level of parallelism in each bolt, the number of recent tweets kept for comparison etc. The configuration is summarized in Table 5.3. Bolts that appear in the topology but are not mentioned in the configuration use a single task. Running in local mode, means that Storm will simulate worker

DotProductBolt	18
CosineSimBolt	12
RecentTweetsBolt	18
L	36
k	13
Recent tweets to compare with	500

Table 5.3: AHT configuration

nodes with threads. For the topology needs, 52 threads were used, as this was found to maximally utilize the machine resources. Table 5.3 shows that DotProductBolt uses 18 tasks whereas CosineSimBolt uses 12 and RecentTweetsBolt 18 tasks. This leaves 4 tasks for the remaining 4 single-task bolts. The number of buckets, L, was set to 36 and the length of the hash, k, to 13 bits. Also, the 500 most recently seen tweets are kept in memory for possible comparison. Using this configuration, the results are presented in Table 5.4

Evaluation Set	Execution Time (minutes)
TWT100	681.72
TWT500	6085.02

Table 5.4: AHT execution times

The configuration used to produce the results listed in Table 5.4 was found to be optimal towards performance. This happened by observing the execution time after a number of changes in the level of parallelism in each bolt. As mentioned in section 4.2.2, this is required to find the most resource-demanding parts of the topology that need additional compute capacity. In a topology such as AHT, where only 3 bolts use more than one task, and bolts have similar functionality,

this may not lead to a significant gain. However, as the topology grows (e.g. MST) and different bolts handle different types of tasks, this is critical for the final performance. Considering that the accuracy of these experiments is not relevant and in order to run multiple configurations on a smaller basis, we reduced the number of tweets to 20.000. Additionally, the number of buckets were reduced to 12 and the number of recent tweets for comparison to 65. Also, to allow more collisions to occur earlier, the length of the hash value in each bucket was set to 5 bits. Table 5.5 demonstrates the execution time along with the different configurations in parallelism.

DotProductBolt	6	6	4
CosineSimBolt	4	6	3
RecentTweetsBolt	6	4	9
Execution time (minutes)	34.61	43.76	37.81

Table 5.5: AHT execution times on different configurations

Finally, in an attempt to measure the correctness of this topology comparing to the BLT results, the *Mean Absolute Error* (MAE) metric was used. This is defined by the equation 4.2. Having set the configurations of Table 5.3 the MAE is shown in Table 5.6.

BLT vs. AHT	
MAE	0.054

Table 5.6: AHT Mean absolute error

5.2.3 Maximum Scalability Topology (MST)

5.2.3.1 Topology description

The Maximum Scalability Topology (MST) is designed to benefit from *unbounded* resources. By the term unbounded we mean a cluster that can support as much compute capacity as needed. Running this topology in such an environment, means that the cost raised by the transfer of messages between components will be minimal compared to the gain the topology will achieve from the maximum parallelization. However, the cost may be relatively higher when resources

offered are limited. To this direction, the topology design is demonstrated in Figure 5.5 and explained as follows.

SplitterBolt (*BaseBasicBolt*): The SplitterBolt receives the tweet id along with the tweet text and a single task splits the tweet text into words. The words are then emitted to tasks of the PartialCounter bolt using a *Shuffle Grouping*.

PartialCounter (*BaseBatchBolt*): This bolt consists of a number of tasks each of which receives an equal amount of words from SplitterBolt. Each task counts the words that arrive and when no more tweet words remain, it emits the partial counts to the single task of UpdaterBolt.

UpdaterBolt (*BaseBatchBolt*): Upon all partial counts arrivals, the single task of UpdaterBolt sums the counts to compute the Term Frequencies (TF) for the input tweet. It also keeps an updated version of the document frequencies of each term. The bolt emits the $< term,\ TF >$ pair for each distinct term along with its IDF. Additionally, it emits the number of new words seen by this tweet, which will be used later to update the random vectors of each bucket. A *Shuffle Grouping* is used on this emission.

PartialVectorBolt (*BaseBatchBolt*): As the name denotes, PartialVectorBolt contains tasks which compute parts of the vector according to TF-IDF weighting. The partial vector values along with their position in the vector are then emitted to the single task of TweetConstructor using a *Global Grouping*.

TweetConstructor (*BaseBatchBolt*): TweetConstructor uses a single task to accumulate the partial vector values and construct the tweet vector. As in AHT VectorBuilderBolt, the URLs as well as the mentions are removed from the tweet, and the vectors are normalized using the Euclidean norm. The tweet object is finally emitted which contains the tweet vector and the new words seen.

The rest of the topology (6 bolts) follows the same architecture as the AHT which is explained in subsection 5.2.2.1. At the end, the output of the topology is the tweet id together with the score assigned. The output follows the same format as the AHT output shown in Table 5.2.

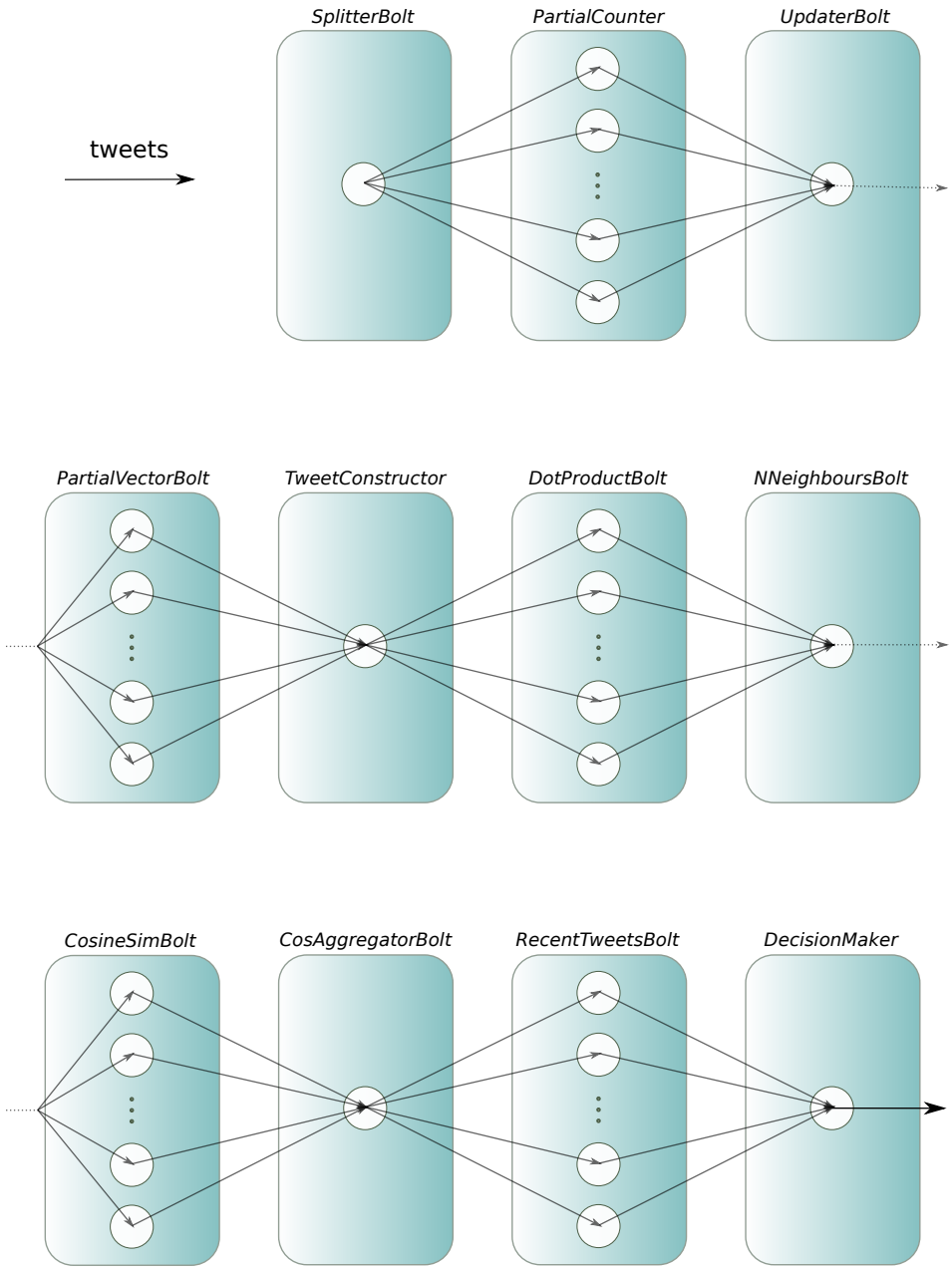

Figure 5.5: Maximum Scalability Topology

5.2.3.2 Results

The MST implementation used the configuration shown in Table 5.7. We only demonstrate the level of parallelism in each bolt since the parameters L, k and *Number of recent tweets to compare*

with, are identical to AHT values. Bolts that appear in the topology but are not mentioned in the configuration use a single task. To conduct a fair comparison with AHT, 52 threads were used as

PartialCounter	4
PartialVectorBolt	4
DotProductBolt	18
CosineSimBolt	8
RecentTweetsBolt	12

Table 5.7: MST configuration

well. For the levels of parallelism shown in Table 5.7 the results are summarized in Table 5.8.

Evaluation Set	Execution Time (minutes)
TWT100	762.65
TWT500	8246.36

Table 5.8: MST execution times

As also performed by the AHT, a number of different configurations leaded to the selection of this configuration. A sequence of changes in bolts' parallelism showed that there is a significant difference in real execution time of different configurations. In the current topology, finding the bottlenecks plays an important role because the resources are limited and the number of bolts more than the AHT ones. Again, to achieve multiple runs and as long as the accuracy is not relevant at this point, we tested multiple configurations on 20.000 tweets. Furthermore, the buckets were reduced to 12 and the number of bits that construct the hash value in each bucket to 5. The configurations together with the execution time results are summarized in Table 5.9.

Similarly to AHT, the accuracy of the results was computed according to the 4.2 equation. Having set the configuration of Table 5.7, the MAE when testing on the TWT100 evaluation set is shown in Table 5.10.

PartialCounter	4	2	2
PartialVectorBolt	4	2	6
DotProductBolt	4	6	4
CosineSimBolt	2	4	2
RecentTweetsBolt	4	6	4
Execution time (minutes)	36	26.61	35.44

Table 5.9: MST configurations

BLT vs. MST	
MAE	0.055

Table 5.10: MST Mean absolute error

5.3 Discussion

Prior to demonstrating the analysis of the results acquired by this research, it is important to re-state the aim of the current project, which is to implement the FSD task on the Storm distributed platform towards scalability, robustness and high throughput rate. As the accuracy metric in

Topology	MAE
AHT	0.054
MST	0.055

Table 5.11: Mean absolute error on TWT100 evaluation set

Table 5.11 demonstrates, the problem has been addressed successfully in a distributed manner. Table 5.11 shows that there is no loss in accuracy of the results when running the FSD task on Storm. LSH is a randomized algorithm which may have slightly different results over different runs. Additionally, the low values of MAE show that robustness has been satisfied. Specifically, all tuples were processed by the consumers since tuple id's were sent along with the batches. The type of components that were used by all topologies ensured that the acknowledgements upon tuple deliveries are automatically sent by Storm.

5.3.1 Efficiency between Different Topologies

When comparing the BLT performance to the two scalable versions, we can derive that AHT and MST clearly outperform the single-bolt topology. AHT and MST completed the TW100 evaluation set in less execution time, demonstrating the gain from distributing the work across workers. In case that a multi-threaded version of BLT is used, AHT and MST also present better results. Specifically, the speed-up achieved in the case of TWT100 evaluation set is summarized in Table 5.12.

	Speed-up (single-threaded)	Speed-up (multi-threaded)
AHT	1357.16 %	365.12 %
MST	1213.15 %	326.37 %

Table 5.12: AHT, MST Speed-up over BLT on TWT100 evaluation set

Table 5.12 showed that implementing a distributed FSD on Storm is remarkably more efficient than the baseline version. This confirms our hypothesis about the performance of a distributed real-time system. However, as assumed in sections 5.2.2 and 5.2.3, there is a cost associated with the deployment of the FSD task when more than one bolts are used. It has been observed that the way Storm components communicate with each other has an overhead which is noticeable as the number of workers decreases. This happens because - as mentioned in section 3.3 - components pass values sequences in tuples, which need to be serialized before emission and de-serialized upon arrival. Serialization is an expensive task which cannot be avoided in such a system[3]. The message passing overhead had greater impact when testing the topologies on smaller machines. Storm versions were found to be slower than the multi-threaded one on an 8-core machine.

Regarding the TWT500 evaluation set, a comparison with the BLT version cannot be performed since the latter did not manage to finish the processing in reasonable time. We can however conclude that the AHT and MST implementations clearly outperform BLT as the data volume increases. Comparing the scalable versions with each other, shows that the AHT is again faster than the MST, achieving a speed-up of 135.52 %. Referring to both evaluation sets, it can be derived that the AHT presents better performance than the MST on the same machine since

[3]Storm uses the *Kryo* (`http://code.google.com/p/kryo/`) serialization framework to achieve fast serialization of primitive data types.

the serialization cost for sending and receiving messages is lower. This happens because AHT consists of 7 components, whereas MST uses 11 components in total. Therefore, on a machine with bounded resources, MST will spend more time on message passing rather than on clear computation. As opposed to the MST, Ad-Hoc Topology was designed to benefit the most from the scalability that Storm offers while keeping the message cost from communication in lower levels. Finally, it is reasonable to expect MST to behave better than AHT as the number of workers increases.

The processing time of the TWT500 evaluation set comparing to the smaller TWT100 did not present a linear growth. This happens because as tweets are hashed into the buckets, more collisions will occur over time until all buckets are filled up. This is also the reason that BLT topology could not finish in reasonable time. As we approach the state where buckets are fully loaded, the probability that a tweet will be found close to a near neighbour is higher, thus more collisions occur and tweets are more likely to skip the comparison with most recently seen tweets. This observation can be useful for future versions of Storm that will support a *rebalance* command. As a result, more tasks can be allocated to RecentTweetsBolt in early stages of the run and the topology can dynamically change over time to offer more resources to DotProductBolt and CosineSimBolt bolts that handle the bucket collisions.

There is a particular interest in the way Storm copes with bursts of incoming data. Speaking of a real-time streaming system, input data rate may be uneven. Therefore, there is a possibility that data arrive at a higher rate than the system can handle. For example when a hurricane happens, a burst of tweets will arrive at the input. It is reasonable that if bolts are unable to process the incoming data, memory problems will eventually occur. To prevent any kind of queue explosion in the topology, Storm tries to accumulate traffic on the spouts and allow a limited number of tuples in the topology at the same time. According to a spout property[4], the user can define the maximum number of tuples that are processed but have not yet completed. The ability of Storm to track tuples makes the latter definitely feasible.

[4]Config.TOPOLOGY_MAX_SPOUT_PENDING

5.3.2 Load Balancing

Results have shown that changing the level of parallelism in the topology produces different execution times. Since load balancing is left to the user, such decisions are critical for the overall performance.

Regarding the AHT topology, Table 5.5 showed that when more resources are allocated to DotProductBolt and RecentTweetsBolt, the topology presents faster processing. Consequently, it can be concluded that DotProductBolt and RecentTweetsBolt are the bolts that need the highest computational power. The reason for this is that the construction of a dot product is an expensive process. The amount of load that DotProductBolt and RecentTweetsBolt consume is similar. Specifically, in the real case scenario, where 36 buckets contain 13-bit long hash values, DotProductBolt will construct 468 dot products by the time all buckets are filled. This will eventually happen. At the same time, RecentTweetsBolt will handle 500 comparisons (using the cosine similarity measure) which according to Function 2.2 contains a dot product computation. However, as mentioned in section 5.2.2, a comparison with recent tweets will occur only if the tweet is found to be above a given distance from the closest neighbour found in buckets. This means that RecentTweetBolt is likely to be idle for some tweets. Finally, CosineSimBolt requires less computational power as it only needs to compare the tweet with the top 3L neighbours that collide more frequently with it, where L is the number of buckets. Consequently, CosineSimBolt will compute up to 108 cosine similarity values. Having performed multiple runs on the data, the first configuration achieved the optimal performance.

Judging from the AHT results, the optimal parallelism proportions between the three identical bolts should be retained in the MST configuration. For this reason, we only changed the number of tasks in PartialCounter and PartialVectorBolt, and reduced or increased the rest of the topology accordingly, to keep the total number of threads constant. As shown in Table 5.9 the topology is more efficient when more resources are allocated to the last three bolts. This happens because PartialCounter and PartialVectorBolt operations are much easier to compute comparing to the dot product ones that the last three bolts include. Finally, the second configuration was found to be optimal.

In conclusion, the results demonstrated that more resources are needed at the bolts that compute the dot product function as it is the most demanding process. The above comparisons helped us

to configure the parallelism inside components on a small basis and then to apply it on the real case scenario.

On the whole, topologies should be designed according to the usage purposes. Considering the trade-off between tuple passing overhead and scalability in the topology, one can design a system that benefits from maximum scalability if plenty of resources are available. On the other hand, in case that resources are the primary concern, the system can reduce the scalability of the program by using fewer components, in an attempt to bound the "suffering" from communication costs. When more resources are available, scaling is easy by defining more tasks to the components or by increasing the number of defined workers in each topology. Furthermore, the distribution of the available resources to bolt tasks is also an important aspect. Overall, Ad-Hoc Topology was found to be the most suitable for deployment on our machine.

This chapter demonstrated the results of running different experiments on Storm. Although the resources were limited and the time-frame restricted, valuable conclusions were drawn about the way that our FSD application behaves in a distributed fashion.

Chapter 6

Conclusions

This chapter comprises a summarization of the observations made on this study and continues with the demonstration of the future work that can be applied further to this field.

6.1 Final Remarks

This work implemented the FSD task in a distributed fashion, to investigate differences in scalability, performance, accuracy and robustness. The idea of using Storm to address this problem was successful and leaded to better performance without loss in results correctness. Three different strategies were followed in order to expose strengths and weaknesses of the Storm platform and to finally find the optimal Storm implementation that addresses the FSD problem efficiently.

Although message serialization adds an inevitable overhead to the program's performance, Storm deployments have achieved a significant gain in speed and throughput rate. In addition, the ability to change the level of parallelism in each topology allows the identification of the system bottlenecks. Therefore the resources were given to more demanding parts (dot product computation) in such a manner that the overall performance is greater. The topology with less components (AHT) benefited from reduced serialization costs and therefore achieved better performance. Regarding the accuracy of both AHT and MST, it minimally deviated from the baseline version but this is acceptable as the approach used was randomized.

Nevertheless, the resources available for the project did not permit further investigation of this

problem. The contribution to the research field is however considerable, in a period that the interest in real-time streaming applications is growing in a large scale.

6.2 Future Work

The limited time-period did not allow a deep cover of every area. Based on our work as well as the current state-of-the-art, the horizon can be expanded as stated in this section.

First of all, the different Storm versions can be deployed on a large cluster. The resources provided for this project allowed only a limited view of the overall picture. An interesting approach would be to use an Amazon Elastic Compute Cloud (EC2) [1] or an equivalent web service. Amazon (EC2) stands as a web service that offers computing resources that can be resized according to the project needs. Storm facilitates this procedure by providing a project which simplifies the launch of Storm clusters on Amazon Web Services[2].

Twitter may be an excellent source of first stories but it is not the only one. A first story detection system can benefit from multiple sources. As a follow-up work, we suggest combining information from different sources such as Wikipedia[3]. This will enable a filtering in the events detected such that false alarms are no longer marked as first stories. Also, *neutral* events which are neither events or spam could be detected more accurately. Towards this direction, [Osborne et al., 2012] have improved the quality of the first story events using Wikipedia as a filtering mechanism. However, they present a limitation to this approach because Wikipedia is not fully synchronized with Twitter but 2 hours slower. There is an area of improvement regarding the scalability and speed of this method if a distributed approach will be used. Since Storm spouts can handle streams from multiple sources, such a system can be built on Storm.

Another improvement that could be considered is the implementation of an FSD system on top of StreamReduce. According to Nodeable[4], the company behind it, "StreamReduce is Nodeables real-time aggregation and processing engine that enhances and analyzes data streams as a cloud-based complement to batch processing via Apache Hadoop and Amazon Web Services Elastic

[1] http://aws.amazon.com/ec2/
[2] The project can be found at https://github.com/nathanmarz/storm-deploy/
[3] http://www.wikipedia.org/
[4] http://www.nodeable.com/

Map Reduce".

Moreover, the FSD task should be certainly developed in future versions of Storm. To the date this project was developed, Storm's most stable version was 0.7.4. Version 0.8.0 will be the upcoming Storm release but is currently at a development stage. It includes further improvements, such as throughput increase up to 10x and faster serialization between components. The new version will allow dynamical change of the parallelism in the running topology according to the needs. As section 5.3 explains, this will be useful for our FSD task. It also includes a higher level abstraction, the *Trident*, which allows to perform joins, aggregations, grouping, functions, and filters. The 0.8.0 version change-log can be found in the Storm user group. It is believed that program's execution time will be significantly reduced and the program will benefit to a greater extent from the available resources.

Bibliography

[Allan, 2002] Allan, J., editor (2002). *Topic detection and tracking: event-based information organization*. Kluwer Academic Publishers, Norwell, MA, USA.

[Allan et al., 2000] Allan, J., Lavrenko, V., Malin, D., and Swan, R. (2000). Detections, bounds, and timelines: Umass and tdt-3. In *Proceedings of Topic Detection and Tracking Workshop (TDT-3)*, pages 167–174.

[Brody and Diakopoulos, 2011] Brody, S. and Diakopoulos, N. (2011). Cooooooooooooooollll-lllllllll!!!!!!!!!!!!!!: using word lengthening to detect sentiment in microblogs. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '11, pages 562–570, Stroudsburg, PA, USA. ACL.

[Charikar, 2002] Charikar, M. S. (2002). Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, STOC '02, pages 380–388, New York, NY, USA. ACM.

[Creutz et al., 2007] Creutz, M., Hirsimäki, T., Kurimo, M., Puurula, A., Pylkkönen, J., Siivola, V., Varjokallio, M., Arisoy, E., Saraçlar, M., and Stolcke, A. (2007). Morph-based speech recognition and modeling of out-of-vocabulary words across languages. *ACM TSLP*, 5(1):3:1–3:29.

[Dean and Ghemawat, 2004] Dean, J. and Ghemawat, S. (2004). Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA. USENIX Association.

[Grauman and Darrell, 2007] Grauman, K. and Darrell, T. (2007). Pyramid match hashing: Sublinear time indexing over partial correspondences. In *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on*, pages 1 –8.

[Hindman et al., 2011] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. (2011). Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 22–22, Berkeley, CA, USA. USENIX Association.

[Hunt et al., 2010] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010). Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA. USENIX Association.

[Indyk and Motwani, 1998] Indyk, P. and Motwani, R. (1998). Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 604–613, New York, NY, USA. ACM.

[Jacobs, 2009] Jacobs, A. (2009). The pathologies of big data. *Commun. ACM*, 52(8):36–44.

[Luo et al., 2007] Luo, G., Tang, C., and Yu, P. S. (2007). Resource-adaptive real-time new event detection. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 497–508, New York, NY, USA. ACM.

[Muthukrishnan, 2005] Muthukrishnan, S. (2005). *Data streams: algorithms and applications*. Now Publishers.

[Neumeyer et al., 2010] Neumeyer, L., Robbins, B., Nair, A., and Kesari, A. (2010). S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, pages 170–177, Washington, DC, USA. IEEE Computer Society.

[Osborne et al., 2012] Osborne, M., Petrovic, S., McCreadie, R., Macdonald, C., and Ounis, I. (2012). Bieber no more: First story detection using twitter and wikipedia. In *Proceedings of the SIGIR 2012 Workshop on Time-aware Information Access*, SIGIR '12, Portland, Oregon, USA.

[Paulevé et al., 2010] Paulevé, L., Jégou, H., and Amsaleg, L. (2010). Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern Recogn. Lett.*, 31(11):1348–1358.

[Petrović et al., 2010] Petrović, S., Osborne, M., and Lavrenko, V. (2010). Streaming first story detection with application to Twitter. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, HLT '10, pages 181–189, Stroudsburg, PA, USA. ACL.

[Petrovic et al., 2012] Petrovic, S., Osborne, M., and Lavrenko, V. (2012). Using paraphrases for improving first story detection in news and twitter. In *HLT-NAACL*, pages 338–346.

[Porter, 1980] Porter, M. (1980). An algorithm for suffix stripping. *Program*, 14(3):130–137.

[White, 2009] White, T. (2009). *Hadoop: The Definitive Guide*. O'Reilly, first edition edition.

[Yang et al., 1998] Yang, Y., Pierce, T., and Carbonell, J. (1998). A study of retrospective and on-line event detection. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '98, pages 28–36, New York, NY, USA. ACM.

[Zaharia et al., 2010] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA. USENIX Association.