View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CWI's Institutional Repository

Tracking Set Correlations at Large Scale

Foteini Alvanaki
Saarland University
Saarbrücken, Germany
alvanaki@mmci.uni-saarland.de

Sebastian Michel
Saarland University
Saarbrücken, Germany
smichel@mmci.uni-saarland.de

ABSTRACT

In this work, we consider the continuous computation of correlations between co-occurring tags that appear in messages published in social media streams. The vast amount and pace these messages are created makes it necessary to parallelise the computation of correlations to various nodes in a computing cluster. The main challenge in this is to ensure that each node will compute a subset of the coefficients and every coefficient will be computed by some node. The core task is to continuously create and maintain partitions of the tags and forward the incoming messages based on them. Our approach proposes and evaluates several algorithms that partition the tags to the nodes while at the same time they minimise the replication of tags to the nodes and balance the load on them. The proposed framework is implemented in Java within the Storm stream processing platform. We evaluate the partitioning algorithms and validate the feasibility of our approach through a thorough experimental study performed using real data.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; H.4 [**Information Systems Applications**]: Miscellaneous

Keywords

Distributed Stream Processing; Partitioning; Set Correlations

1. INTRODUCTION

In 2012, Twitter alone contributed over 200 million tweets a day and around 2.3 billion posts were shared on Facebook[1]. Such user generated content is usually carrying short textual annotations – so called tags – that specify the topic of a tweet, content of videos or images, locations and persons, or events. Analysing these massive and topically diverse streams of user generated content in real time is one of the dimensions of the big data challenge, with the ultimate goal to extract valuable information for business settings, political campaigns, disease outbreak warnings etc. One of the applications most prominently visible in recent research aims specifically at extracting trends out of Twitter tweets. Here, the choice of terms used as hashtags is not imposed by Twitter, which results in a wild use of such tags – person names, brands, locations, events, etc. Computing correlation measures between co-occurring tags proves to be challenging, due to high rate of arriving data together with the massive amounts of contained tags. In order to meet the needs of real-time applications, it is necessary to distribute the computations to multiple nodes (machines), where each node will be responsible to compute the measures of interest for a subset of all co-occurring tags. In this work, we specifically consider the computation of the Jaccard coefficient; for a set of tags it is defined as the ratio of the number of documents annotated with all of the set's tags to the number of documents annotated with any of the tags. This paper devises and evaluates algorithms and the necessary information processing topology within the Storm stream processing platform [19] to allow computing Jaccard coefficients for all co-occurring tags of a Twitter-scale data stream.

[1]http://www.statisticbrain.com/social-networking-statistics

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SIGMOD/PODS'14, June 22–27 2014, Snowbird, UT, USA.
Copyright 2014 ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2610510.

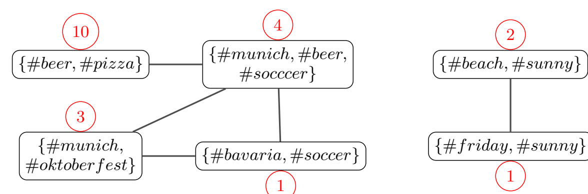


Figure 1: Example of a tagset graph with relations

Consider for instance the graph in Figure 1. Each vertex represents a set of co-occurring tags with a weight equal to the number of received documents that were annotated with the specific tagset. There is an edge between two vertices if the corresponding tagsets have common tags. A partition that assigns connected components of vastly different size to different nodes creates large load imbalances. Assuming that the ratio of incoming documents does not change over time, the node assigned the partition with tags {**#munich**, **#beer**, **#pizza**, **#socccer**, **#oktoberfest**, **#bavaria**} will receive and process 86% of the incoming documents, while the node assigned the partition with tags {**#beach**, **#sunny**, **#friday**} only the remaining 14%. On the other hand, a partition that cuts the large connected component creates replicated tags, which increases/causes the communication overhead. Assuming that the partitions are {**#munich**, **#beer**, **#socccer**, **#oktoberfest**, **#beach**, **#sunny**, **#friday**} and

{#beer, #pizza, #socccer, #bavaria} all documents containing either #beer or #socccer will have to be forwarded to both nodes.

1.1 Problem Statement

We consider a stream of documents $\mathcal{D}$ obtained through Twitter or other social media. Each document d_i in this stream is annotated with a set of tags (or hashtags) $s_i = \{t_1, t_2, \ldots, t_k\}$.

We use a time-based sliding window over the documents to limit the focus on the most recent of them. Before moving the window, we estimate the similarity of the tags in any subset of co-occurring tags currently in the window computing its Jaccard coefficient. Assuming a set $TG = \{t_1, t_2, \ldots\}$ of all tags in the window, we compute the Jaccard coefficient for all sets $\{t_i, t_j, \ldots\} \in 2^{TG}$ that *co-occur in at least one document*. While the consideration of all subsets in TG is computationally intractable in theory, in practice, the natural restriction to tags that actually co-occur in the observed data renders the computation feasible. This is due to the relatively low number of tags per tweet and the, in general, quite disperse use of hashtags in thematically unrelated communities.

We want to distribute the computation of the Jaccard coefficients to multiple nodes. For that, we want to assign a set of tags pr_i out of TG to each node n_i. Each node n_i will then be responsible to compute the Jaccard coefficient for all sets of co-occurring tags in 2^{pr_i}. The distribution of the tags to the nodes should ensure that

1. All co-occurring tags are assigned to some node, such that all Jaccard coefficients can be computed.
2. Assignment of the same tag to multiple nodes is avoided, so that the network traffic is kept low.
3. Assignment of popular tags is spread, such that the load in the nodes is balanced.

Formally: *For a set $\mathcal{S}$ of tagsets $s_i = \{t_1, t_2, \ldots\}$ annotating a set of documents $D = \{d_1, d_2, \ldots\}$ create k partitions $pr_1, \ldots, pr_k$ such that*

1. $\forall s_i \in \mathcal{S} \ \exists pr_j : s_i \subseteq pr_j$ *(i.e. coverage)*
2. $\sum_{t_i \in \{\bigcup_j s_j\}} (|\{pr_m \mid t_i \in pr_m\}|)$ *is minimised (i.e. little/no replication)*
3. *given* $l_i = |\{d_m \mid s_j \subseteq pr_i \wedge s_j \text{ annotates } d_m\}|)$ *to be the load of a node i; minimise the Gini coefficient, or in general the load variance, among the nodes (i.e., balanced load)*

1.2 Contributions and Outline

In this paper, we make the following contributions.

- We derive a data dissemination scheme that assigns tags to nodes considering their co-occurrences and using communication and computational load as optimisation criteria.
- We discuss a theoretic view on the performance of the presented partitioning algorithms.
- We describe a detailed operator topology that we implemented within the Storm distributed stream processing framework.
- We present the results of a carefully conducted experimental evaluation in a Storm cluster, using real Twitter data.

The remainder of this paper is organised as follows. Section 2 discusses the related work. Section 3 describes the key idea and framework. Section 4 presents algorithms to distribute tags to nodes. Section 5 estimates theoretically the expected performance of the presented partitioning algorithms. Section 6 describes in details the implementation of our system using the distributed data stream processing platform Storm. Section 7 addresses the problem of dynamically adapting partitions to evolving tag usage and Section 8 presents the results of our experimental evaluation and summarizes the lessons learned. Section 9 concludes the paper.

2. RELATED WORK

Analysing massive streams of user generated content in real time is one of the dimensions of the big data challenge, and most prominently visible in works specifically aiming at extracting trends out of Twitter tweets. Among the early works along these lines is the Twitter Monitor approach, by Mathioudakis and Koudas [15], which extracts bursty tags and then groups them to form meaningful descriptions of individual trends. Our own approach, enBlogue [2], computes the magnitude of a trend as the prediction error of the tagset correlation, where the Jaccard coefficient is computed for pairs of observed tags – or a subset of them.

There are many works on identifying topics on Twitter. For example Sriram et al. in [18] try to assign tweets in broad categories like News, Opinions etc. In [16] the authors limit the topics they identify to those containing known entities and in [20] to those happening at specific places. Weiler et al. in [21] identify topics using the inverse document frequency and Ritter et al. in [17] use more sophisticated NLP techniques and unsupervised clustering. All these works and many more focus on identifying qualitative topics over Twitter and overlook any performance issues. Similarly, in the work of Eftekhar et al. [7] the main focus is to identify high quality topics. The procedure is preformed offline over data gathered during a whole day. In this setting they also try to develop algorithms that can identify topics efficiently. In this work, we focus on continuously computing tag correlation statistics over multiple servers in a computing cluster. The computed statistics can serve as input to various works above. The task to compute tag correlations in a distributed fashion requires partitioning the incoming data stream. As we deal with set-valued data (i.e. Tweets and their hashtags), the problem is related to graph partitioning. The graph in our case has tagsets as nodes and edges between tagsets that share tags; we will review this in more detail in Section 4. Graph partitioning is a well studied problem and many algorithms on solving this problem exist. Among the best known is the Kernighan-Lin algorithm [12] and the algorithms based on spectral analysis [6]. The algorithms based on spectral analysis use the eigenvectors of the connection matrices to estimate the areas in the graph where the partitions should be performed. Given a set of partitions, the Kernighan-Lin algorithm transfers vertices from one partition to the other trying to minimise the edge-cut. The combination of these algorithms [11] has been shown to produce better partitions. The above and any other partitioning algorithm could be used in our setting to create the partitions of tag-sets. However, in a dynamic environment like ours all these techniques are deemed computationally expensive considering the observation that any partitioning computed will be valid/appropriate only for a short period. Additionally, there are algorithms that maintain graphs dynamically (e.g. [8]). Dynamic techniques are more appropriate for our setting. However, methods for dynamic graph maintenance

$\mathcal{TG}$, t_i	Global set of tags, A single tag
$\mathcal{D}, d_i$	A set of documents, A single document
T_i	The set of documents annotated with tag t_i
$\mathcal{T}$	A set of sets T_i
$\mathcal{S}, s_i$	A set of tagsets, A single tagset
$\mathcal{PR}, pr_i$	A set of tag partitions, A single tag partition
k	Number of partitions
$\mathcal{DS}, ds_i$	A set of disjoint sets, A disjoint set
l_i, c_i	Load of tagset s_i, Cost of tagset s_i
P	Number of Partitioners
thr	The threshold allowed before repartitions are requested
tps	Incoming tweets per second

Table 1: Notations used in the paper

are restricted to changes in the edges and cannot handle additions or deletions of vertices. Changes in vertices (e.g. new tagsets) are very common in our approach.

Using probabilistic sketching data structures like Bloom Filters [3] or Count-Min sketches [5] has been proposed to accelerate set operations like intersection of union. If we use them to represent the sets of documents annotated with each tag we will falsely consider tags to be co-occurring, due to "false positives" imposed by the sketches. In a setting as ours were most of the tags do in fact not co-occur, i.e. their sets of documents have an empty intersection, using sketches will pose a significant overhead forcing us to consider many non co-occurring tags.

An early version of this paper has been published in [1]; we refer to it for its modeling of the expected communication overhead of set-cover–based partitioning methods that we briefly pick up in this work.

3. IDEA AND FRAMEWORK

We consider a stream $\mathcal{D}$ of incoming documents d_i each one annotated with a set of tags $s_i = \{t_1, t_2, \ldots\}$ from a global set of tags TG. We have at our disposal k machines that are independent from each other, i.e. they cannot communicate. We want to *assign* the tags in TG to the k machines (or nodes) in such a way that each machine can compute the Jaccard coefficient for a subset of the sets of co-occurring tags seen in the input. The tagsets for which a Jaccard coefficient is computed should equal the co-occurring tags found in the incoming stream of documents. After the assignment, each machine receives all documents annotated with tags it has been assigned.

Assume there are 2 machines and the received documents along with their tagsets are summarised by the graph in Figure 1. One of multiple possible partitions is the following

- pr_1={`#munich`, `#beer`, `#soccer`, `#oktoberfest`, `#beach`, `#sunny`, `#friday`}
- pr_2={`#beer`, `#pizza`, `#bavaria`, `#soccer`}

The node assigned partition pr_1 will compute the Jaccard coefficient for the sets of co-occurring tags in the following list

- {`#munich`, `#beer`}
- {`#munich`, `#soccer`}
- {`#beer`, `#soccer`}
- {`#munich`,`#beer`, `#soccer`}
- {`#munich`, `#oktoberfest`}
- {`#beach`, `#sunny`}
- {`#friday`, `#sunny`}

The node will not compute a Jaccard coefficient for {`#oktoberfest`,`#friday`}, for instance, since the two tags do not co-occur in any of the received documents. Similarly, the node assigned partition pr_2 will compute a Jaccard coefficient for the tagsets

- {`#bavaria`, `#soccer`}
- {`#beer`, `#soccer`}
- {`#beer`, `#pizza`}

In the above partitioning, the tags `#beer` and `#soccer` are assigned to both nodes, introducing *communication overhead* as any document containing either of these tags would be forwarded to both nodes. However, this cannot be avoided. Removing for example `#beer` from pr_2 will result in loosing the coefficient for the tagset {`#beer`, `#pizza`}, since there will be no node assigned both the tags.

Assuming that the relative number of documents annotated with each tagset remains the same through time, the expected load in each node is computed as the number of documents annotated with any of the tags the node has been assigned. In this example, the tags in pr_1 are found in 21 documents (the documents annotate with {`#beer`, `#pizza`}, {`#bavaria`, `#soccer`} or {`#munich`, `#beer`, `#soccer`}) and the tags in pr_2 in 15 (the documents annotated with {`#munich`, `#beer`, `#soccer`}, {`#munich`, `#oktoberfest`} or {`#beer`, `#soccer`}). Thus, the node assigned pr_1 will have a load of 58% and the node assigned pr_2 the remaining 42%.

Ideally, partitions are mutually disjoint and cause equal load to the nodes that are responsible for handling them. In practise, such an ideal partitioning does not necessarily exist due to the characteristics of the data. Algorithms aim at low mutual overlap for low communication overhead and, as much as possible, equally loaded nodes.

With evolving time, new tags and unseen tag combinations are introduced by users, and the relative popularity of the assigned tagsets changes deteriorating the quality of the partitions. We propose a practical online solution that

(i) computes partitions based on the recently observed tags and their co-occurrences,
(ii) introduces updates/additions to computed partitions to account for new tags and new tag co-occurrences
(iii) monitors the quality of the partitions and triggers their recomputation

The framework we propose consists of three main operators

Calculator: it counts occurrences of tagsets and computes the Jaccard coefficients for the co-occurring tags

Partitioner: it computes tag partitions that indicate which Calculator receives which documents

Disseminator: it forwards the documents to the Calculators according to the defined partitions and monitors the quality of the partitions

The implemented topology within the Storm framework is presented in Section 6. The next paragraphs give an introduction to the high-level task each individual operator is continuously performing.

3.1 Calculator Operator

Each Calculator is assigned a set of tags and is responsible to compute the Jaccard coefficients for all sets of co-occurring tags in this set. The Jaccard coefficient for a set of tags $s_i = \{t_1, t_2, \ldots, t_n\}$ is computed as the ratio of the number of documents annotated with all tags in s_i to the

number of documents annotated with any of the tags in s_i:

$$J(t_i|_{i=1}^n) = \frac{|\bigcap_j \{d_j\}|}{|\bigcup_j \{d_j\}|} \quad (1)$$

where d_j is a document annotated with any of the tags in s_i In order to compute this measure, the Calculator operator needs to be provided with all documents that contain any of the Calculator's partition of tags. For each set of co-occurring tags existing in the data, the Calculator maintains a simple counter for its occurrences, that is the count of documents that contain *all* of the tagset's tags. The count of documents that are annotated with *any* of the tagset's tags is given by the inclusion-exclusion principle:

$$\left|\bigcup_{i=1}^{n} a_i\right| = \sum_{k=1}^{n} (-1)^{k+1} \left(\sum_{1 \le i_1 < \cdots < i_k \le n} |a_{i_1} \cap \cdots \cap a_{i_k}| \right) \quad (2)$$

Still, Calculators have to receive all documents annotated with *any* of the tags they have been assigned. At first glance, this exact counting of occurrences of all subsets sounds prohibitively expensive due to the combinatorial explosion for large number of tags. However, since tags are used to indicate the topic of a document and the documents we consider in this work are posts in the Blogosphere, e.g. tweets (i) less than 10 tags are used per document (cf. e.g. [7]) and (ii) not all possible tag combinations are used.

3.2 Partitioner Operator

The Partitioner operator uses a set of recent documents and decides how the tags found in these should be partitioned to the Calculators. The partitioning algorithm, used by the Partitioner to split the tags to the Calculators, should take care to split them such that for any tagset seen in the input there is one Calculator assigned all tags comprising it. For any tagset which is not completely assigned to a Calculator the Jaccard coefficient cannot be computed. Section 4 discusses algorithms used for the partitioning.

3.3 Disseminator Operator

The Disseminator operator has a global view of the tags assigned to each Calculator. It receives the incoming documents and is responsible to forward them to the appropriate Calculators, i.e. those that have been assigned tags used for the annotation of the document. Finding efficiently the Calculators that should be informed about each document is important. The work [10] on indexing set-valued attributes suggests that using an inverted index is more efficient compared to other techniques. For each tag t_i a set $\{C | C \text{ is counting tag } t_i\}$ of Calculators is kept and indexed by key t_i, usually in a simple hash-based index. For a received document, an index lookup for each tag in the document is performed to obtain the Calculators that should receive this document.

4. PARTITIONING ALGORITHMS

Any partitioning algorithm used to split the tags to the partitions must ensure that for every set of co-occurring tags there is one partition containing *all* its tags – the Jaccard coefficient of a not completely captured tagset cannot be computed. Additionally, it should create partitions of equal load and minimise the communication overhead.

This problem can be modeled as a *graph partitioning problem*. Each vertex v_i in the graph corresponds to a tagset s_i and there is an edge $e_{(v_i,v_j)}$ between two vertices v_i, v_j if the corresponding tagsets s_i, s_j have common tags. The weight of a vertex v_i represents the number of documents that will be forwarded to the Calculator assigned the tagset s_i. The weight of an edge $e_{(v_i,v_j)}$ represents the reduction in the number of documents that will be forwarded to the Calculator assigned both tagsets s_i, s_j.

For the kind of data we consider in this work – tagsets of low cardinality occurring in social media messages like tweets – a graph constructed like this contains a large amount of small connected components. In the following, we present a partitioning algorithm that uses these connected components to create disjoint sets of tags, grouped afterwards into k partitions (Subsection 4.1). Additionally, we present algorithms based on the Set Cover [4] problem (Subsection 4.2).

4.1 Disjoint Sets Algorithm (DS)

In social media like Twitter users annotate their messages with tags that reflect the topics discussed on them. Tags describing the same topic are thus strongly connected, i.e. found in the same messages, while being completely disconnected from the other tags. Algorithm 1 is based on this observation.

Algorithm 1: Disjoint Sets Algorithm **(DS)**

Input: Integer k, Set $\mathcal{S}$ of tagsets s_i,
Set of tags $\mathcal{TG}$, Set $\mathcal{T}$ of sets of documents T_i
Output: Set $\mathcal{PR}$ of k tag partitions pr_i

```
1  DS = {}, PR = {}
2  while TG ≠ ∅ do
3  |  find ds_j = ⋃_i s_i disjoint set of tags
4  |  l_j = |⋃_i T_i|, t_i ∈ ds_j
5  |  DS = DS ∪ {ds_j}
6  |  TG = TG \ ds_j
7  end while
8  while DS ≠ ∅ do
9  |  ds_i = argmax_{ds_j} l_j
10 |  DS = DS \ {ds_i}
11 |  if k > 0 then
12 |  |  pr_k = ds_i
13 |  |  PR = PR ∪ {pr_k}
14 |  |  k = k − 1
15 |  else
16 |  |  pr_i = argmin_{pr_j} Σ_{ds_k ∈ pr_j} l_k
17 |  |  pr_i = pr_i ∪ ds_i
18 |  end if
19 end while
20 return PR
```

Initially the sets of tags that form connected components, *disjoint sets*, are identified (Algorithm 1, Lines 2-7). The identified disjoint sets are merged into k sets/partitions, where k is the number of available machines (Algorithm 1, Lines 8-19). Each disjoint set ds_j carries a load l_j equal to the number of documents annotated with any of the tags $t_i \in ds_j$ (Algorithm 1, Line 4). As long as there are more disjoint sets to be assigned the one with the biggest load is selected (Algorithm 1, Line 9) and assigned to the partition with the lowest current load (Algorithm 1, Line 16).

At the end of the algorithm there are k tag partitions with approximately the same load. Because of the initial phase where disjoint sets are identified and never split after that, the algorithm guarantees that for any set of co-occurring tags there is a single node that has been assigned all its

tags. So, the Jaccard coefficient for any set of co-occurring tags can be computed.

In case there are not enough disjoint sets to create k partitions or there is a disjoint set that is very big, set-cover–based algorithms, presented below, can be used in combination with the disjoint sets algorithm to split this set (or these sets) to smaller ones.

4.2 Set-Cover–Based Algorithms

The following algorithms treat the creation of k partitions as a *Set cover problem* (SCP) over the input tagsets s_i. The general SCP assumes that all elements of the given sets are independent to each other. Therefore, a collection of sets $S = \{\{1,2,3\},\{2,4\},\{3,4\},\{4,5\}\}$ can be represented with less sets like $\{\{1,2,3\},\{4,5\}\}$ without losing information. Considering $\{1,2,3\}$ and $\{4,5\}$ to create two partitions there is one partition containing every single element in S.

In our setting, assigning each single element in a partition is not enough. Instead, every occurring tagset should be assigned as a whole to some partition as the Jaccard coefficient for non-assigned tagsets cannot be computed locally at the Calculator nodes. We use a greedy approach of the *Budgeted Maximum Coverage Problem* [13] to create k initial partitions and then assign to them all non-assigned tagsets.

In the *Budgeted Maximum Coverage Problem* there is a collection of sets $S = \{s_1, s_2, \ldots, s_n\}$ defined over a collection of elements $TG = \{t_1, t_2, \ldots, t_n\}$. Each set $s_i \in S$ has a cost c_i, each element $t_i \in TG$ has a weight w_i and there is a budget B. The goal is to find a collection of sets S' with total cost that does not exceed the budget B with maximised total weight of covered elements. In our setting, the sets s_i are the sets of co-occurring tags and the elements t_i are the tags. The weight of each single tag is equal to the unit, i.e. there are no tags more important than others.

In each iteration of the *Budgeted Maximum Coverage Problem* there is a subset $C \subset S$ of n sets s_i that have been selected to be part of the final set cover. The best set to be added in C is the one that covers the most elements not already covered by the sets in C, while at the same time the total cost does not exceed the budget B. We do not have a hard limit on the budget, but we want to minimise the final cost of the cover.

Algorithm 2 outlines the procedure followed for the selection of the k initial sets that will be later used as the basis for the k partitions. At each iteration the tagset s_i with the minimum cost that covers the most uncovered tags (Algorithm 2, Line 3) is added to the set of selected tagsets.

Algorithm 2: Set Cover Based Algorithms – Phase 1: Creation of initial partitions

Input: Integer k, Set $\mathcal{S}$ of tagsets s_i,
Set $\mathcal{T}$ of sets of documents T_i
Output: Set $\mathcal{PR}$ of k tag partitions pr_i

```
1  CV = {} // Set of already covered tags
2  while k > 0 and S ≠ ∅ do
3      s_i = argmin_{s_j} c_j and argmax_{s_j} |s_j \ CV|
4      pr_k = s_i
5      PR = PR ∪ {pr_k}
6      S = S \ {s_i}
7      CV = CV ∪ s_i
8      k = k − 1
9  end while
10 return P
```

The cost of each set s_i differs depending on whether the algorithm optimises for the communication overhead or for the processing load. In case the measure of interest is the **communication overhead**, the cost c_i of each set s_i in each iteration is equal to the number of tags $t_j \in s_i$ that are already covered by the sets in C. When the measure of interest is the **processing load**, the cost c_i of each set s_i in each iteration is equal to the difference of the share this set has in the load to the optimal share. Each tag t_i has been found in a set of documents T_i. The cardinality of the union of these documents for all tags $t_i \in s_j$ is considered to be the load l_j of the tagset s_j. Assuming we are in the mth iteration of the algorithm, C contains $m-1$ sets and we will select the mth set. The optimal share of load in this iteration is $pl_{op} = \frac{1}{m}$, i.e. the load is equally distributed to all nodes. The real share of load of a candidate set s_n is $pl_n = \frac{l_n}{\sum_i^{m-1} l_i + l_n}$ and the cost of s_n is define as $|pl_{op} - pl_n|$.

To the initial k partitions created using Algorithm 2 are added the remaining tagsets until there is no unassigned tagset. The best partition to assign a tagset depends again on the measure of interest. When optimising for the communication overhead (**SCC** algorithm shown in Algorithm 3), in each iteration the set with the most not covered tags having the least total tags is selected (Algorithm 3, Line 3). The selected tagset is added to the partition sharing with it the most tags having the least load (Algorithm 3, Line 4). When optimising for the load distribution (**SCL** algorithm shown in Algorithm 4) in each iteration the set with the most load having the least already covered tags is selected (Algorithm 4, Line 3). The selected tagset is added to the partition having the least load sharing the most tags with the selected tagset (Algorithm 4, Line 4).

Algorithm 3: Set Cover Based Algorithms – Phase 2: Assigning the remaining tagsets minimising the communication overhead **(SCC)**

Input: Set $\mathcal{PR}$ of k initial tag partitions pr_i,
Set $\mathcal{S}$ of tagsets s_i,
Set $\mathcal{T}$ of sets of documents T_i
Output: Set $\mathcal{PR}$ of k final tag partitions pr_i

```
1  CV = ⋃_i pr_i // Set of already covered tags
2  while S ≠ ∅ do
3      s_i = argmax_{s_j} |s_j \ CV| and argmin_{s_j} |s_j|
4      pr_i = argmax_{pr_j} |s_i ∩ pr_j| and
       argmin_{pr_j} Σ_{s_k ∈ pr_j} l_k
5      pr_i = pr_i ∪ s_i
6      S = S \ {s_i}
7      CV = CV ∪ s_i
8  end while
9  return PR
```

In prior work [1], the partitioning problem is treated as a *Maximum Coverage Problem without budget* to select the k first tagsets. The assignment of the remaining tagsets to partitions is performed using the number of tags shared among the tagset and the partition. Algorithm 2 is used also in this case to select the k initial tagsets setting the cost of each tagset to zero. The second phase of this set cover based algorithm (called **SCI**) is described in Algorithm 5. In each iteration, a random set is selected (Algorithm 5, Line 2) and added to the partition with which it shares the most tags (Algorithm 5, Line 3)

Algorithm 4: Set Cover Based Algorithms – Phase 2: Assigning the remaining tagsets balancing the processing load **(SCL)**

Input: Set $\mathcal{PR}$ of k initial tag partitions pr_i,
Set $\mathcal{S}$ of tagsets s_i,
Set $\mathcal{T}$ of sets of documents T_i
Output: Set $\mathcal{PR}$ of k final tag partitions pr_i

1 $CV = \bigcup_i pr_i$ `// Set of already covered tags`
2 **while** $\mathcal{S} \neq \emptyset$ **do**
3 $s_i = argmax_{s_j} l_j$ **and** $argmin_{s_j} |s_j \cap CV|$
4 $pr_i = argmin_{pr_j} \sum_{s_k \in pr_j} l_k$ **and** $argmax_{pr_j} |s_i \cap pr_j|$
5 $pr_i = pr_i \cup s_i$
6 $\mathcal{S} = \mathcal{S} \setminus \{s_i\}$
7 $CV = CV \cup s_i$
8 **end while**
9 **return** $\mathcal{PR}$

Algorithm 5: Set Cover Based Algorithms – Phase 2: Assigning the remaining tagsets as described in [1] **(SCI)**

Input: Set $\mathcal{PR}$ of k initial tag partitions pr_i,
Set $\mathcal{S}$ of tagsets s_i,
Set $\mathcal{T}$ of sets of documents T_i
Output: Set $\mathcal{PR}$ of k final tag partitions pr_i

1 **while** $\mathcal{S} \neq \emptyset$ **do**
2 $s_i = S.random()$
3 $pr_i = argmax_{pr_j}(s_i \cup pr_j)$
4 $pr_i = pr_i \cup s_i$
5 $\mathcal{S} = \mathcal{S} \setminus \{s_i\}$
6 **end while**
7 **return** $\mathcal{PR}$

5. THEORETIC PROPERTIES

The performance of the described algorithms depends on how well they can create similar-sized and non-overlapping partitions. We review these aspects by investigating the expected size of the biggest disjoint set of tags – which is crucial for the DS algorithm – and the expected degree of communication overhead for equally sized tag partitions. For additional details on the latter we refer to [1].

5.1 Number of Disjoint Sets

Assume a tagger that randomly annotates tweets with tags following uniform distribution. The derived graph G having one vertex for each tag and one edge for each pair of co-occurring tags can be described by the Erdős–Rényi graph model [9]. According to Erdős and Rényi's theory a graph G can be described either by the number of vertices n and the number of edges M, $G(n, M)$, or by the number of vertices n and the probability p than an edge between two vertices exists, $G(n, p)$. The number of edges M and the probability p are related with each other with $M = \binom{n}{2}p$.

Erdős and Rényi [9] derive properties of G, depending on the ratio between the number of vertices n and the edge probability p. For $np < 1$, the graph is expected to not have any connected component larger than $O(log(n))$, while for $np > 1$, it it likely to have one large component, and no other component contains more than $O(log(n))$ vertices (with a theoretical special case of $np = 1$, left out in the discussion).

We investigated the frequency of tweets with respect to the number of tags they contain using a sample of 15 million tweets received through Twitter's streaming API (on a randomly chosen day; Jan 28, 2012). The results showed that the number of tags used to annotated the tweets follows Zipf's law, i.e. no tags at all are the most popular case, one tag the second most popular case, and so on, with skew parameter $s = 0.25$.

A tweet annotated with m tags adds $\binom{m}{2}$ edges in the graph, one edge for each pair of co-occurring tags, and according to Zipf's law the frequency of tweets annotated with m tags, considering that a tweet can be annotated with m_{max} tags at most, is given by the formula $f(m, m_{max}, s) = \frac{\frac{1}{m^s}}{\sum_{i=1}^{m_{max}} \frac{1}{i^s}}$. The expected number of edges in $G(n, M)$ is computed as

$$E[M] := t \times \sum_{m=2}^{m_{max}} \left[f(m, m_{max}, s) \times \binom{m}{2} \right]$$

where t is the *distinct* number of tweets – as obviously duplicate tweets do not add any additional edges.

Out of the roughly 15 million tweets per day that we observed in the 10% sample obtained from Twitter, we observed around $600,000$ distinct tags in roughly $700,000$ distinct tweets. To have statistics about the full stream (100%), we assume the worst case for the DS algorithm: $600,000$ distinct tags, as in the 10% sample stream, but 7 million distinct tweets. Considering a graph over a 5 minute window of tweets leads to an np value of 0.76, if a maximal value of $m_{max} = 8$ tags per tweet is assumed. For a 10 minute window, we get $np = 1.52$, for the same m_{max}, but $np = 0.85$ for $m_{max} = 6$.

This model assumed independence of tags. If we actually count the number of distinct tag pairs (i.e., edges) that we see per day in the Twitter data (assuming the 100% stream has 10 times more of them than our 10% sample), we have around 5.5 million distinct pairs per day, i.e., around $34,000$ distinct tag pairs (edges) added per 10 minutes, which leads to $np = 0.11$ for the graph after 10 minutes, instead of $np = 1.52$ assuming independence of tag usages in the above model. We see that the above model is given a pessimistic behaviour of the expected disjoint sets, as the number of edges (tag pairs) is over estimated. On the other hand, it still describes pretty well how the various factors render the DS algorithm applicable or not – according to Erdős and Rényi's theory.

Further, as long as users select tags from topic-specific vocabularies, graph G falls apart in as many connected components as topics. This comes to rescue the DS algorithm to a large extent. However, if tags from multiple topic-specific vocabularies are mixed there is still the danger to have one large connected component. The existence of a large connected component is more likely when tweets from the more distant past are considered together with tweets from the more recent past since the content drift in tweets can cause mixing tags from different topics. Additionally, if tags from a joint vocabulary are used with probability $1 - \alpha$ a large connected component can develop for any $\alpha < 1$, with faster development for smaller values of α.

5.2 Communication

We presented the modeling of the expected communication overhead for equal-sized and randomly-created partitions in [1], where the expected number of partitions that are affected by a single tweet is studied. We call this the *communication load*, with a value of 1 indicating zero *com-*

munication overhead. Considering a vocabulary of v tags, n tweets over which the partitions are formed, k partitions and m tags per tweet, the expected communication load is given as

$$E[communication] := k \times \left(1 - \left[\frac{\binom{v-m}{m}}{\binom{v}{m}}\right]^{\frac{n}{k}}\right)$$

This means that for small vocabulary and large number of tags per tweet, each incoming tweet needs to be sent to (almost) all partitions; a knockout blow for any decentralised approach. For large vocabularies and few tags per tweet, as it is the case for Twitter data, the problem appears tractable. This is also verified by the experimental evaluation.

6. OPERATORS AND TOPOLOGY

For the implementation of our framework, we have used the distributed stream processing system Storm [19]. In this section, we briefly describe the basic characteristics of it and the specific implementation of our approach.

6.1 Storm

Storm is a framework that provides a fault-tolerant, distributed stream processing infrastructure, similar to what MapReduce/Hadoop is for batch processing. It can scale to big loads/rates of data by adding more machines for individual processing tasks running in parallel. Application developers have to *provide the implementation* of stream sources and operators, following the provided API.

Spouts, Bolts and Topologies.

Operators in Storm are organised in a graph, called topology. The nodes in the topology are connected to each other depending on how they communicate. The communication in Storm is performed following a push-based model. Storm distinguishes two kinds of nodes, **Spouts** and **Bolts**. Spouts are sources of streams. Bolts consume incoming streams, process them and emit new streams.

The application developer that defines a topology is specifying at the same time the number of instances for each operator. Storm creates that many threads for each operator to perform the processing across the cluster of machines. For each Bolt in the topology the developer should provide its functionality by implementing the *execute* method, which is invoked in every incoming tuple and, in general, outputs a new tuple to the consuming (subscribed) Bolts.

Data Flow Specification.

The operators in Storm work with tuples, i.e. simple lists of named values. Bolts consume data from Spouts or other Bolts by registering to their output streams. One of the key properties of Storm is to allow **multiple instances** of Spouts and Bolts. If multiple instances of a Bolt exist, Storm offers various rules that dictate how the tuples flow from producing Bolts or Spouts to the consuming multi-instance Bolt:

shuffle grouping: Tuples are distributed randomly over the various instances of the registered Bolt ensuring that each instance receives approximately the same number of tuples.

all grouping: Tuples are broadcasted, thus each instance gets all tuples.

fields grouping: Tuples are forwarded based on the values on one or multiple of their fields. It enables directing tuples based on their semantics/content.

local grouping: Tuples are forwarded on instances that reside on the same JVM.

direct grouping: Tuples are forwarded to specific instance (or instances) by using the instance's unique identifier.

6.2 Implemented Topology and Operators

The topology of our approach, implemented in Storm, is shown in Figure 2. It shows the basic operators – Source, Disseminator, Partitioner and Calculator – as introduced in Section 3 and a few more – Parser, Merger and Tracker. The numbers inside the circles indicate whether there is one (1) or multiple (n) instances/tasks created for each operator. In the following we describe in more details the implementation of all the operators.

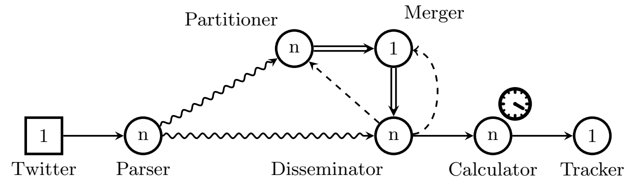


Figure 2: Topology

Parser.

The Source (Spout) in our implementation produces a stream of Twitter tweets, either based on live data through Twitter's streaming API or for repeatability of experiments read from a file. Tweets are sent using shuffle grouping to one of the multiple instances of the Parser Bolt. Parser Bolts are responsible to extract for each tweet d_i a set of tags s_i containing the hashtags used by the users to annotate it. This tagset can be enriched with named entities, location, or sentiment, extracted from the messages' body and interpreted as additional tags. For each incoming tweet d_i the Parser emits a tuple of the form $(timestamp_i, s_i)$, where $timestamp_i$ is the time of the arrival of d_i in the system. Disseminator and Partitioner both register to Parser.

Partitioner.

The Partitioner operator is responsible to create the tag partitions using one of the algorithms presented in Section 4. To accelerate the procedure of partitioning multiple instances of this operator can be created. Each Partitioner instance receives tuples from the Parser using *field grouping* on the whole tagset s_i. This way the same tagsets are forwarded always to the same Partitioner instances.

Partitioners maintain a sliding window (cf. e.g. [14]) of size W over the incoming tagsets. Conceptually, this window can be time-based (e.g. capturing 5 minutes of tweets) or count-based (e.g. 10000 tweets). When the Partitioners are asked to create partitions they use the tagsets currently in the window. The creation of new partitions is triggered by the Disseminator when the quality of the current partitions has deteriorated significantly.

Partitioners emit tuples of the form $(\{pr_1, pr_2, ..., pr_k\})$.

Merger.

With multiple Partitioners present, the final number of created partitions amounts to more than k, the requested number of partitions. This creates the need for an additional operator, the Merger, which takes the partitions from the Partitioners and creates the final k partitions.

The Merger can be viewed as another Partitioner. It receives tagsets and outputs tag partitions. The tagsets it re-

ceives are the tag partitions created by the Partitioners. Following this principle, the Merger creates the final partitions using the same algorithm the Partitioners use.

To preserve the general idea of the Disjoint Sets algorithm when executing it in the Merger, we make a slight change in the Disjoint Sets (DS) Algorithm executed in the Partitioners. Partitioners execute only the first part of it, i.e. they create *all* possible disjoint sets but do not merge them into k partitions. The Merger receiving these sets is thus able to combine them into bigger disjoint sets and merge them afterwards into k final partitions.

The Merger sends the final partitions to the Disseminator and waits for messages from it regarding tagsets seen in the input but not found in any of the partitions. For any such tagset, sent by the Disseminator, the Merger finds the best fitting partition and informs back the Disseminator about its decision. This procedure is described in detail in Section 7.

Disseminator.

The Disseminator receives the partitions from the Merger and uses them to create an index from tags to Calculators (Section 3). For every tagset s_i, sent by the Parser using *shuffle grouping*, the Disseminator searches this index for the Calculators that have been assigned any of the tags in s_i. It sends a tuple of the form (s_i^j) to each of the involved Calculators. s_i^j is a subset of s_i containing all tags assigned to Calculator j. For example, suppose that $s_i = \{a, b, c\}$ and Calculator 1 is assigned the tags a, b, c and Calculator 2 is assigned the tags a, c. The Disseminator will output the tuples $(\{a, b, c\})$ and $(\{a, c\})$, each one delivered to the appropriate Calculator using *direct grouping*. These messages are called *notifications*. To accelerate the notification of Calculators for the received tagsets multiple instances of the Disseminator operator can be created.

Apart from notifying the Calculators, the Disseminator is responsible to inform the Merger about the tagsets in the input that are not reflected in the partitions. Additionally, they monitor the partitions and trigger repartitions when the quality of them, with respect to communication overhead and processing load, is not any more acceptable. More details on this are provided in Section 7.

Calculator.

Calculators are responsible to compute the Jaccard coefficients for a set of co-occurring tags. They register to the Disseminators and receive from them tuples of the form $(\{t_1, t_2, \ldots, t_n\})$ using *direct grouping*.

Calculators are oblivious to the tags they have been assigned. They infer the information about the sets of co-occurring tags for which they should compute a Jaccard coefficient from the messages they receive from the Disseminators. Calculators compute the Jaccard coefficients as describe in Section 3. For this, each Calculator should know the cardinalities of the intersections of co-occurring tags and thus this is the information stored by them.

Consider for example that Calculator 1 receives the tuple $(\{a, b, c\})$. From that it infers that it should compute the Jaccard coefficient for the tagsets $\{a, b, c\}, \{b, c\}, \{a, b\}, \{a, c\}$ (i.e all subsets of tags included in the received tuple). For each of these tagsets it creates a counter. If the counter already exists it updates it increasing it by one.

Every y time units the Calculators use the counters to compute the maximum possible number of Jaccard coefficients. The coefficients are emitted and the counters are deleted.

Tracker.

The Tracker operator receives the Jaccard coefficients emitted by the Calculators and uses them to perform further computations. If the same tags are assigned to multiple partitions it might happen that multiple Calculators emit Jaccard coefficients for the same tagset simultaneously. In such a case, the Tracker should select one of them for further usage. We opted for the coefficient computed over data tracked for a longer period. For this reason, Calculators emit tuples of the form $(s_i, J(s_i), CN(s_i))$. $J(s_i)$ is the Jaccard coefficient for s_i and $CN(s_i)$ is the value of the counter used for s_i. When receiving multiple tuples for the tagset s_i, the Tracker keeps the one with the maximum $CN(s_i)$. This heuristic guarantees that at least all tagsets assigned to the partitions during the creation of them will have a correct Jaccard coefficient not mixed with a Jaccard coefficient computed in a Calculator as a result of the evolution of the partitions.

7. HANDLING DYNAMICS

Twitter is highly dynamic. Old topics evolve through time and new topics appear very frequently introducing new tags and tag combinations. These dynamics are even more acute when focusing on a small subset of the data, i.e. data obtained during the last 5 or 10 minutes. However, theory and real data show that it is not feasible to create partitions over large windows as the existence of a large number of tweets causes the DS algorithm to break due to a large connected component (Section 5.1), while the set-cover–based algorithms suffer from large amounts of redundant communication (Section 5.2). In our setting, we identify and handle the following two requirements:

Evolving Partitions: tags and tag co-occurrences not reflected in the partitions are continuously seen in the input. Triggering the recreation of partitions for each of them is clearly not feasible. Instead they are incrementally added to the existing partitions.

Partition Quality Monitoring: enriching partitions with additional tags affects the quality of them in terms of communication overhead and load balance. Identifying that the quality is not within acceptable limits anymore and creating new partitions is necessary.

As Disseminators connect the two logical parts of our approach, the creation of the partitions and the computation of the Jaccard coefficients, they have a central role in addressing the above two points.

7.1 Evolving Partitions

Disseminators directly notice when a tagset s_i is not found in any of the partitions since they cannot find a Calculator being assigned all tags in it. If s_i is seen in the input sn times without being found in any Calculator the Disseminator informs the Merger about it and the Merger adds it in the best possible partition. The best partition to add a tagset depends on the partitioning algorithm. For the DS, SCC and SCI algorithms the partition is selected so that the increase in communication is minimised. For the SCL algorithm the partition is selected so that the load in the Calculators remains as balanced as possible. The addition of a tagset to some partition is called *Single Addition*. The occurrence threshold sn is ideally small, not to miss important tagsets, but large enough to ignore spam or typos.

When a Single Addition is performed the Disseminators receive a message from the Merger telling them the Calculator that was assigned the tagset. Disseminators use this message to update their indices. All Disseminators receive

the message independently of whether they asked for the addition or not.

7.2 Partition Quality Monitoring

Every time a new partition is created the Merger notifies the Disseminators about it sending to each of them a tuple of the form $(partitions, avgCom, maxLoad)$. Disseminators use $partitions$ to create their index. $avgCom$ and $maxLoad$ contain the average communication and the maximum load of the created partitions as computed immediately after their creation. Disseminators use $avgCom$ and $maxLoad$ as *reference values* to ensure that the quality of the partitions through time remains within acceptable bounds.

In order to estimate the quality of the partitions at any time the Disseminators maintain some statistics representing the current average communication $avgCom'$ and maximum load $maxLoad'$ of the partitions. For each received tagset s_i the Disseminators store for each Calculator whether a notification was sent to it or not. Consider the example where $s_i = \{a, b, c\}$, Calculator C_1 is assigned $\{a, b, c\}$ and Calculator C_2 is assigned $\{a, c\}$. Assume there is one more Calculator C_3 not assigned any of the tags in s_i. Two of the Calculators, the Calculators C_1 and C_2, receive a notification and one Calculator, Calculator C_3 does not. Only tagsets for which there was at least one notification sent are considered in the statistics.

When z tagsets have been considered, the Disseminators compute the average sent notifications $avgCom'$ as the sum of all sent notification divided by z. $maxLoad'$ is computed as the percentage of the most notifications sent to a single Calculator. As long as both $avgCom'$ and $maxLoad'$ do not exceed $avgCom$ and $maxLoad$ respectively more than a threshold thr the Disseminators reset the statistics and continue using the existing partitions. Otherwise, the Disseminators ask from the Partitioners to create new partitions. Increasing thr decreases the number of repartitions allowing for worse performance with respect to communication overhead or processing load.

7.3 Topology Scaling

In the used version (v0.8.2) of Storm a reconfiguration of a running topology is not possible. Storm only allows to rebalance a topology creating more threads (executors) or workers but does not allow changing the number of instances (tasks) for each operator. In order to be able to adjust the number of Calculators to the observed load the necessary logic should be implemented in the operators. The maximum expected number of Calculators should be defined before submitting the topology. The Partitioners can specify the actual number of Calculators that are used at any time by adjusting the number of partitions they create. Only Calculators that are assigned a partition are indexed by the Disseminators, receive documents and compute Jaccard coefficients.

8. EXPERIMENTS

We have implemented the above operators in Java 1.7. For the experiments we used a cluster of 26 Linux (3.12.0) servers, each running Storm 0.8.2 (with Zookeeper 3.4.5). Each machine has an Intel quad-core i7-2600K CPU@3.4GHz and 16GB RAM.

The dataset we used consists of 6 hours of tweets obtained through the **Twitter's streaming API** on September 5, 2013. In all experiments, we compare four algorithms: Set Cover Based focusing on communication (**SCC**), Set Cover Based focusing on processing load (**SCL**), Disjoint Sets (**DS**) and the approach in [1] (**SCI**), which we described in detail in Section 4.

8.1 Parameters

Number of partitions k: We set the number of partitions (the number of calculators in the topology) to 5, 10 and 20. In general, keeping the communication low while maintaining the processing load equally distributed are contradicting goals. Keeping the load in each Calculator close to the average means that tagsets sharing tags have to be assigned to different partitions and keeping the communication low means that tagsets sharing tags should be assigned to the same partitions resulting in unbalanced partitions. The attempt to balance the two measures is becoming harder as the number of partitions increases.

Number of Partitioners P: We set the number of Partitioners that in parallel try to create the partitions to 3, 5 and 10.

Data arrival rate tps: In real world 1300 tweets are created every second. For that we set the arrival rate of tweets on the system to 1300 and 2600 tweets per second.

Repartition Threshold thr: Repartition threshold defines the maximum percentage of change that is considered acceptable by the Disseminators. When the processing load or the communication have changed more than the provided threshold the Disseminators ask for repartitioning. We set the repartition threshold to 0.5 and 0.2. A threshold of 0.5 means that the communication (or processing load) is 50% worse than the communication (processing load) when the partitions were computed.

8.2 Experimental Results

The results for the various parameters show the same trend. For this reason we decided to set the parameters to a specific value and show the corresponding results. More specifically, unless otherwise mentioned to be varied, we set the parameters to the following values *P=10, k=10, thr=0.5, tps=1300.* All configurations use one Parser and one Disseminator.

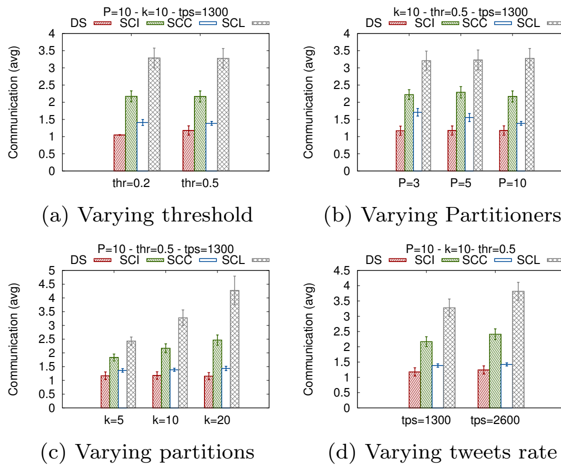
The Disseminator asks for a Single Addition when a tagset is seen in the input without being found in any Calculator 3 times. The statistics used to estimate the quality of the partitions (average communication and maximum load) are computed for every 1000 tweets for which there was a notification sent. Calculators report the Jaccard coefficients every 5 minutes. The Partitioners create the partitions using the set of tweets seen in the previous 5 minutes.

8.2.1 Communication

We define the Communication to be the average number of messages sent from the Disseminator to Calculators for each received tagset. We do not consider tagsets which are not found in any Calculator and, thus, do not cause any message to be sent. The plots in Figure 3 show how the change in each parameter affects the communication.

We see that the number of partitions plays the most important role in the communication (Figure 3c). Having more partitions makes it difficult to assign tags to them without making partitions overlap. This is not true for DS which creates the partitions using only disjoint sets. For this reason DS has in general the best performance with respect to communication. On the other hand, SCL which focuses mainly on balancing the processing load shows the worst performance with respect to communication. Surprisingly SCI performs significantly worse than SCC although the algorithms are very similar in principle. This justifies our choice

to develop a new algorithm with the communication as its primary optimisation criterion.



(a) Varying threshold (b) Varying Partitioners

(c) Varying partitions (d) Varying tweets rate

Figure 3: Communication

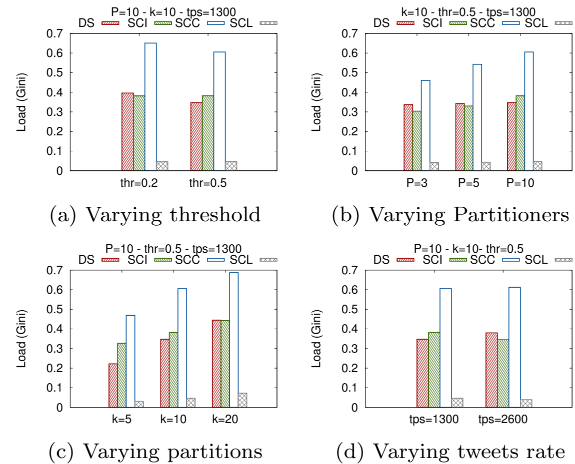
8.2.2 *Processing Load*

We define the Processing Load of a single Calculator C_i to be the percentage of notifications sent to C_i out of the total sent notifications. To measure the inequality of Processing Load in the various Calculators we use the Gini coefficient. The Gini coefficient is one of the most common measures of statistical dispersion and it is defined mathematically based on the Lorenz curve which depicts the cumulative proportion of ordered individuals mapped onto the corresponding cumulative proportion of their size. A small value of the Gini coefficient indicates a more balanced distribution.

The plots in Figure 4 show how the various parameters affect the load distribution. SCL with primary optimisation criterion the balanced processing load shows the best performance. The parameter that affects the load balance most is the number of partitions (Figure 4c). The reasoning is similar to that used for the communication. Having more partitions makes it more difficult to balance the load on them without increasing substantially the communication. Interestingly SCC, in contrast to SCI, is affected also by the number of Partitioners (Figure 4b). The difference between SCC and SCI is that SCI randomly chooses the next tagset to be added to some partition while SCC selects the more appropriate tagset to be added to some partition. The plot in Figure 4b suggests that the careful selection of the next tagset although keeps communication low (cf. Figure 3b) cannot help load balance.

8.2.3 *Jaccard Accuracy*

The Calculators can compute the Jaccard coefficient only for the tagsets they have been assigned. During the partitioning we make sure that all tagsets present in the data are assigned to some Calculator. However, as new documents are received new tag combinations arise. The Disseminator waits until it has seen such a tagset $sn = 3$ times before asking for a Single Addition. After the Single Addition is completed the tagset might or might not be seen in the input again. In case it is seen again, then a Jaccard coefficient will be reported that will deviate from the correct coefficient since until the addition is completed none of the Calculators tracks the counter needed for the tagset. In case it is not seen again, a Jaccard coefficient will not be computed at all



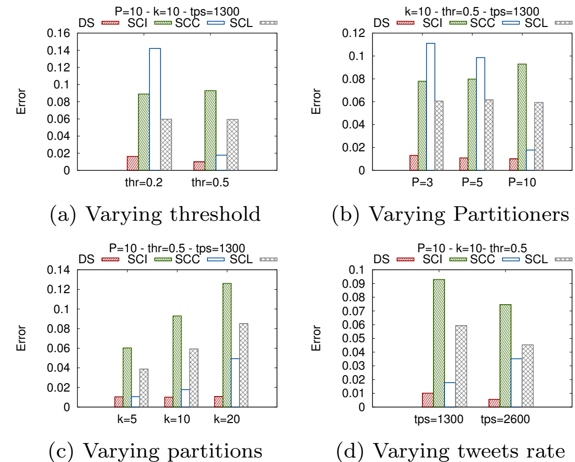
(a) Varying threshold (b) Varying Partitioners

(c) Varying partitions (d) Varying tweets rate

Figure 4: Processing Load

resulting in missing completely the information about this tagset. To measure the error in accuracy and the total loss of coefficients we implemented a centralised approach which gets all tagsets and computes their Jaccard coefficients. We use the centralised results as our baseline. Since a tagset is added when seen at least 3 times the centralised approach considers only tagsets appearing more than 3 times.

Our experiments showed that all algorithms manage to compute a Jaccard coefficient for more than 97% of the tagsets seen more than 3 times in the input. In Figure 5 we report on the average error of these Jaccard coefficients compared to the Jacacrd coefficients computed for the same tagsets by the centralised approach.



(a) Varying threshold (b) Varying Partitioners

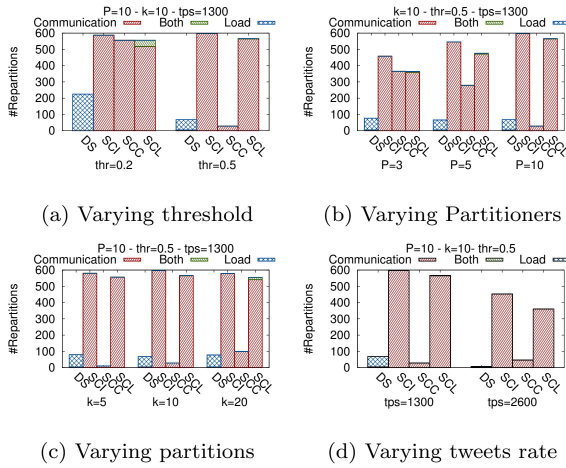
(c) Varying partitions (d) Varying tweets rate

Figure 5: Error for tagsets seen more than 3 times

In general, DS is the algorithm computing the most accurate coefficients. It is interesting that an increase in the number of Partitioners causes a significant reduction in the error of SCC (Figure 5b). Additionally, an increase in the rate decreases the error in all algorithms but SCC (Figure 5d). Both these cases are related to the number of repartitions (see plots in Figure 6). When a repartition happens it might be the case that tagsets assigned to some Calculator before it are assigned to another Calculator after it. This causes multiple coefficients for the same tagsets being reported while none of them is accurate.

8.2.4 *Number of Repartitions*

The number of repartitions depicts how Single Additions affect the quality of the partitions. A repartition is triggered if either the communication or the processing load exceeds the threshold. In some cases both measures had been found to have exceeded the threshold. The plots in Figure 6 show the effect of the various parameters on the number of repartitions. As expected, DS does repartitions caused by load imbalance. SCC and SCI although focusing on communication, similarly to DS, have repartitions caused by big communication overhead. What is interesting is that SCL and SCI do not manage to drop the number of repartitions for bigger threshold (Figure 6a) . This contradicts our expectations and suggests that it is very difficult in general for these algorithms to maintain acceptable communication.



(a) Varying threshold (b) Varying Partitioners

(c) Varying partitions (d) Varying tweets rate

Figure 6: Number of Repartitions

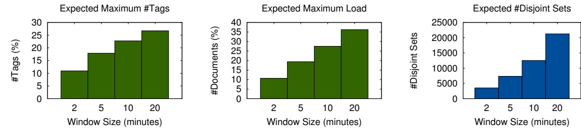
8.2.5 *Evolution of Partition Quality*

The plots in Figures 8 and 9 show the changes in communication and processing load with evolving time. For the communication the average communication is used while for the processing load we show the detailed load in each Calculator. The processing load has been sorted so that one line has always the load of the most loaded Calculator, another line has the load of the second most loaded Calculator and so on. One vertical line has been drawn representing the points when a repartition was performed.

The plots regarding DS clearly show the effect of the Single Additions and the repartitions to the communication (Figure 8a) and processing load (Figure 9a). As long as there are repartitions happening the communication stays low while there is one Calculator having more load that the others. Between repartitions the communication increases and processing load tends to become more balanced, i.e. the load of the most loaded Calculator decreases, until the next repartition when communication decreases again and load becomes more unbalanced. Similar results are seen for SCC in Figures 8b and 9b. For SCL and SCI the results are not that clear since there are very many repartitions (there is approximately one repartition every 2750 processed documents). However, for SCL Figure 9c clearly shows that the processing load is balanced through the whole experiment.

8.2.6 *Connectivity of Tagsets*

In Figure 7, we report some statistics regarding the Twitter dataset as they are fundamentally related to the problem we consider in this work and, in particular, relevant for the performance of the DS algorithm. For the measurements, we used the same tweets we used for the experiments. Over them we defined non-overlapping sliding windows of 4 different sizes (2, 5, 10 and 20 minutes). Every time the window slides we measure the maximum percentage of tags contained in a single connected component of tags and the maximum number of documents related with a single connected component.



(a) Maximum size of connected tagset per round (b) Maximum load of connected tagset per round (c) Number of connected tagsets per round

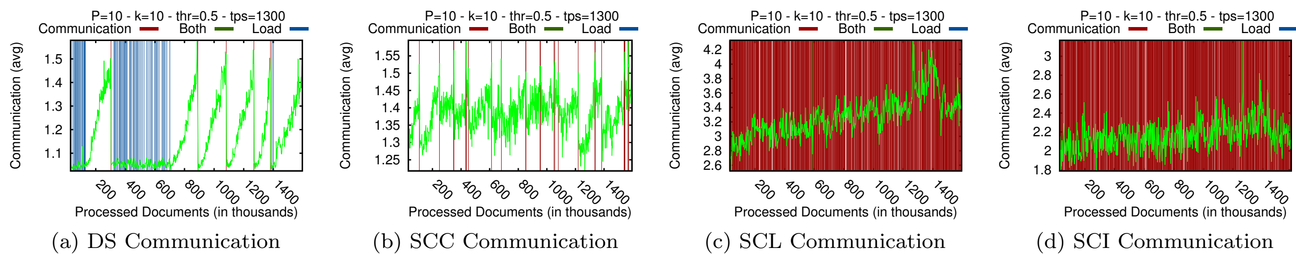
Figure 7: Tagsets connectivity and load

8.3 Lessons Learned

We observe by the experimental evaluation, supported by the theoretic reasoning in Section 5, that our presented approach is feasible for Twitter-style data characteristics. It further has proven doable to tune the set-cover–based methods according to which cost (communication or processing) is most critical. The disjoint sets (DS) algorithm has by design optimal communication load (zero redundancy) but in practise and theory can reach scenarios where the load imbalances are too drastic, given one (or few) large evolving disjoint sets. If that happens, splitting the large disjoint sets is required, but current data characteristics did not exhibit such a case. Thus, as the rule of thumb, for current, Twitter-scale streams, DS is the method of choice, particularly since it reaches the most accurate results as fewer tagsets are assigned to multiple nodes, thus, there are fewer Jaccard coefficients reported multiple times. However, we did see that some partitions in the DS algorithm have higher load than others, so if balancing load is crucial and network cost is not critical, SCL should be used instead of DS. Ultimately, disjoint sets should form the basis of all partitioning algorithms, but large ones need to be split (to not impair the load balancing), for instance by applying set-cover–based algorithms like SCL.

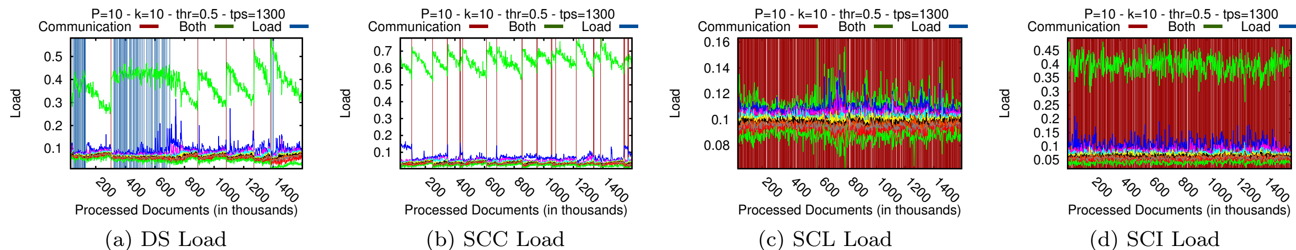
9. CONCLUSION

Detecting shifts in correlation between annotations in social media content is a common technique to analyse streams of social media, most importantly Twitter. With data arriving at high rates is becomes computationally infeasible to compute co-occurrence statistics on a single machine or semantically too restrictive to impose a fixed, small tag vocabulary to focus on. In this work, we addressed the problem of computing correlation statistics for tagsets arriving in a massive data stream, involving the consideration of all subsets of co-occurring tags with no restriction on the used tags. The key idea was to spread the computation to multiple machines in a computing cluster by continuously defining partitions of tags according to which the data is spread to counting nodes. We implemented a prototype of our approach using the distributed stream processing engine Storm and evaluated several partitioning algorithms together with the generic behaviour of the system under realistic load, using data obtained through the Twitter streaming API. Overall, we showed in theory and practise that it is feasible to track the ocurrences/correlation of all existing tagsets in



(a) DS Communication (b) SCC Communication (c) SCL Communication (d) SCI Communication

Figure 8: Communication over Time



(a) DS Load (b) SCC Load (c) SCL Load (d) SCI Load

Figure 9: Processing Load over Time

Twitter-scale data streams. We believe this is an enabling step toward high-quality, large-scale social trend mining as no ad-hoc pruning of the tagset space is anymore required.

ACKNOWLEDGMENTS

We thank the anonymous reviewers at SIGMOD for their constructive feedback that helped improving this work and the Database Architectures group at CWI Amsterdam for providing access to their computing cluster. This work has been partially supported by the Excellence Cluster on Multimodal Computing and Interaction (MMCI) and the German Research Foundation (DFG) under grant MI 1794/1-1.

10. REFERENCES

[1] F. Alvanaki and S. Michel. Scalable, continuous tracking of tag co-occurrences between short sets using (almost) disjoint tag partitions. *DBSocial*, 2013.

[2] F. Alvanaki, S. Michel, K. Ramamritham, and G. Weikum. See what's enblogue: real-time emergent topic identification in social media. *EDBT*, 2012.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7), 1970.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[5] G. Cormode. Count-min sketch. *Encyclopedia of Database Systems*, 2009.

[6] W. Donath and A. Hoffman. Algorithms for partitioning graphs and computer logic based on eigenvectors of connection matrices. *IBM Technical Disclosure Bulletin*, 15(3), 1972.

[7] M. Eftekhar and N. Koudas. Partitioning and ranking tagged data sources. *PVLDB*, 6(4), 2013.

[8] D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. *Algorithms and Theory of Computation*, 1999.

[9] P. Erdős and A. Rényi. On the evolution of random graphs. In *Publication of the Mathematical Institute of the Hungarian Academy of Sciences*, 1960.

[10] S. Helmer and G. Moerkotte. A performance study of four index structures for set-valued attributes of low cardinality. *VLDB J.*, 12(3), 2003.

[11] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. *Supercomputing*, 1995.

[12] B. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell Systems Technical Journal*, 49(2), 1970.

[13] S. Khuller, A. Moss, and J. Naor. The budgeted maximum coverage problem. *Inf. Process. Lett.*, 70(1), 1999.

[14] J. Krämer and B. Seeger. Semantics and implementation of continuous sliding window queries over data streams. *ACM Trans. Database Syst.*, 34(1), 2009.

[15] M. Mathioudakis and N. Koudas. Twittermonitor: trend detection over the twitter stream. *SIGMOD*, 2010.

[16] A.-M. Popescu, M. Pennacchiotti, and D. Paranjpe. Extracting events and event descriptions from twitter. *WWW (Companion Volume)*, 2011.

[17] A. Ritter, Mausam, O. Etzioni, and S. Clark. Open domain event extraction from twitter. *KDD*, 2012.

[18] B. Sriram, D. Fuhry, E. Demir, H. Ferhatosmanoglu, and M. Demirbas. Short text classification in twitter to improve information filtering. *SIGIR*, 2010.

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

[20] K. Watanabe, M. Ochi, M. Okabe, and R. Onai. Jasmine: a real-time local-event detection system based on geolocation information propagated to microblogs. *CIKM*, 2011.

[21] A. Weiler, S. Mansmann, and M. H. Scholl. Towards an advanced system for real-time event detection in high-volume data streams. *PIKM*, 2012.