NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

DIPLOMA THESIS

Distributed tag-set correlation calculation using storm

Sotirios P. Fokeas

Supervisor: **Alex Delis,** Professor NKUA

ATHENS

JULY 2015

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Κατανεμημένος Υπολογισμός συσχετίσεων συνόλων ετικετών

Σωτήριος Π. Φωκέας

Επιβλέπων: Αλέξης Δελής, Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

ΙΟΥΛΙΟΣ 2015

DIPLOMA THESIS

Distributed tag-set correlation calculation using storm

Sotirios P. Fokeas
A.M.: 1115200700157

SUPERVISOR: **Alex Delis,** Professor NKUA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Κατανεμημένος Υπολογισμός συσχετίσεων συνόλων ετικετών

Σωτήριος Π. Φωκέας
Α.Μ.: 1115200700157

ΕΠΙΒΛΕΠΩΝ: **Αλέξης Δελής,** Καθηγητής ΕΚΠΑ

ABSTRACT

In this work, we target on analyzing data published in Social Media. Users of Social Media networks post text, images or videos and annotate each one of them with a set of tags describing their content. What we seek is to find an efficient way to compute the correlations of co-occurring tags over time. The amount and pace of published posts makes it necessary to parallelize the computations. Data is divided to multiple nodes making each one of them responsible of computing correlations of its own share. What is challenging in a setting like this is to ensure that each node will compute a subset of the coefficients and the processing load will be evenly distributed across nodes. For this reason, a graph is devised that can maintain all essential data. This graph is dynamically and continuously partitioned across nodes. The proposed approach seeks to build an efficient model to not only effectively calculate correlations, but also divide the load in a natural and self-organizing way amongst peers. Finally, the scheme is prototyped in Java, using Apache Storm Stream Processing platform, which effectively demonstrates that this approach is feasible.

SUBJECT AREA: Data Stream Processing

KEYWORDS: graph, social media, tags, distributed systems, correlations

ΠΕΡΙΛΗΨΗ

Στην εργασία αυτή, στοχεύουμε στην ανάλυση των δεδομένων που δημοσιεύονται στα μέσα κοινωνικής δικτύωσης. Οι χρήστες των κοινωνικών δικτύων καταχωρούν κείμενα, εικόνες ή βίντεο και υποσημειώνουν το καθένα από αυτά με ένα σύνολο ετικετών που περιγράφουν το περιεχόμενό τους. Βασική μας επιδίωξη είναι να βρούμε έναν αποτελεσματικό τρόπο για τον υπολογισμό των συσχετισμών των συνυπαρχόντων ετικετών. Ο τεράστιος όγκος και ο ρυθμός των δημοσιευόμενων μηνυμάτων καθιστά απαραίτητο τον παραλληλισμό των υπολογισμών. Τα δεδομένα χωρίζονται σε πολλαπλούς κόμβους, όπου ο καθένας από αυτούς είναι υπεύθυνος να υπολογίσει τις συσχετίσεις που του αναλογούν. Στην εργασία αυτή, αποτελεί μια μεγάλη πρόκληση, η εξασφάλιση ότι κάθε κόμβος θα υπολογίζει ένα υποσύνολο των συσχετίσεων και ότι το φορτίο επεξεργασίας θα είναι κατανεμημένο ομοιόμορφα σε όλους τους κόμβους. Για το λόγο αυτό, επινοήθηκε ένα γράφημα που μπορεί να διατηρήσει όλα τα απαραίτητα στοιχεία. Αυτό το γράφημα δημιουργείται δυναμικά και διαιρείται συνεχώς ανάμεσα στους κόμβους. Η προτεινόμενη προσέγγιση επιδιώκει να οικοδομήσει ένα αποτελεσματικό μοντέλο που όχι μόνο θα υπολογίζει αποτελεσματικά τις συσχετίσεις, αλλά επίσης θα χωρίζει το φορτίο επεξεργασίας με ένα φυσικό τρόπο ανάμεσα στους κόμβους. Τέλος, το μοντέλο υλοποιείται σε Java με τη χρήση της πλατφόρμας Apache Storm Stream Processing, αποδεικνύοντας έτσι πως η προσέγγιση μας ειναι εφικτή.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Επεξεργασία Ροών Δεδομένων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: γράφημα, κοινωνικά δίκτυα, ετικέτες, κατανεμημενα συστηματα, συσχετίσεις

ACKNOWLEDGMENTS

I would first like to thank my supervisor Alex Delis for his counseling and cooperation. Moreover, I would like to thank Foteini Alvanaki for her valuable ideas and guidance. Finally, I would like to thank my friend Isabel Mendonca for her patience and corrections.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PROLOGUE

The following work has been completed in Athens, 2015. It was completed as part of the bachelor program in the department of Informatics and Telecommunications of the University of Athens. It was conducted in the hope of providing useful knowledge for future studies and research in the topic.

1. INTRODUCTION

Social media users receive an endless flow of information, often at a rate far higher than they are capable to process. It is impossible for them to comprehend all the amount of information created on a daily basis. This could lead to information overload that would simply get them tired and reduce their enjoyment and productivity.

User generated content in these sites is usually annotated with short descriptive text. These are called hash-tags or simply tags. Our work focuses on finding a competent way to compute the correlations between co-occurring tags that appear in messages and posts published in social media. We aim to calculate these correlations in real time, thus, it is essential to parallelize the work across different nodes in a computing cluster. We design a graph that holds all information about the tags of parsed content. We then present a model to partition it across nodes. The complicated part is on keeping the load of nodes balanced, without hindering the performance in any way. At the end, we build a prototype using the Java language and the Apache Storm Framework. This prototype is built upon the principles outlined during the designing process.

One could then use these correlations in order to effectively detect popular-trending topics. With this tool at hand, users could find the topics they are really interested in. They can, therefore, spend less time searching and more time towards their end goal, ultimately resulting in a greater web experience.

1.1 Outline

The chapters are organized in the following way.

- Chapter 2 outlines all the necessary background information for the user to be able to follow the content of the next chapters.
- Chapter 3 examines the previous work related to the topic.
- Chapter 4 takes a closer look in the related work, detects what can be improved, and proposes the general idea of an alternative design based on these observations.
- Chapter 5 presents a new model of how to keep and handle information about incoming data from Social Media, as well as dividing it to multiple parts.
- Chapter 6 presents the technical details of an application that was built following the concepts of the previous chapter. It also displays a first test on the prototype application.
- Chapter 7 is the conclusion where what was done and which were the results is discussed.
- Chapter 8 is the last chapter where thoughts about possible future work are considered.

2. BACKGROUND

This chapter acts as an overview of some basic ideas and methods that will allow readers to better understand the work presented in the following chapters. These concepts are used throughout the text with the assumption that readers are accustomed to them. For more verbose presentations please refer to bibliography links.

2.1 Social Media

Social Media are a collective of online communications channels, where individual users publish the majority if not all the content. It is based on the idea of sharing and interaction between users. Popularity of Social Media is increasing with a high rate today, and is evident that it is something that is here to stay. Their popularity is high in academia as well. For example, you can refer to [8], [9] for research focusing on analyzing the content or to [10], [11] for research in users' interactions.

2.1.1 Twitter

Twitter is a good example of Social Media. Users of twitter communicate by publishing short messages limited to 140 characters. Whenever a user posts a message it is broadcasted across all other users that have subscribed to this user.

Posts in Twitter are called tweets and are public unless otherwise specified by their author. Tweets are annotated by a number of tags, assigned by the author, that describe its content. From this point forward, we will can this set of tags, a tag-set. In Twitter, tags are preceded by the special character # and for that they are called *hashtags*. The primary goal of hashtags is to organize information in twitter. Anyone can make a hashtag at any time, simply by typing a "#" in front of a phrase in a tweet. For instance, "#twitterIsAwesome". After a hashtag has been created, other Twitter users can use that hashtag in their own tweets to add to the larger conversation about that topic. Hashtags can be as general or specific as the user desires. They are created and managed by Twitter users, not Twitter itself.

Tweets can be posted by anyone having a tweeter account. Registered users can also subscribe to receive tweets of other users. Moreover, users can interact with each other by replying to tweets. Hence, a community is created by this network of users.

2.2 Stream Processing

A data stream is a sequence of packets of data used to transfer information. Each of these packets contains a small amount of information.

Data Stream Processing is the method of processing these data streams. Data arrives continuously and for each arriving packet an action is performed which, each time, depends on the application. Various examples and applications of Stream Processing exist.

For instance, [17] concerns Network monitoring, [20] sensor networks and [18], [19] web applications.

2.2.1 Apache Storm Framework

Apache Storm is a popular Distributed Data Stream Processing framework. The term Distributed Data Stream processing means that the Stream Processing is being performed not by one, but by multiple machines (nodes). Storm makes it easy to reliably process streams of data in real time. As in every distributed data stream processing setting multiple nodes receive portions of the data stream. Each node processes the data it receives locally using only its own resources, memory and CPU, and pushes the results to other nodes. Storm uses tuples as its data model. A tuple is a named list of values, and a field in a tuple can be an object of any type.

Topology

A Storm cluster is similar to a Hadoop cluster. Whereas on Hadoop you run ”MapReduce jobs”, on Storm you run ”topologies”. One key difference is that a MapReduce job eventually finishes, whereas a topology runs forever. A topology is a graph of computation. Each node in a topology contains processing logic, and links between nodes indicate how data should be passed around between nodes. An example of a storm topology can be seen in figure 1

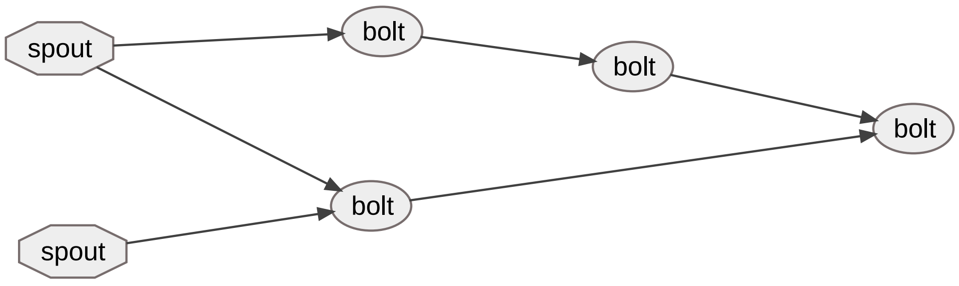


Figure 1: An example topology

Components

The basic primitives Storm provides for doing stream transformations are ”spouts” and ”bolts”. Spouts and bolts have interfaces that you implement to run your application-specific logic.

A spout is a source of streams in a topology. Generally spouts will read tuples from an external source and emit them into the topology. Spouts can either be reliable or unreliable. A reliable spout is capable of replaying a tuple if it failed to be processed by Storm, whereas an unreliable spout forgets about the tuple as soon as it is emitted. Spouts can emit more than one stream.

All processing in topologies is done in bolts. Bolts can do anything from filtering, functions, aggregations, joins, talking to databases, and more. A bolt consumes any number of input streams, does some processing, and possibly emits new streams.

Parallelism

Storm distinguishes between the following three main entities that are used to actually run a topology in a Storm cluster:

1. Worker processes
2. Executors (threads)
3. Tasks

A machine in a Storm Cluster may run one or more worker processes for one or more topologies. Each worker process runs executors for a specific topology. One or more executors may run within a single worker process, with each executor being a thread spawned by the worker process. Each executor runs one or more tasks of the same component. A task performs the actual data processing. Tasks are instances of the components (bolts and spouts). The picture 2 depicts a possible setting of what was described.

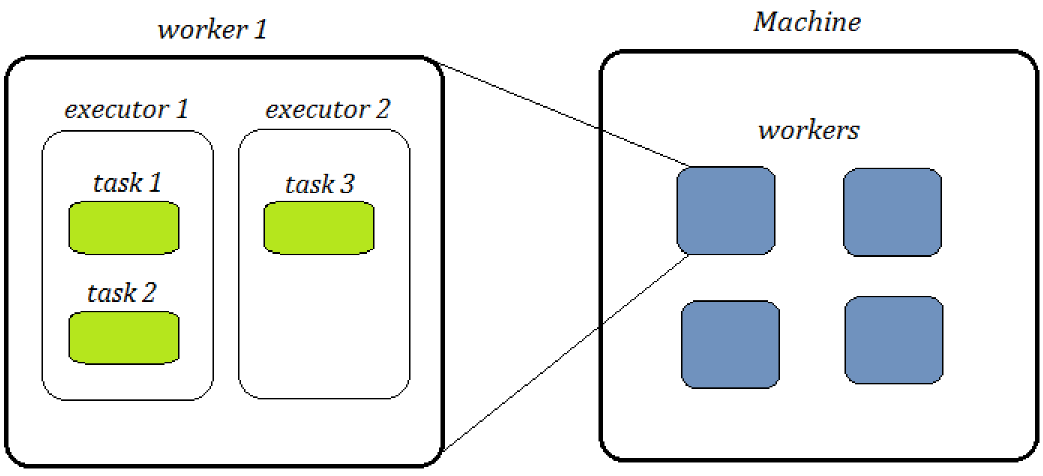


Figure 2: Parallelism in Storm

Storm Streams

The stream is the core abstraction in Storm. A stream is an unbounded sequence of tuples that is processed and created in parallel in a distributed fashion. Streams are defined with a schema that names the fields in the stream's tuples. Every stream is given an id when declared.

Part of defining a topology is specifying for each bolt which streams it should receive as input. A stream grouping defines how that stream should be partitioned among the bolt's tasks. The stream groupings that are most used are the following:

1. Shuffle grouping: Tuples are randomly distributed across the bolt's tasks in a way such that each bolt is guaranteed to get an equal number of tuples.
2. Fields grouping: The stream is partitioned by the fields specified in the grouping. For example, if the stream is grouped by the "user-id" field, tuples with the same "user-id" will always go to the same task, but tuples with different "user-id"'s may go to different tasks.
3. All grouping: The stream is replicated across all the bolt's tasks.
4. Direct grouping: This is a special kind of grouping. A stream grouped this way means that the producer of the tuple decides which task of the consumer will receive this tuple. Direct groupings can only be declared on streams that have been declared as direct streams.

Information for this chapter has been drawn out of Apache Storm Project's homepage [5], which you can visit to learn more about the framework. For a paper describing the use of Storm at Twitter please see [12].

2.3 Similarity Measures

Similarity measures are used to determine the similarity between objects. Numerous methods had been proposed to quantify similarity. In paper [6], Marie-Jeanne Lesot et al. present some basic and more advanced similarity measures. Depending on whether the objects in question are represented using a binary or a real vector, different similarity measures are defined.

In this work, we will focus on binary objects. When objects are represented as binary vectors, the similarity of them is measured by checking whether they have common elements or not. More specifically, the similarity between two binary objects **X** and **Y** can be expressed as: The number of elements present in both **X** and **Y** ($|\mathbf{X} \cap \mathbf{Y}|$) or the number of attributes present in **X** but not in **Y** ($|\mathbf{X} - \mathbf{Y}|$) or the number of attributes present in **Y** but not in **X** ($|\mathbf{Y} - \mathbf{X}|$) or the number of attributes absent from both **X** and **Y** ($|\overline{\mathbf{X}} \cap \overline{\mathbf{Y}}|$) .

2.3.1 Jaccard Coefficient

One of the most frequently used similarity measures for binary data is the Jaccard Coefficient (JC). Jaccard Coefficient was first described in [7] by Paul Jaccard. It increases as the number of elements present in both objects, increases and decreases as the number of elements present in either of the objects, but not in both, decreases. Assuming that there are two objects **X** and **Y** the Jaccard Coefficient between them is defined as:

$$J(\mathbf{X}, \mathbf{Y}) = \frac{|\mathbf{X} \cap \mathbf{Y}|}{|\mathbf{X} \cup \mathbf{Y}|}$$

The above formula can be extended to a general one, fitting to compute the similarity of more than two objects. Thus, assuming that we have binary objects $(\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n)$, the general formula would be:

$$J(\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n) = \frac{|\bigcap_{i=1}^{n} \mathbf{X}_i|}{|\bigcup_{i=1}^{n} \mathbf{X}_i|}$$

3. RELATED WORK

In this chapter we present work related to our own. Works under this section can be proven useful for a more spherical view of the work at hand. Apart from that, it can be a great place to start if you want to extend this work. Jaccard Coefficient is widely used and a lot of research has been published that uses this method of measuring similarity. Here are a few applications that utilize the Jaccard Coefficient similarity measure to accomplish some interesting goals.

Michihiro Kobayakawa et al. [15] uses Jaccard Coefficient to retrieve images based on their similarity to a query image. They propose an algorithm and a data-structure and study their theoretical aspects. The similarity measures how much overlapping is happening between areas of the images. These images could have been taken from a different angle or they could be zoomed in/out. Their algorithm detects that by comparing their pixel composition using the Jaccard Coefficient. Finally they perform tests in thousands of images and they show that their approach is many times faster than a naive algorithm.

An interesting example of Jaccard use is the one presented in [16] by Mustafizur Rahman et al. Often, in Enterprise grids, the availability of the resources varies widely. Their work focuses on finding a way to predict the availability of resources. To accomplish that they utilize the Jaccard Coefficient. The availability is represented by a sequence of zeros and ones, where zero represents unavailability and one represents availability. Their algorithm takes as input a sequence like that, which contains historical data of availability. They define a window of a number of elements, for which they will compute the Jaccard Coefficient each time. They begin from the oldest data and slide to the most recent one. Before each slide they try to make prediction of the next element. Finally, they use the statistics they have amassed to make a prediction on future elements. In the end, they test their method by comparing it with other established techniques. The experimental results show that their approach achieves better prediction accuracy with reduced complexity.

Another related work is presented in [4] by Foteini Alvanaki et al. with enblogue. They focus on blog posts and tweets published in social media. These posts are annotated with tags that describe their content. What their approach does is to calculate statistics about tags and tag-pairs that these documents are marked with. More specifically, the goal is to measure tag Correlations. The formula they use to compute the correlation of a pair of tags is $C = \frac{|T_1 \cap T_2|}{|T_1 \cup T_2|} + \frac{|T_1 \cap T_2|}{N}$. Where t_1, t_2 are tags, T_1, T_2 are sets of documents annotated with these tags and N is the total number of documents. The basis of this formula is the Jaccard Coefficient which is actually the first part of the equation. These statistics are then used to identify unusual shifts in correlations. Emerging topics are then being identified by ranking tag-pairs based on the strength and unpredictability of these shifts.

There is also work that tries to tackle the problem of computing Jaccard Coefficient in a distributed fashion. Papers presented below are closer to what we are trying to accomplish with our work.

Again, Foteini Alvanaki and Sebastian Michel in another work [13] they utilize Jaccard Coefficient to compute correlations between tags. This time, they extent their work to not only tag-pairs, but to an arbitrary number of tags. More specifically, they consider a stream of set-valued attributes $S_i = \{a_1, a_2, a_3..\}$, which they take as input. In the case of twitter, S_i would represent a set of tags or hashsets, as twitter users likes to call them. To perform computations faster, they parallelize the work-flow across multiple nodes. They consider two different stream processing platforms to accomplish this task . The first is the S4 [14] and the second the Apache Storm [5]. The goal is to genarate K covers, as many as the nodes of the distributed system, in order to partition the load to them with as little overlap as possible. They devise an algorithm that creates these covers, while at the same time, they make sure that nodes have enough information to compute the Jaccard Coefficients of all sets of tags. Finally, they use Twitter's streaming API to harness data from twitter to test their application.

In [1] Foteini Alvanaki and Sebastian Michel expand their work further. They take as input posts from Social Media like twitter and weblogs. These posts are annotated with a set of tags. Each time, they focus on the most recent ones and compute the similarity of the tags in any subset of co-occurring tags. In order to calculate these correlations they use the Jaccard coefficient. Due to the vast amount of incoming data, they devise a distributed model to distribute the computation of the Jaccard coefficients to multiple nodes. They assign a set of tags in each node. Each node is then responsible to compute the Jaccard coefficient for all sets of co-occurring tags in the power set of the set assigned to it. They face many challenges while trying to effectively materialize this model. First, all Jaccard Coefficients should be able to be computed, thus all co-occurring tags must be assigned to some node. Secondly, in order to avoid duplicate computations, assignment of the same tags to multiple nodes is to be kept minimum. Finally, the last challenge is to maintain the load of nodes balanced and without much disparity. To overcome these challenges they examine various partitioning algorithms, first from a theoretical perspective and then from practical standpoint. To evaluate the partition algorithms and the application as a whole, they implement their model using the Apache Storm Framework. They, then, put this prototype to test and present the outcomes. The most important of these outcomes is that their idea is proven doable even when working with high rate sources.

4. PROPOSED SCHEME

As described in the Introduction Chapter (chapter 1) our end goal is to find an efficient way to calculate correlations between co-occurring tags that appear in posts published in Social Media, and especially twitter. In this chapter, we look at the related work in a more critical way. First, we outline and explain some of the flaws of previous work. Next we describe how and what we are trying to accomplish with this work. Finally, we take a look at some challenges that we faced.

4.1 Drawbacks of previous work

After a thorough consideration of previous related work, the current work uses as its base the model presented in [1]. Foteini et al efficiently compute the Jaccard Coefficient even when the incoming rate is high (i.e. twitter). However, social media and especially twitter are volatile in nature, therefore the repartitioning algorithm had to be run repetitively in a very sort time period. This is far from ideal since repartitioning is a very laborious work. The entire set of tag-sets that have arrived need to be analyzed again. This work is trying to avoid that, making it faster and more adaptable.

4.2 A more flexible implementation

In the light of this setback, the problem has been approached in a more flexible and modular way. The proposition is to not repartition the whole input once a node has been overloaded, but instead make limited, yet targeted, changes one at a time. These changes will move small chunks of load from one node to another. Naturally, the flow of these changes will be from nodes with more load to nodes with less node. While these changes take place, only the nodes involved are aware of them. All other nodes are oblivious of them, hence, they continue to do their work like usual.

We aim more at creating a synergy between nodes, without them relying too much on a central node. The plan is to have as less variety of nodes as possible, forming this way a group of peers that do all calculations, while cooperating with each other.

The image 3 shows an example of such a schema. The organizer should perform organizational tasks that cannot be performed without global knowledge, but these should be kept as minimum as possible. Therefore, a way must be devised for each node to maintain all the necessary information to perform, at least, the basic operations. These operations are to:

1. Calculate correlations.
2. Move load from one node to another.

Our thesis is that this is a more efficient way to distribute load and do calculations. In other words, we believe that it is faster to make small adjustments in the distribution of the load rather than redistribute it all again. In the following chapters, we present a more specific

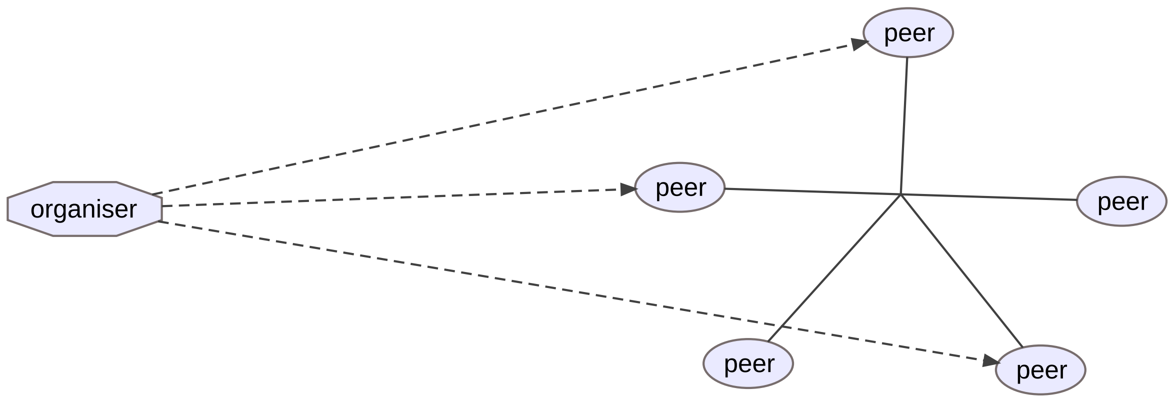


Figure 3: General Schema

model based on that concept.

4.3 Challenges

The most obvious of the challenges is that the application must be built in a certain way as to withstand streams of messages coming from high rate sources. This issue is tackled by using the storm framework and distributing the load in a collection of machines. Yet, this introduces the issues of load balancing and cooperation overhead. In fact, these are the most difficult obstacles we had to overcome. However, dividing the load evenly and minimizing redundant information optimizes resource use and maximizes throughput. Therefore, they were the most important considerations we had, while we were designing the application.

5. DESIGN

With the flaws of previous work in mind and with a clear goal as to what we are trying to accomplish we set forward to try to design a viable solution. The designing process is divided in three parts. First is the designing of a graphical representation of all incoming messages and the relations between them. Secondly, we define what similarity measures will be used to compute correlations and how they fit to the whole picture. Finally, the complex issue of parallelization is discussed at the end of the chapter.

5.1 The Graph

The application is built around the concept of a graph containing all data which is essential for not only computing correlations, but distributing load as well. Apart from maintaining data, graph's structure helps performing all appropriate actions faster and efficient. In order to accomplish that, it maintains one type of vertices and two types of undirected edges. Consequently, one can say that it is effectively two graphs with the same vertices, but different edges. For the sake of simplicity, though, it is considered as one graph for the remaining of the thesis.

5.1.1 Vertices

Each vertex in the graph corresponds to a tag-set. Vertices are weighted and the weight denotes how many documents (posts) have been seen annotated with the tag-set up until now. It is important to note that one tag-set corresponds to only one vertex. In other words there are no duplicates vertices.

At any time, the graph contains an arbitrary number of vertices/tag-sets. Some of these tag-sets are subsets of other tag-sets in graph, while some others are not. Tag-sets for which a superset does not exist in the graph are treated specially. We will call these uppermost supersets as *head tag-sets* or simply *heads* for the rest of the text. Later in this text, it will become clear why these are treated differently.

5.1.2 Edges

To relate tag-sets with each other, two kind of edges are used and sustained throughout the lifetime of the graph. Both kind of edges are undirected and weightless and the presence of one does not exclude the presence of the other.

Set Relation Edges

The first type of edges describes set relations between tag-sets of vertices. If a tag-set is a subset of another existent tag-set in the graph and the two differ by only one tag, the two of them are connected by an edge in order to mark this relationship. From now on subset relations in figures and diagrams are signified by dashed lines. You can refer to figure 4 to see an example of this edge type. As you can see in this figure, vertex {a,b,c,e}

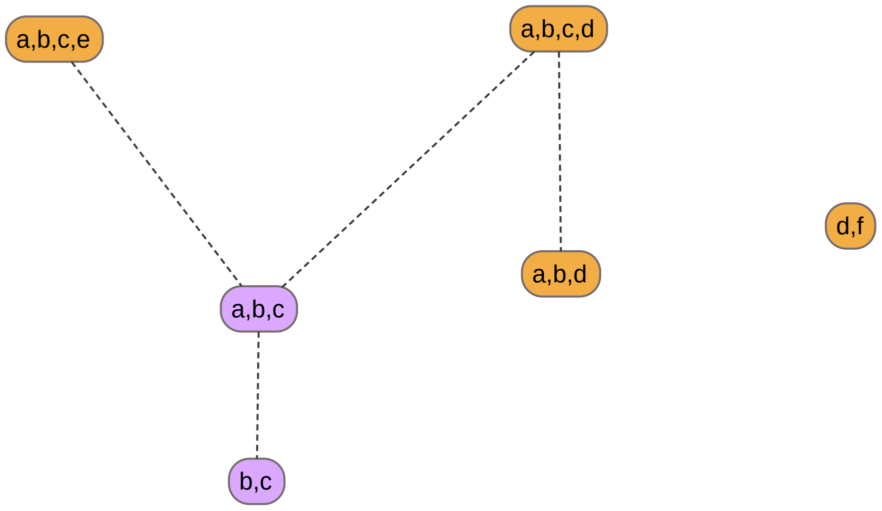


Figure 4: Set relations edges

is connected to the vertex {a,b,c}, but not to the vertex {b,c}. This is because {a,b,c,e} has two more tags compared to {b,c}. Also, the vertex {d,f} does not have any relation to the other ones, so it is not connected with any other vertex.

It is not uncommon for two tag-sets to have common subsets. Hence, any tag-set can have an arbitrary number of connections with subsets and supersets. In figure 4 vertices {a,b,c,d} and {a,b,c,e} tag-sets have two common subsets which are colored in purple.

Common Tags Edges

Two vertices are connected with a *common tags edge* if and only if they represent two tag-sets that contain at least one common tag. Common tags connections are used not only for computing correlations, but for other auxiliary operations as well, which will be discussed later. In figure 5, one can see the same graph as in Figure 4, but now the common tags edges have been included.

It is worth mentioning that the two types of edges are not mutually exclusive. In fact, whenever two vertices are connected with a *set relation edge*, they are also connected with a *common tags edge*. However, a common tag edge does not imply a set relation edge. This might seem like an excess of information, yet set relation edges are necessary to refer to subsets or supersets, instantly. If only common tag edges where there, we would have to check whether a connected tag-set is also a subset/superset or not and that would decrease performance.

In figure 6, we present what a complete graph looks like, based on the structure of figures 5 and 4.

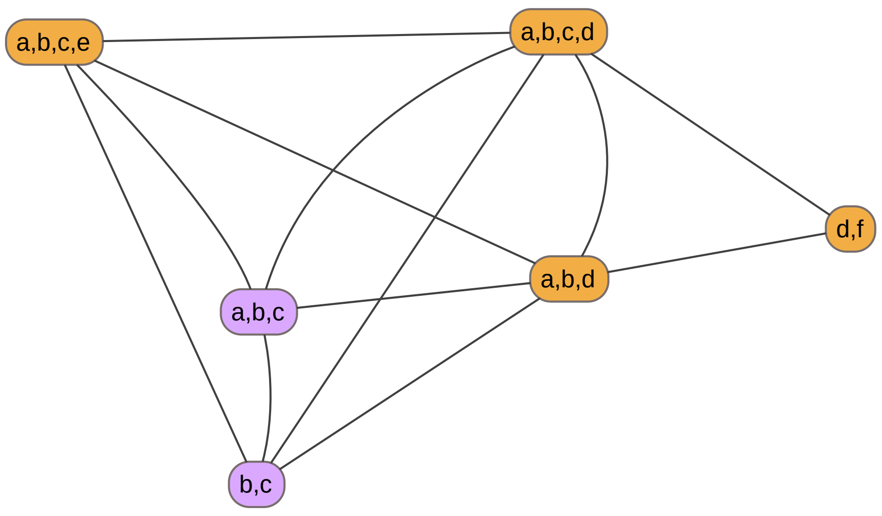


Figure 5: Common tags edges

5.2 Computing Jaccard Coefficient

In this paragraph, we are going to start with the general formula of Jaccard Coefficient and convert it to a form which fits to the graph model presented in the previous paragraph. The general formula for computing the Jaccard Coefficient is formula 1. You can refer to background chapter (2.3.1) is you are not accustomed with the method.

$$J(X_1, X_2, ..., X_n) = \frac{|\cap_{i=1}^{n} X_i|}{|\cup_{i=1}^{n} X_i|} \tag{1}$$

A vertex in the graph counts documents annotated with a specific tag-set, thus equations in this chapter consider as vertex (v) the set of documents annotated with this tag-set.

$$v_i = \{d_1, d_2, ..., d_n\}$$

t_i	A single tag.
T_i	The set of documents annotated with t_i tag.
s_i	A set of tags.
v_i	The set of documents annotated with s_i.
C_i	The set of vertices connected with a *Common Tags Edge* to v_i
w_i	The wight of vertex v_i, i.e. the number of documents annotated with s_i.
S_i	The set of vertices that correspond to a tag-set which is a superset of s_i.

Each vertex is unique, there are no duplicate vertices and every tag-set corresponds only to one vertex. Therefore $v_i = v_j \Leftrightarrow s_i = s_j$.

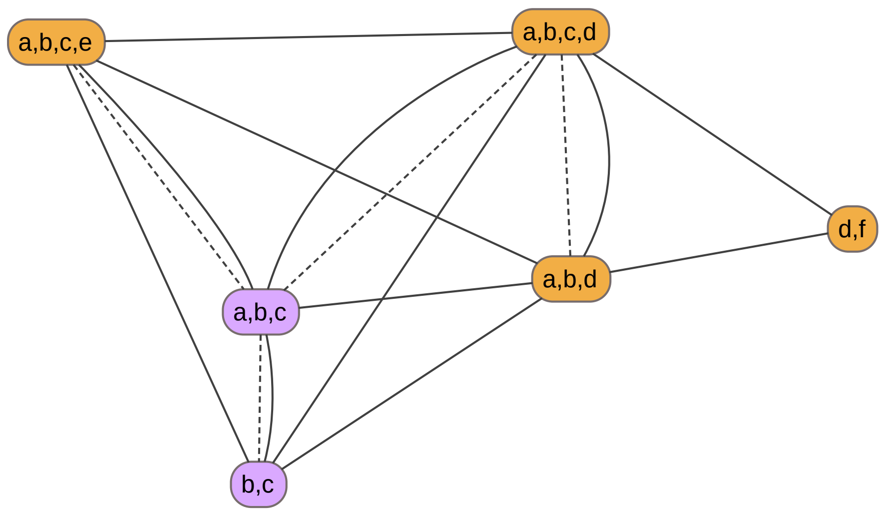


Figure 6: example graph with all edges

Two vertices are connected if their corresponding tag-sets have common tags.

$$v_j \in C_i \Leftrightarrow s_j \cap s_i \neq \emptyset, j \neq i \tag{2}$$

Based on formula 1, in order to calculate the Jaccard Coefficient of tag-set $s_i = \{t_1, t_2, ..., t_n\}$ the formula 3 is used.

$$J(T_1, T_2, ..., T_n) = \frac{|\cap_{i=1}^{n} T_i|}{|\cup_{i=1}^{n} T_i|} \tag{3}$$

The intersection $\cap_{i=1}^{n} T_i$ denotes all documents which are annotated with all of the tags in s_i. In other words, all the documents that are annotated with either s_i or a superset of s_i. Therefore, $|\cap_{i=1}^{n} T_i| = \sum_{v_j \in S_i} w_j + w_i$

The Union $\cup_{i=1}^{n} T_i$ denotes all documents which are annotated with any of the tags in s_i. In other words, all documents that are annotated with either s_i, or any tag-set having common tags with s_i. Hence, $|\cup_{i=1}^{n} T_i| = \sum_{v_j \in C_i} w_j + w_i$

Finally, the JC of a tag-set s_i can be computed using the following formula:

$$J(T_1, T_2, ..., T_n) = \frac{\sum_{v_j \in S_i} w_j + w_i}{\sum_{v_j \in C_i} w_j + w_i} \tag{4}$$

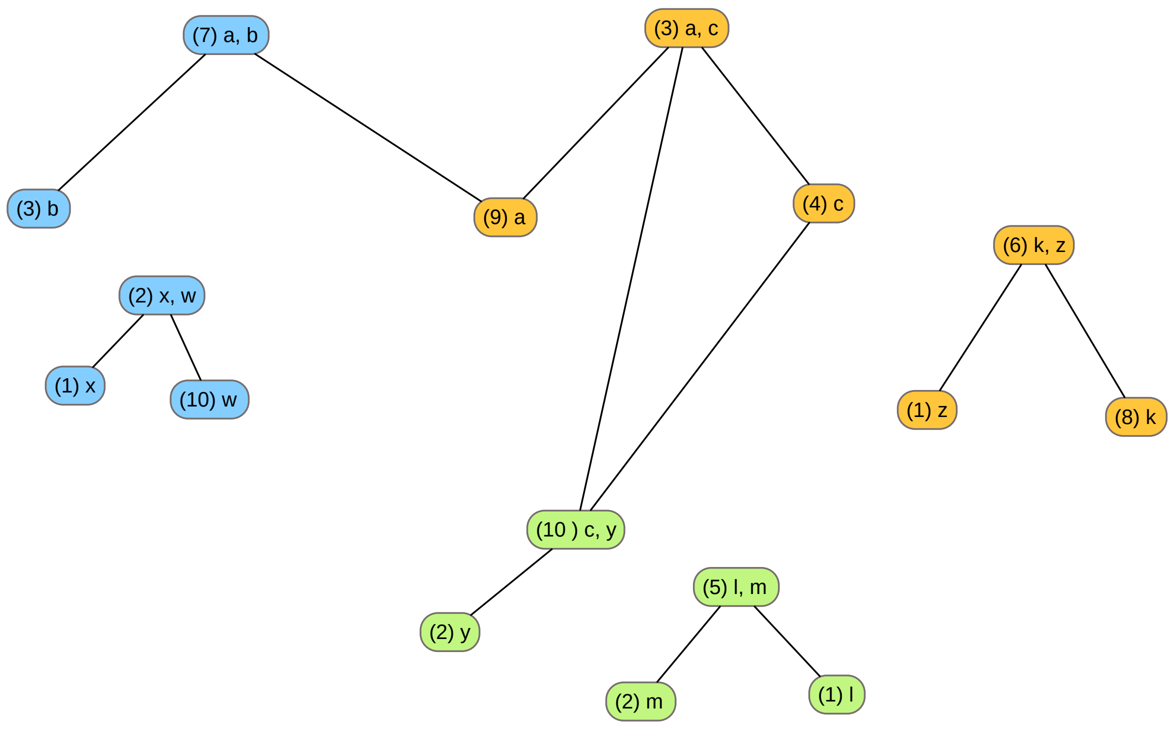


Figure 7: Example of a graph of tag-sets

5.3 Parallelization

The general idea is to distribute the computations of the coefficients to multiple machines (nodes). To accomplice that, each node must sustain part of the data and make part of the calculations. First, we must find a viable solution to make this partitioning happen. By doing so, however, we should be extra careful as to not create imbalances between nodes. These are the concepts discussed in this chapter.

The model devised to comply to these constraints, is the following. Each node will hold a segment of the whole graph and subsequently, receive only a share of all the documents. To demonstrate how the partitioning works, we will begin from a complete graph. A simple instance of a graph is seen in figure 7. Vertices belonging to different nodes are annotated with different colors. Here the graph is pictured before the segmentation, as if only one node was holding all data. Only *common tag edges* are shown in the graph. Numbers inside parentheses represent the weights of the vertices i.e. how many documents have been annotated with the corresponding tag-set so far.

If one was to compute the JC of tag-set {c,y}, they would need all tag-sets with at least one common tag. Based on the example graph, formula 4 and assuming that $w(v)$ symbolizes the weight of a vertex *v*, the JC of {c,y} would be the following:

.

$$
\begin{aligned}
JC(\{c,y\}) = \\
\frac{w(\{c,y\})}{w(\{c,y\}) + w(\{y\}) + w(\{c\}) + w(\{a,c\})} = \\
\frac{10}{10+2+4+3} = \\
0.526315789
\end{aligned}
$$

Let us try to segment the graph of figure 7. If we naively, assign to each node the vertices of a specific colour, then we would end up with what is pictured in figure 8. In this case green node would be oblivious of vertices {a, c} and {c}. Therefore the JC it would computed like:

$$JC(\{c,y\}) = \frac{w(\{c,y\})}{w(\{c,y\}) + w(\{y\})} = \frac{10}{10+2} = 0,833$$

In practice, graph segmentation, in some cases, leads to redundancy of data. This is a direct consequence of Jacard Coefficient requiring all occurrences of all tag-sets which have common tags. Figure 9 shows all vertices each node has to maintain after the segmentation, based on the example of figure 7. On this occasion the duplicate vertices are the {a}, {c, y}, {a, c}, {c}, which appear in more than one node.

5.3.1 Redundancy

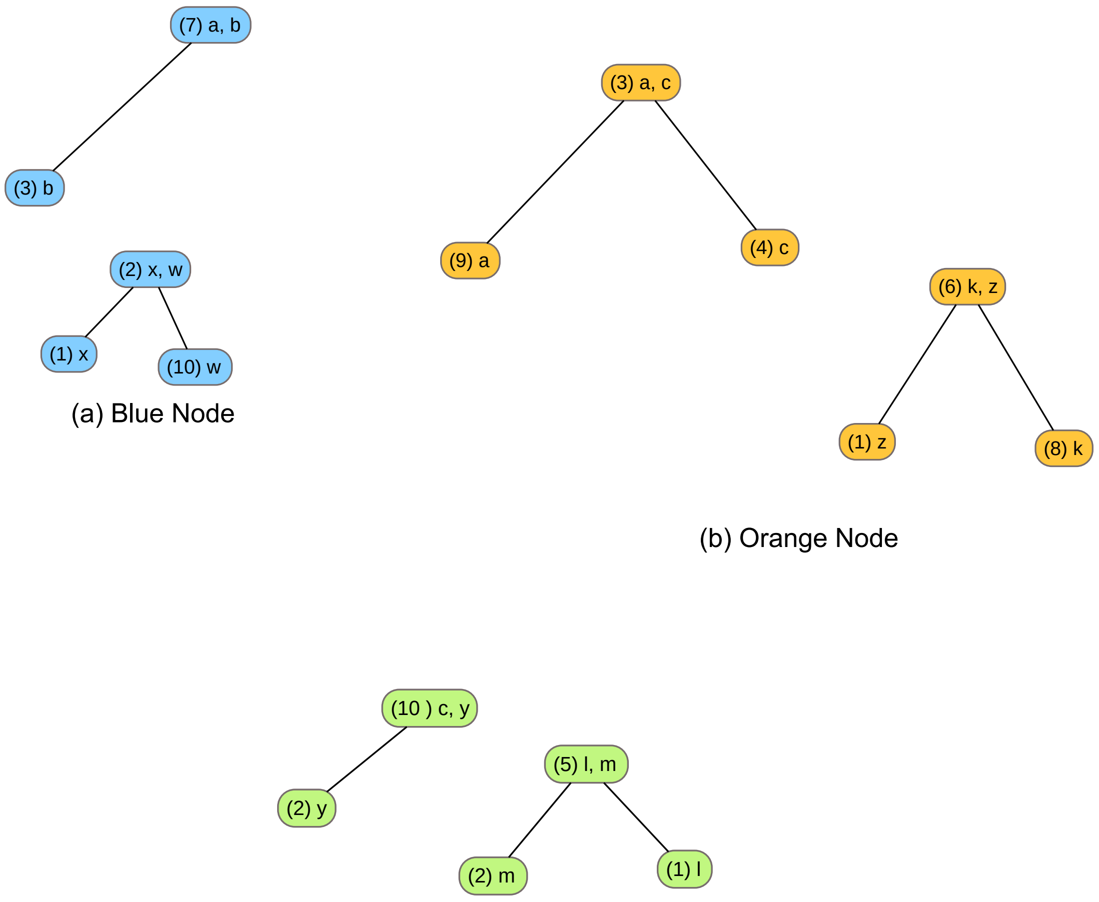
We will now take a closer look in the phenomenon of redundancy in nodes. We start with defining a way to quantify and measure it. Afterwards, we give a more detailed example.

As Redundancy between two nodes we define the number of the same documents that they both hold. In other words, redundancy is the sum of the weights of the vertices they both maintain. It is introduced whenever edges of the graph connect two or more vertices that belong to different nodes. More formally, if *V* where the vertices of *node* 1 and *V′* where the vertices belonging to *node* 2, then the redundancy *R* between these to nodes would be:

$$R = \sum_{v_i \in (V \cap V')} w(v_i) \qquad (5)$$

Take for instance the example of figure 9. In this context, redundancy between green node and orange node, based on equation 5, would be:

$$R = w(\{c,y\}) + w(\{c\}) + w(\{a,c\}) = 10 + 4 + 3 = 17$$



(a) Blue Node

(b) Orange Node

(c) Green Node

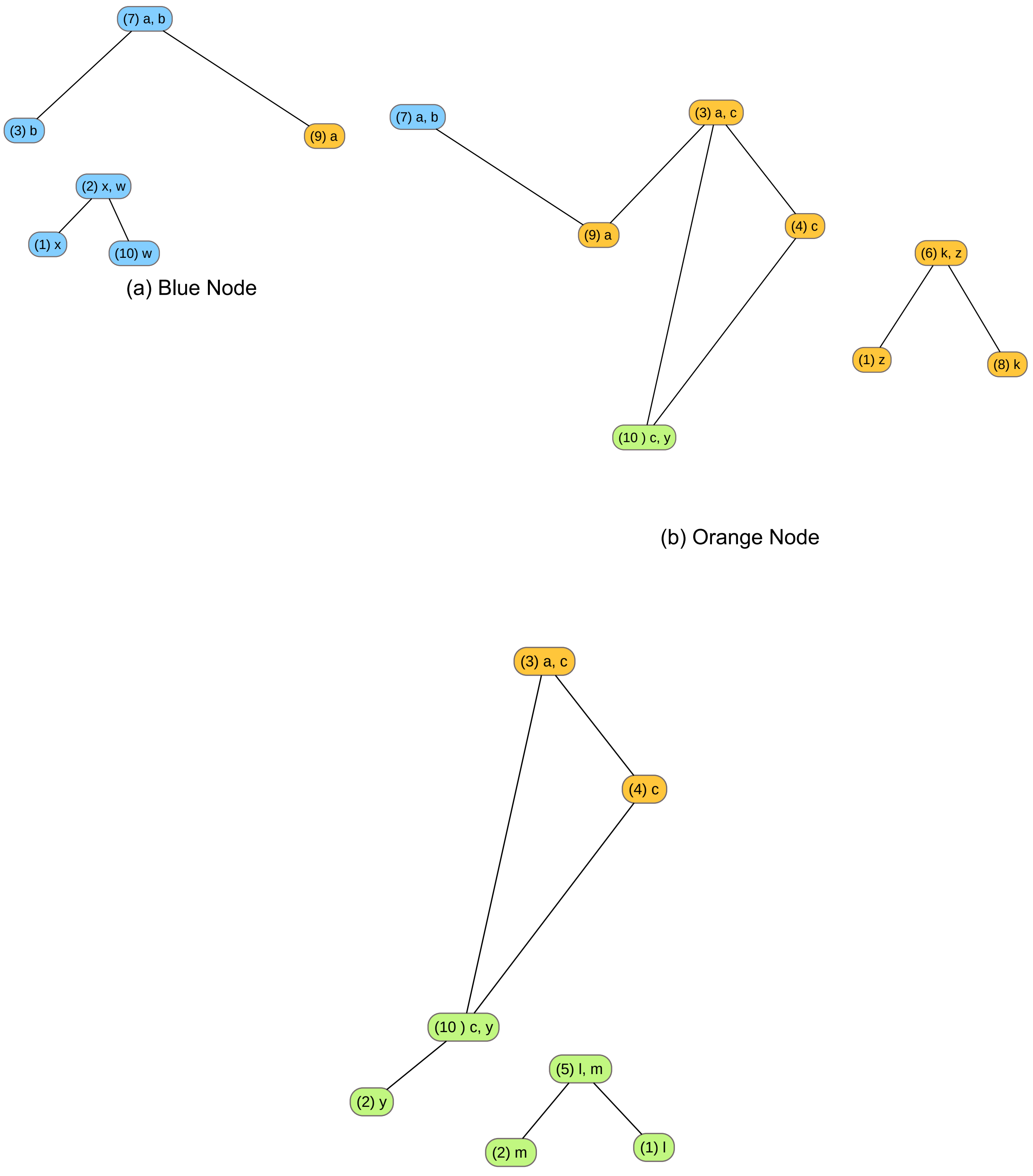
Figure 8: Wrong Graph Segmentation

In the final analysis, one of our goals should be to reduce redundancy whenever possible. Otherwise, it could lead to duplicate work being done across nodes, and thus, to decreased performance. In a later chapter, we discuss what can be done about this issue.

5.3.2 Load Balancing

In every distributed system, the load is monitored closely. After all, decreased performance and bottlenecks could appear if the majority of load is directed only to few nodes. In this paragraph, we discuss how we avoid load disparities.

Load's unit is defined as one document annotated with a tag-set. Hence, the total load of a node is the total number of documents that are being forwarded to it. Now, we must also define a way to balance load across nodes. This is done by defining a limit which any node, at any time, must not exceed. This limit is dynamically calculated during execution, although its parameters are predefined. These parameters can be set so that the load is as much balanced as the user desires. At first, it might seem pointless to predefine how much you want the load to be balanced between nodes, since the easy answer would be



(a) Blue Node

(b) Orange Node

(c) Green Node

Figure 9: Correct Graph Segmentation

as balanced as possible. However, as stated in paragraph 4.3 it is hard to keep the load balanced while minimizing duplicate data at the same time.

Simply put, the more balanced the load is, the more overhead and data redundancy the application suffers. To illustrate this, imagine that no constrains were imposed on how much the load was balanced. In this case, each node could hold a segment of the graph that was not connected to any other segment, despite the fact that some segments could be substantially bigger than the others. This would effectively eliminate any redundancy. On the other hand, if we would want to balance the load we would transfer vertices from one node (or segment) to another, thus creating duplicate data across nodes.

With these concepts in mind, the formula to examine whether a node is overloaded or not is now defined. In every case, the load of a node is compared with the load of other nodes. The first step is to calculate the average load and then compare the load of each node with it. If a node exceeds a predefined difference between its load and the average load, it will then be considered as overloaded. The average load is computed by simply dividing the total load by the number of nodes.

Formally, the formula used to determine if a node has become overloaded is the following:

$$\textit{average load} = \frac{\textit{total load}}{\textit{number of nodes}}$$

$$\textit{load threshold} = \textit{average load} + \textit{overload factor} \times \textit{average load} \tag{6}$$

Where *overload factor* is a real number, smaller than one, which indicates the percentage of tolerated variance from the average load. For instance, if $\textit{total load} = 3000$, the number of active nodes is 10 and the *overload factor* variable is $0,20$ then:

$$\textit{average load} = \frac{3000}{10} = 300$$

$$\textit{load threshold} = 300 + 0,20 \times 300 = 360$$

The optimal value of *overload factor* is tricky to be found and varies from implementation to implementation. Some factors that could affect it, are the number of nodes, the connectivity of the graph and so forth. The only way to really find a good value to set the *overload factor* is by extensive testing.

In any case, however, when a node becomes overloaded, what should be done, is to move some of its vertices to another node, thus reducing its load. Which vertices should be moved and when is left to the implementation process.

6. IMPLEMENTATION

A prototype system has been implemented with the purpose of testing our hypothesis. The Design Chapter (Chapter 5) was used as a compass that guided the construction of an application that efficiently computes the jaccard coefficients. At the beginning, we show how the graph is represented by the application. Afterwards, we display the topology of nodes and explain each node type. Then, we describe how nodes are communicating with each other and also the sequence of actions performed in order to calculate the Jaccard Coefficients. At the last section, we present technical informations of how we tested our application.

6.1 Graph Representation

In order to implement the graph a new simple data-type has been created and it is called Vertex. On the same lines as its theoretical counterpart, each vertex corresponds to only one tag-set. Vertex types hold five crucial variables.

1. A *counter* which counts how many posts have been seen annotated with the tag-set in question.
2. The node who is accountable for calculating the Jaccard Coefficient of it.
3. A list of tag-sets with which it is connected with a *common tags edge*.
4. A list of proper superset tag-sets that have only *one* more tag.
5. A list of proper subset tag-sets that have only *one* less tag.

Vertices together with a map which maps tag-sets to vertices are all that is needed to model the graph. Additionally, there are a few more constrains imposed in the graph to help organizing it a bit more. The first one is that if a tag-sets is on the graph then all its subsets are on it as well. The second one is that subset tag-sets have the same owner as their supersets, effectively making head tag-sets the ones that define where their whole tree of subsets belongs to. These are rules that are kept throughout the execution of the application.

6.2 Topology

The topology of nodes is a simple one, with only three different node types. One parser node, one disseminator node and an arbitary number of plotter nodes. The bulk of computations is performed by plotters which is the main node of the application. The simple topology comprised mainly by plotters furthers supports the idea that the computations are performed by equal peers that self-organize themselves.

Figure 10 displays the complete image of an example topology with three plotters. Arrows signify with which other nodes, each node communicates and in which way.

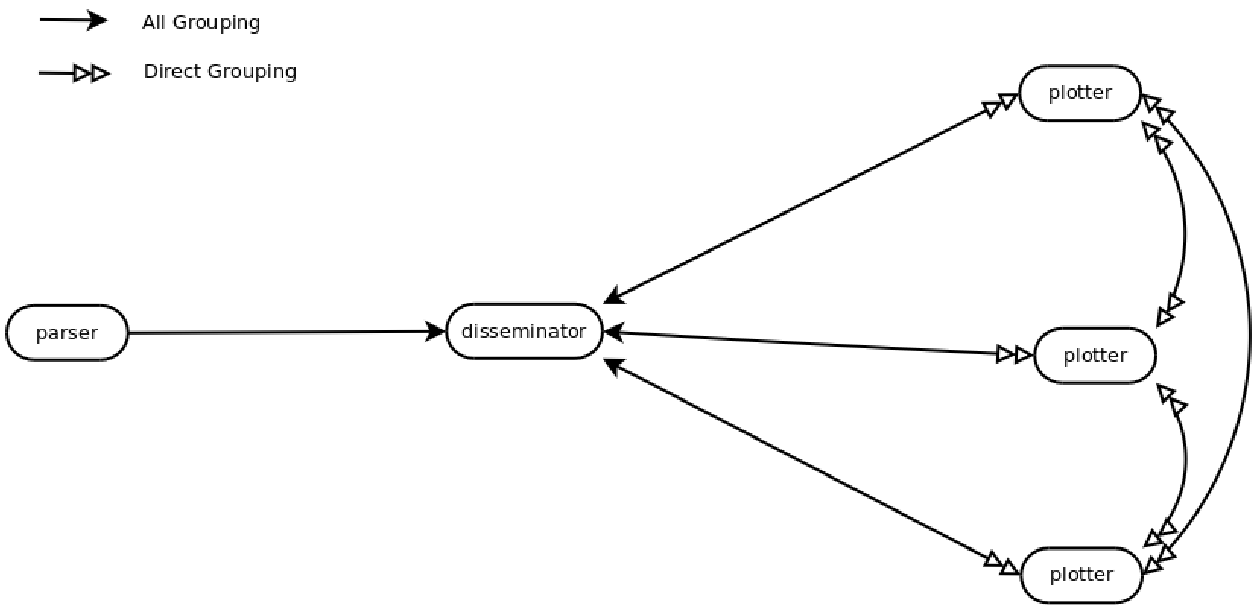


Figure 10: Storm Topology

6.2.1 Parser

Parser is the only *spout* node of the topology. It is responsible of parsing messages and emit them into the topology in a proper format. Every time a document (e.g. tweet, post) is produced by a media (twitter, on-line newspaper etc.), the parser strips the text and keeps only the set of tags.

The tag-set is then reformed. Individual tags are modeled as strings and the whole tag-set as sets of strings. For instance tag-set {#smoking, #health, #children} would be a set of strings with values ”#smoking”, ”#health”, ”#children”.

6.2.2 Disseminator

The first *bolt* of the topology is the disseminator node. Disseminator is a node of great importance, which handles all the organizational work that is not performed by the plotters themselves. Its main responsibility is to forward tag-sets to the appropriate Plotters. To manage that the disseminator consults an index that maps tags to plotters. when a tag-set arrives the disseminator pushes a tuple containing the tag-set to all plotters the index indicates. Furthermore, the disseminator is in charge of examining the load of plotters. If a plotter has become overloaded, disseminator notifies the plotter, so it can take actions to reduces its load. This is essential in order to keep the load of plotters balanced. Last but not least, whenever a tag-set that has not been previously seen, arrives, the dessiminator is the one responsible of deciding which plotter will be accountable for it from now on.

To successfully perform these tasks, disseminator sustains the following data-structures.

1. A collection of active tag-sets that are presently inside the graph.
2. An index from tags to lists of plotters, which is advised before forwarding tuples to plotters

3. A map from plotters to the number of tag-sets that they have received i.e. the load of each plotter.

These data-structures are used by the disseminator to efficiently and accurately accomplish the tasks it is responsible for. The most important one is the index from tags to plotters. A plotter is present on a list corresponding to a specific tag if and only if it is responsible for a tag-set that contains this tag. If we take the example of figure 9c, the list of plotters in the index, for tag ”a”, will not contain the green plotter. However, the green node will be in the list for the ”c” tag. Hence, it will receive documents annotated with the tag-set {a, c}, for instance.

6.2.3 Plotter

Plotter *bolts* realize the core functionality of the application. Each plotter maintains a segment of the whole graph. Therefore, Plotters are in charge of keeping information for only a subgroup of all tag-sets, which then utilize to do all calculations needed.

Their main responsibility is to calculate and emit Jaccard Coefficients. At regular intervals of time, they employ all data that they have gathered and compute the coefficients of the tag-sets for which they are responsible. All the tag-sets that are necessary to make these calculations are being forwarded to them by the dessiminator. It is their task to keep a counter for every tag-set.

Moreover, plotters might need data that other plotters hold. Hence, another responsibility of plotters is to provide information to whichever of their partners asks for it.

Plotters can also receive messages from disseminator, which notify them that they have become overloaded. They are the ones in charge of deciding which vertices should be moved to which plotters, in order to reduce their excessive load. Afterwards, they wrap up all the necessary information and update all other plotters that are affected by these changes.

Finally, moving vertices from one plotter to another, can potentially affect which tag-sets should be forwarded to which plotters. Any such event is detected by the plotters immediately. They then send messages to disseminator to alter its index. This way the disseminator’s index is kept sound and updated at all times.

Plotters maintain the following data-structures to aid them accomplish their tasks.

1. An index from tag-sets to vertices.
2. An index from tags to tag-sets.
3. A list of head tag-sets.

The index from tag-sets to vertices is in the core of the implementation. It effectively models the graph presented in the design chapter (section 5.1). From a theoretical perspective, it is actually an adjacency list that saves, for every vertex(tag-set), which other vertices are adjacent to them. One key difference is that it is backed up by a hash table to make access faster. You can refer to section 6.1 for more information on vertex data type.

Besides the tag-sets to vertices index there is another index. An index from tags to tag-sets that maps tags to all tag-sets that contain that specific tag. This index helps connecting new tag-sets with existing ones.

Finally, plotters keep a list of all head tag-sets inside their graph segment. As stated in 6.1, subset tag-sets have the same owner as their supersets, therefore only head tag-sets are considered for relocation to other plotters and their subsets follow them. For this reason it is important to know which are the head tag-sets, at any time.

6.3 Processes

A process is a systematic series of actions directed to some end. In this chapter we describe the processes performed towards accomplishing the final goal, which is of course the computation of Jaccard Coefficients. Some of these actions are simple and some more complex. They could involve one or more nodes and one or more node *types*. Moreover, more than one streams could be used to complete them.

In particular, the processes explained in this chapter are the following:

1. How the application treats incoming *existing* tag-sets.
2. How the application treats incoming *new* tag-sets.
3. How vertices are relocated from one node to another.
4. And finally, how the Jaccard Coefficients are calculated.

Before we go into further details about them, let us outline the streams utilized. Storm uses streams in a specific way. You can refer back to background chapter 2.2.1 if you find it difficult to comprehend the terminology used here.

The application uses many streams to transfer tuples from one node to another. The main reason is because messages from different streams are processed differently by nodes. Furthermore, different streams transfer tuples of different data-types and sizes, depending on the action required. Table 1 outlines all streams and the stream groupings which are used. Another table, table 2, displays the format of the messages transfered by each stream.

Information in these tables might be difficult to comprehend for now. However, do not worry if it is not clear yet how everything works. The paragraphs that follow will clarify the ideas introduced here.

Table 1: Streams and their groupings

Stream Name	Source	Subscribers	Stream Grouping
tagset	Parser	Disseminator	All grouping
tagset	Disseminator	All plotters	Direct grouping
disseminator_new_tagset	Disseminator	All plotters	Direct grouping
plotter_new_tagset	All Plotters	All Plotters	Direct grouping
overloaded	Disseminator	All plotters	Direct grouping
relocation	All plotters	All plotters	Direct grouping
change_owner	All plotters	All plotters	Direct grouping
index_update	All plotters	Disseminator	All grouping

Table 2: Tuple formats of streams

Stream Name	Tuple Format
tagset	(tag-set)
disseminator_new_tagset	(tag-set, plotter in charge)
plotter_new_tagset	(heads, tag-sets)
overloaded	(plotter loads, load threshold)
relocation	(relocated head, connected heads, connected vertices)
change_owner	(head, plotter)
index_update	(operation, tag)

6.3.1 Existing Tag-sets

The simplest and easiest of all processes is when the application receives a document which is marked with a tag-set which is already present in the graph. In this case, the disseminator forwards the tag-set to the appropriate plotters, based on its index. Then plotters increase their counter which corresponds to this tag-set by one. Figure 11 shows an example of messages exchanged between nodes during this process.

6.3.2 New Tag-sets

At times, a post arrives, which is annotated with a new tag-set. A tag-set not contained in the graph. Whenever that happens, the decision must be made as to which plotter will be responsible for it.

Disseminator's Side

New tag-sets should be assigned appropriately so as to minimize redundancy. An efficient way of choosing, is to assign them to the plotter already receiving the most of the new tag-set's tags. This way, the plotter node will not have to pull a lot of (if any) vertices from other plotters, because it will already have all vertices which are connected to the new tag-set. In case the new tag-set contains only new tags, though, it is handled differently. If it contains only tags never seen before, it means that it will not connect with any other vertex. Thus, it can be assigned to any node and to any graph segment easily. For obvious reasons,

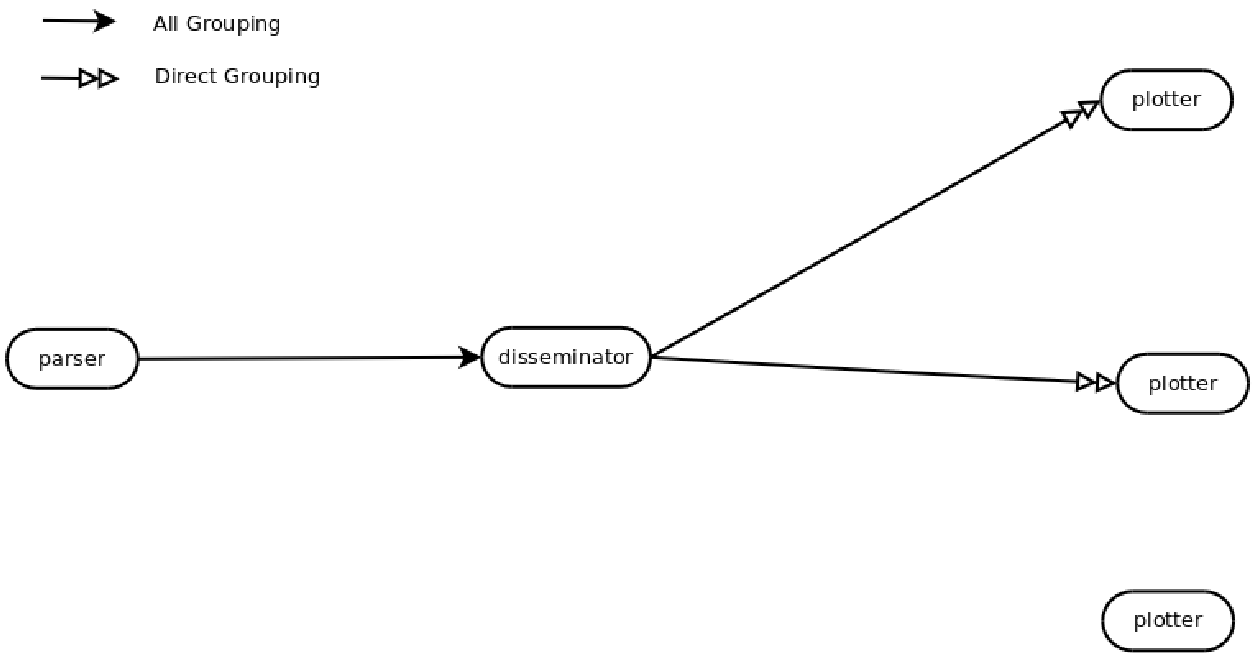


Figure 11: Forwarding existing tag-sets

the node with the least load is chosen. In all cases, though, the node which makes these decisions is the disseminator, with the aid of its index from tags to lists of plotters.

After the disseminator decides which will be the plotter in charge, it follows the standard procedure of forwarding it to all appropriate plotters, much like when an existing tag-set arrives. The difference is that now the destination plotters need to know which plotter has been decided to be in charge of it. Therefore the tuples forwarded have the following arrangement: (new tag-set, plotter in charge).

Please notice that subsets of the new tag-set are inserted into the graph as well, and thus, a new tag-set can only be a superset of those already in. Therefore, the last step for the disseminator is to produce all subsets and insert them, if not already there, into the *active tag-sets* list. This is done in order to not misinterpret future subsets as new tag-sets.

Plotter's Side

On the plotter's side, the new tag-set is inserted using algorithm 1. What this does is to insert the new tag-set together with its subsets (if not already in) and make all appropriate edge connections. When a plotter receives a tuple regarding a new tag-set the actions it takes differentiate slightly depending on whether or not it has been chosen to be the plotter in charge of it.

If a plotter is *not* responsible for the new tag-set, then data should be sent to the plotter which is responsible for it. The purpose behind this is because the latter might not have all vertices that the new tag-set needs to connect to. Therefore, other plotters provide all the connected vertices that they have in order for the graph to remain complete and sound.

Lets now examine the instance which a plotter takes a new tag-set tuple which *is* in charge for. In this case, the plotter does not do something special. It applies algorithm 1 and then

Algorithm 1: Add a new tag-set in the graph

```
   input: new tag-set, plotter
1  initialize stack;
2  push new tag-set into stack;
3  while stack not empty do
4      pop ts from stack;
5      foreach $s \subset ts$ and $s.size = ts.size - 1$ do
6          if s not in graph then
7              put s in graph;
8              connect s to tag-sets that have common tags;
9          end
10         set owner of s to plotter;
11         connect s as a subset of ts;
12         if $s.size > 1$ then
13             push s to stack;
14         end
15     end
16 end
```

waits for messages to arrive by other plotters. When such a message arrives, the plotter makes the appropriate changes to its graph. A diagram about the messages exchanged, during this process, can be seen in figure 12.

As discussed before, when a head tag-set belongs to a plotter, then all its subsets belong to the same plotter as well. Therefore, when a new tag-set arrives there is the possibility that subsets of it, that were previously head tag-sets themselves, will change owner.

A vertice changing owner is a form of relocation from one plotter to another. This may lead to some vertices no longer needed. For the graph to be sound and calculations to be performed faster these redundancies are removed. Hence, a clean up is performed that deletes all unneeded vertices. Possible index updates are also sent to the disseminator.

The way plotters determine if an index update is needed is by examining the tag to tag-sets index which they maintain. If the list of tag-sets regarding a specific tag contains only tag-sets for which the plotter is not responsible for, then an index update should be sent to the disseminator. Otherwise no action is taken.

A final aspect to be considered is that new tag-sets can cause a node to become overloaded quickly. This is because when a tag-set is assigned to a plotter, then the plotter becomes in charge not only for the tag-set itself but for all its subsets as well. Issues arise whenever subsets of the new tag-set, which other plotters are responsible for, are pulled to the plotter, which is responsible for the new tag-set. With the way the plotter in charge is chosen, however, this is rarely an issue.

The following example will demonstrate this. Assume that the new tag-set {a,b,c,d,e} has just arrived. There are only three plotters: plotter1, plotter2 and plotter3. And the

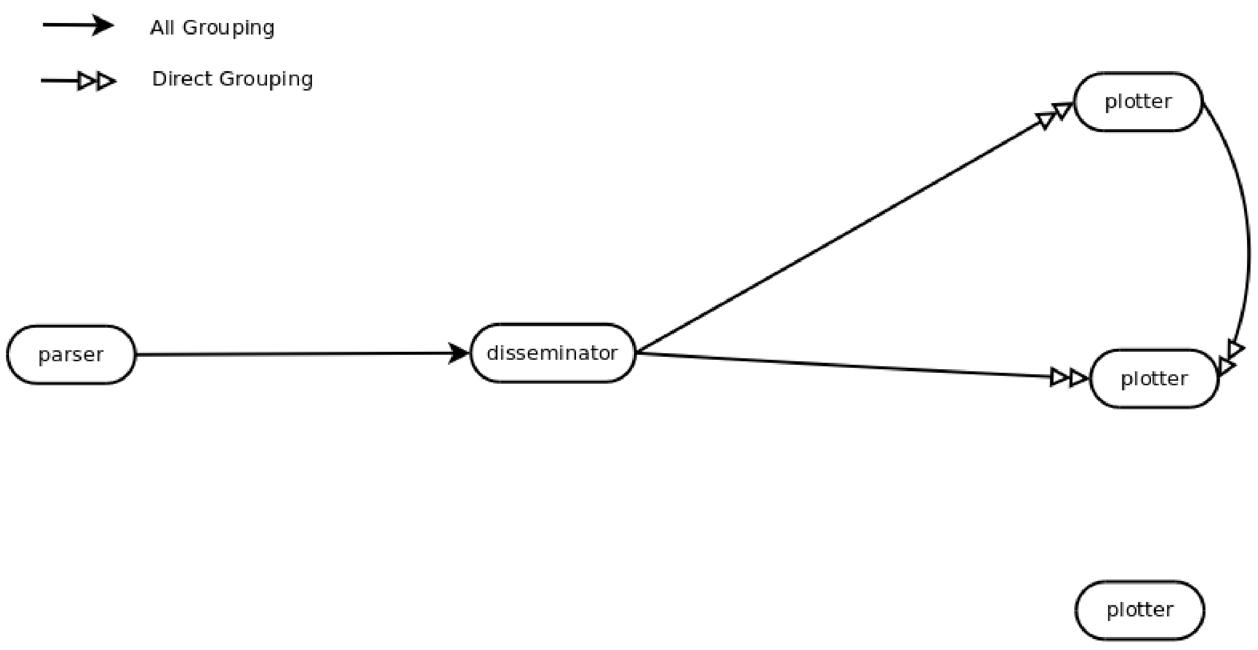


Figure 12: Forwarding new tag-sets

disseminator's index from tags to plotters is the following:

a $\rightarrow$ plotter1, plotter2

b $\rightarrow$ plotter1

c $\rightarrow$ plotter1, plotter2, plotter3

d $\rightarrow$ plotter1, plotter3

e $\rightarrow$ plotter2, plotter3

The plotter that will be in charge of the new tag-set will be plotter1, since most of the tags already point to it. What that means is that plotter1 already gets tag-sets annotated with a, b, c, and d. Therefore, it already gets most of the tag-sets that will be connected to the new tag-set. Plotter1's load will not change much, even after {a,b,c,d,e} subsets change owner to plotter1.

6.3.3 Relocating vertices

In order to have balanced load across plotters, whenever a plotter becomes overloaded, it moves (relocates) one or more of its vertices to a plotter with less load. For this to happen, synergy between disseminator and the overloaded plotter is required.

Disseminator's Side

Disseminator node keeps a counter for each plotter that counts how many tag-sets have been forwarded to it. This way the disseminator is aware of plotters' load. After a specified time interval, it examines whether any plotter has become overloaded. From here on, we will call this interval as *load check window*, and use this term to refer to it. If any of the plotters is overloaded disseminator notifies the plotter of its overloaded state. After that, the disseminator does not have to perform any more actions. The overloaded plotter is responsible to reduce its load on its own. To determine if a plotter is overloaded the formula 6 described in chapter 5.3.2 is applied.

Only one overloaded message is sent at a time, on the grounds that synchronization would be extremely difficult otherwise. A lot of nodes trying to relocate vertices could cause data loss and even graph inconsistencies. The Disseminator uses a separate stream to inform the plotters that they are overloaded. The tuple messages that are sent contain only a map from plotter ids to plotter loads. Whenever a plotter receives a message from the overloaded stream, then it automatically knows that it is overloaded.

Plotter's Side

After receiving this message, the overloaded plotter is responsible to relocate one or more of its vertices to another plotter, as a means of reducing its load. The first thing it does is to decide which vertex (or vertices) to relocate to which plotter(s). The decision is taken on the basis of reducing the redundancy between plotters.

Let us take the example of figure 9. Assuming that the overloaded node was the green one, it will choose to move tag-set {c,y} to orange node, rather than {l,m}. This way the need for the green node to maintain data for tag-sets {a,c} and {c} is eliminated. Hence, the redundancy is reduced. The Algorithm used to calculate, how much the redundancy overhead will change if a move is made, is algorithm 2.

Every time a vertex is moved from an overloaded plotter to another plotter, the second plotter is in danger of becoming overloaded too after the relocation. Hence, the overloaded plotter should be careful not to overload other plotters, since this would only propagate the problem. This is when the loads of other plotters sent by the Disseminator are proven useful. After a possible relocation has been identified, a second check is performed on destination plotter's load and if the action will overload the plotter, then the action is canceled. Instead, the second best move is considered. if this action will again overload a node then the third best option is considered and so on.

After every move, the disseminator is informed for any possible index updates. Plotters connected, through their vertices, with the selected vertex are informed of the change in ownership as well. Figure 13 displays the messages exchanged during the relocation process.

Be aware that, if more than one vertex needs to be moved, the plotter does not consider them as a group. In other words, they are chosen and moved one by one. If all groups of vertices were considered as a whole before relocating them, the decision might be different. However, that would require to test the relocations of all groups belonging to the power set to the set of vertices a plotter has, thus, it would require a lot more computational time to complete.

Algorithm 2: Calculating overhead change

```
input  : head tag-set to be moved, destination plotter
output: overhead change if head moves to destination
foreach tag-set s ⊂ head do
    change owner to none;
end
overheadChange = 0;
foreach s ⊂ head do
    if it is not connected to destination plotter then
        overheadChange += counter of subset
    end
    if it is not connected to destination plotter then
        overheadChange -= counter of subset
    end
end
foreach s having common tags with head do
    if it belongs to source plotter then
        if it does not have any connection to destionation then
            overheadChange += counter of s;
        end
    end
    if it belongs to destination plotter then
        if it does not have any connection to source plotter then
            overheadChange -= counter of s;
        end
    end
end
foreach tag-set s ⊂ head do
    change owner back to source;
end
```

6.3.4 Calculating Jaccard Coefficient

Despite all these processes, the end goal remains the same. The application must emit the Jaccard Coefficients (JC) of all tag-sets that documents were annotated with. Therefore, once every a predefined period of time, plotters emit JC based on the information that they have amassed inside that interval of time. We call this time interval *JC window*. In this chapter we discuss the details of how and when JC are calculated.

The JC is calculated using pseudocode 3, which is a direct product of formula 3. Plotters calculate the JC only for tag-sets they are responsible for. Pseudocode 3 is a high level illustration of how plotters perform this process. In practice, supersets are found by following the supersets links in each vertex data-structure. Tag-sets with common tags are easier to find, since all that is needed is to iterate through the list each vertex maintains for this purpose. To find tag-sets quickly the tag-set to vertex index is used. Plotter maintains this index sound and complete throughout execution.

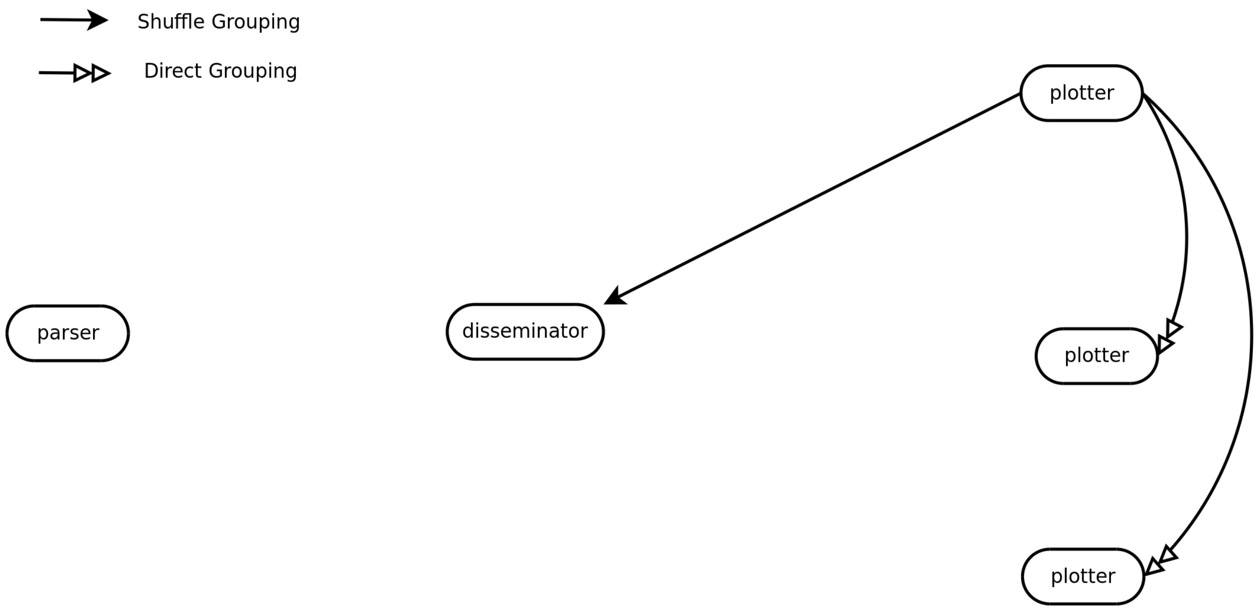


Figure 13: Tuples exchange on relocation

Extra care is required for JC and load check window not to end at the same time. Plotters use the counters stored in vertices in order to determine which vertices will be relocated and where. At the end of the *JC window*, counters are deleted and they start counting from the beginning again. Hence, the windows should not end at the same time, because plotters will not have enough information to determine which relocation is the best option.

Algorithm 3: Calculating Jaccard Coefficients

```
   output: Jaccard Coefficient of every tag-set node is responsible for
1  foreach tag-set s of plotter do
2      SupSum = 0;
3      ConSum = 0;
4      foreach tag-set $s' \supseteq s$ do
5          SupSum = SupSum + $s'$ counter;
6      end
7      foreach tag-set $s''$ having common tags with s do
8          ConSum = ConSum + $s''$ counter;
9      end
10     emit $SupSum/ConSum$;
11 end
```

In general, JC is only computed once for every tag-set. Nevertheless, in some cases head tag-sets that belong to different plotters have common subsets. Basically these subset tag-sets are owned by more than one plotter, therefore the JC for them is calculated more than once. However, there are not any discrepancies. Plotters emit the same JC for the same tag-set.

6.4 Testing

The proposed application has been implemented in Java 1.7 using the Apache Storm 0.9.3. Testing was performed in Local mode, hence instead of tasks running in a cluster of computers, each task was a separate thread in the same machine . The specifications of the machine are Intel(R) Core(TM)2 Quad CPU Q9300 @ 2.50GHz and 4GB RAM.

6.4.1 Data Source

Mining real data from twitter would be inefficient in this stage due to the volume and volatility of the data. Consequently, a new node has been created to produce controlled and supervised data. This node is called *Generator*.

Generator's one and only responsibility is to produce synthetic data and forward it to disseminator. The synthetic data was crafted in order to resemble as closely as possible to the data produced by twitter. Twitter documents are annotated with different tags. Most of the times, however, these tags are not unrelated and belong to a general category. For instance #enrollment, #thesis, #course, #college, #sophomore could belong to Academia category. That means that they appear together (in one tweet) with a greater probability. On the other hand, there are hash-tags that appear across many categories with the same chance. For example, #stress could belong to any category that contains stressful situations, such as Academia, Work, Financial Markets etc.

We model the above as follows: We have a predefined number of categories from which tags can be drawn from. There is also a pool of general tags that can appear across all categories. Tag-sets are of random size, up to a maximum. Moreover, each tag-set produced follows the following rules:

1. Every tag-set is assigned to a random category.
2. Every tag of the tag-set has a chance of being a general tag.
3. If the tag is not a general tag, then it will always be of the category that was chosen for the whole tag-set.

In the end, the synthetic data produced was suitable to test the application. The quality of the data relies heavily on how the different parameters are set.

6.4.2 Execution Example

In this section you can find a complete example of a typical run of the application. First we set the parameters, then we observe the messages different nodes exchange and at the end, we see a sample of the results produced.

Synthetic Data Parameters

To get good quality data, as discussed previously, we set generator's parameters to the following values:

1. The number of different categories is set equal to thirty.
2. We set the number of different generic tags to fifty.
3. We limit the maximum number of tags in a tag-set to five.
4. The probability for a tag to be a generic tag is set to five percent.
5. The number of different tags per category is set to one hundred.

Application Parameters

These parameters change the way the topology runs.

1. The number of plotters is set to five.
2. The *overload factor* (paragraph 5.3.2) is set to twenty percent.
3. JC window is set to thirty seconds
4. Load Check window is set to thirty seconds as well, although with a fifteen seconds head start, so it will not end at the same time with JC window.

Tuple Exchange

The box below displays the inner workings of the application. These logs demonstrate the exchange of tuples between topology's components. Whenever a component emits or receives a tuple, it prints what was emitted or received accordingly. Since many threads run at the same time, actions do not come in a linear, sequential manner. However, all the kinds of messages, together with their tuple structure, are present. If you find it difficult to comprehend these lines, take a closer look at tables 1, 2, which explain the tuples that components exchange.

```
...........
8221 [Thread-10-generator] INFO  backtype.storm.daemon.task - Emitting: generator tagset [[#2_48
    ]]
8224 [Thread-10-generator] INFO  backtype.storm.daemon.task - Emitting: generator tagset [[#28_80
    , #28_6, #28_81, #28_28]]
8224 [Thread-8-disseminator] INFO  backtype.storm.daemon.executor - Processing received message
    source: generator:3, stream: tagset, id: {}, [[#19_18]]
8224 [Thread-8-disseminator] INFO  backtype.storm.daemon.task - Emitting direct: 8; disseminator
    disseminator_new_tagset [[#19_18], 8]
...........
8301 [Thread-8-disseminator] INFO  backtype.storm.daemon.executor - Processing received message
    source: generator:3, stream: tagset, id: {}, [[#17_90, #17_18]]
8302 [Thread-8-disseminator] INFO  backtype.storm.daemon.task - Emitting direct: 6; disseminator
    disseminator_new_tagset [[#17_90, #17_18], 6]
...........
51156 [Thread-18-plotter] INFO  backtype.storm.daemon.executor - Processing received message
    source: disseminator:2, stream: disseminator_new_tagset, id: {}, [[#27_70, #27_45], 4]
51156 [Thread-18-plotter] INFO  backtype.storm.daemon.task - Emitting direct: 4; plotter
    plotter_new_tagset [[[#27_70, #27_38, #27_42], [#27_70, #27_47, #27_4, #27_31]], {[#27_70,
    #27_38]=0, ... [#27_70, #27_31]=0}]
51156 [Thread-18-plotter] INFO  backtype.storm.daemon.task - Emitting: plotter index_update [
    remove, #27_45]
...........
20841 [Thread-12-plotter] INFO  backtype.storm.daemon.task - Emitting direct: 6; plotter
    change_owner [[#7_58, #7_57, #7_79], 7]
```

```
20842 [Thread-12-plotter] INFO  backtype.storm.daemon.task - Emitting direct: 7; plotter
    relocation [[#7_58, #7_57, #7_79], {[#7_31, #7_90, #7_2, #7_79]=6, ... [#7_33, #7_92, #7_78,
    #7_79]=4}, {[#7_57, #7_79]=0, .... [#7_31, #7_90, #7_79]=0, [#7_79]=0}]
20842 [Thread-12-plotter] INFO  backtype.storm.daemon.task - Emitting: plotter index_update [
    remove, #7_58]
20842 [Thread-12-plotter] INFO  backtype.storm.daemon.task - Emitting: plotter index_update [
    remove, #7_57]
...........
51639 [Thread-18-plotter] INFO  backtype.storm.daemon.executor - Processing received message
    source: plotter:4, stream: relocation, id: {}, [[#7_58, #7_57, #7_79], {[#7_31, #7_90, #7_2,
    #7_79]=6, ... [#7_33, #7_92, #7_78, #7_79]=4}, {[#7_57, #7_79]=0, ... [#7_79]=0}]
51639 [Thread-8-disseminator] INFO  backtype.storm.daemon.task - Emitting direct: 8; disseminator
     disseminator_new_tagset [[#2_90, #2_30, #2_50, #2_97], 4]
51639 [Thread-18-plotter] INFO  backtype.storm.daemon.task - Emitting: plotter index_update [add,
     #7_58]
...........
```

Results

After the end of JC window, the Jaccards Coefficients are emitted by the plotters. A sample of the results is displayed in the box below.

```
...........
JC for [#0_52, #0_86] is 0.1
JC for [#18_64] is 1.0
JC for [#8_92, #8_13, #8_0, #8_60] is 0.05555555555555555
JC for [#9_25, #9_10, #9_77, #9_78] is 0.1
JC for [#23_51, #23_24, #23_55] is 0.25
JC for [#14_38, #14_9] is 0.14285714285714285
JC for [#16_43, #16_87] is 0.25
JC for [#6_50, #6_32] is 0.2
JC for [#6_41, #6_45] is 0.25
JC for [#6_10, #6_18, #6_33, #6_30] is 0.0625
JC for [#7_75, #16] is 0.0625
JC for [#0_90, #0_49] is 0.125
JC for [#14_65, #14_40, #14_97] is 0.0625
JC for [#1, #16_29, #16_51, #16_56] is 0.05263157894736842
JC for [#16_10, #16_22, #16_56, #16_84] is 0.05263157894736842
JC for [#13_19, #13_58, #8, #13_7] is 0.041666666666666664
JC for [#22_3, #22_56, #22_13] is 0.1
...........
```

7. CONCLUSION

To summarize, what we did was to design a flexible system of communicating nodes, that can calculate correlations without relying on a ’know it all’ node. Despite the challenges faced, a viable solution was found. A graph was devised with all the theoretical aspects deemed necessary to adequately tackle the issue. After that, the Jaccard formula was transformed in a form well fitting to the features of the graph. At the next step of the designing process an efficient way to parallelize the work-flow was found. Finally, all this was successfully implemented in a prototype built using the Storm Framework.

In conclusion, what this work has shown is that it is possible to build a flexible topology that can calculate the correlations between tags. Plotter nodes had no problem communicating and cooperating with each other. The model has resulted in quite a modular build. Modular, generally, means more extensible and more scalable. We hope that this would be the beginning of an exciting path towards achieving something even greater.

8. FUTURE WORK

From here and onwards, there are a number of things that should be further investigated. First and foremost the performance of the application should be tested. Secondly, innovative ideas that could be imagined, would make organic changes in the structural components of the application and decisively change it to the better.

In local mode, is not easy to test the application's efficiency. The more plotters run simultaneously, the more threads are executed. The more threads Storm has to maintain the more burdensome it was for the application. Therefore, in contrary to a real situation, more plotters resulted in a slower performance.

If the application is to be used in real life situations, extensive testing should take place. These tests should take place using a cluster of computers while harnessing real data from a social media site. This will not only keep the incoming rate high, but also take data from a real source. While synthetic data can be sufficient for the first tests, good quality of synthetic data is difficult to produce. In the light of these tests, possible structural improvements may be possible.

A possible structural improvement could be in the form of designing a better algorithm for a process. For instance, the algorithm used by the plotter to decide which vertices to move to another plotter is just a heuristic. As such, it can be replaced easily with another one that will take into account different parameters.

Likewise, the storm topology and its components could be altered to increase performance. Storm *bolts* should be designed to do as few processing as possible before forwarding the results to the next bolt in line. Hence an alternative topology might exist, with more components, that can split the work-flow even more. However, I have to warn you that this could be a dangerous path, if you are not careful. Due to the nature of the problem, extensive communication between nodes is inevitable, therefore having an abundance of nodes could result in slow performance. In the end, the possibilities are unlimited. It remains to the reader's imagination as to what could come next.

ABBREVIATIONS, INITIALS AND ACRONYMS

JC	Jaccard Coefficient

BIBLIOGRAPHY

[1] Foteini Alvanaki and Sebastian Michel. Tracking set correlations at large scale. In Proceedings of the 40th ACM International Conference on Management of Data, SIGMOD ’14, pages 1507–1518, Snowbird, UT, USA, 2014. ACM.

[2] Qiankun Zhao, Prasenjit Mitra, and Bi Chen. Temporal and information flow based event detection from social text streams. In Proceedings of the 22Nd National Conference on Artificial Intelligence - Volume 2, AAAI ’07, pages 1501–1506, Vancouver, British Columbia, Canada, 2007. AAAI Press.

[3] Hassan Sayyadi, Matthew Hurst, and Alexey Maykov. Event detection and tracking in social streams. In Proceedings of the International Conference on Weblogs and Social Media, ICWSM ’09, 2009.

[4] Foteini Alvanaki, Sebastian Michel, Krithi Ramamritham, and Gerhard Weikum. See what’s en-blogue: Real-time emergent topic identification in social media. International Conference on Extending Database Technology (EDBT), 2012.

[5] Storm: Distributed and fault-tolerant realtime computation. http://storm-project.net/.

[6] Marie-Jeanne Lesot, Maria Rifqi, and H. Benhadda. Similarity measures for binary and numerical data: a survey. International Journal of Knowledge Engineering and Soft Data Paradigms, 1(1):63–84, December 2009.

[7] Paul Jaccard. The distribution of the flora in the alpine zone. New Phytologist, 11(2):37–50, February 1912.

[8] Eugene Agichtein, Carlos Castillo, Debora Donato, Aristides Gionis, and Gilad Mishne. Finding high-quality content in social media. In Proceedings of the 2008 International Conference on Web Search and Data Mining, WSDM ’08, pages 183–194, Palo Alto, California, USA, 2008. ACM.

[9] Ana-Maria Popescu and Marco Pennacchiotti. Detecting controversial events from twitter. In Proceedings of the 19th ACM International Conference on Information and Knowledge Management, CIKM ’10, pages 1873–1876, Toronto, ON, Canada, 2010. ACM.

[10] Vicenc Gomez, Andreas Kaltenbrunner, and Vicente Lopez. Statistical analysis of the social network and discussion threads in slashdot. In Proceedings of the 17th International Conference on World Wide Web, WWW ’08, pages 645–654, Beijing, China, 2008. ACM

[11] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: Membership, growth, and evolution. In Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD ’06, pages 44–54, Philadelphia, PA, USA, 2006. ACM.

[12] Ankit Toshniwal , Siddarth Taneja , Amit Shukla , Karthik Ramasamy , Jignesh M. Patel , Sanjeev Kulkarni , Jason Jackson , Krishna Gade , Maosong Fu , Jake Donham , Nikunj Bhagat , Sailesh Mittal , Dmitriy Ryaboy, Storm@twitter, Proceedings of the 2014 ACM SIGMOD international conference on Management of data, June 22-27, 2014, Snowbird, Utah, USA

[13] Foteini Alvanaki and Sebastian Michel. Scalable, continuous tracking of tag co-occurrences between short sets using (almost) disjoint tag partitions. In Proceedings of the ACM SIGMOD Workshop on Databases and Social Networks, DBSocial ’13, pages 49–54, New York, New York, USA, 2013. ACM.

[14] S4: Distributed stream computing platform. http://incubator.apache.org/s4/.

[15] Michihiro Kobayakawa, Shigenao Kinjo, Mamoru Hoshi, Tadashi Ohmori, Atsushi Yamamoto: Fast Computation of Similarity Based on Jaccard Coefficient for Composition-Based Image Retrieval. PCM 2009: 949-955

[16] Mustafizur Rahman, Md. Rafiul Hassan, Rajkumar Buyya: Jaccard Index based availability prediction in enterprise grids. ICCS 2010: 2707-2716

[17] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, SIGMOD ’03, pages 647–651, San Diego, California, 2003. ACM.

[18] James Allan, Ron Papka, and Victor Lavrenko. On-line new event detection and tracking. In Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR ’98, pages 37–45, Melbourne, Australia, 1998. ACM.

[19] Evgeniy Gabrilovich, Susan Dumais, and Eric Horvitz. Newsjunkie: Providing personalized newsfeeds via analysis of information novelty. In Proceedings of the 13th International Conference on World Wide Web, WWW ’04, pages 482–490, New York, NY, USA, 2004. ACM.

[20] Themistoklis Palpanas, Dimitris Papadopoulos, Vana Kalogeraki, and Dimitrios Gunopulos. Distributed deviation detection in sensor networks. SIGMOD Record, 32(4):77–82, December 2003.