Mining Interesting Events on Large and Dynamic Data

Dissertation
zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Foteini Alvanaki

Universität des Saarlandes

Saarbrücken
2014

Dekan der Naturwissenschaftlich-Technischen Fakultät I	Prof. Dr. Markus Bläser
Vorsitzender der Prüfungskommission	Prof. Dr. Jens Dittrich
Berichterstatter	Prof. Dr.-Ing. Sebastian Michel
Berichterstatter	Prof. Dr.-Ing. Gerhard Weikum
Berichterstatter	Prof. Dr. Alexis Delis
Beisitzer	Dr. Lili Jiang
Tag des Promotionskolloquiums	22.12.2014

Abstract

Nowadays, almost every human interaction produces some form of data. These data are available either to every user, e.g. images uploaded on Flickr or to users with specific privileges, e.g. transactions in a bank. The huge amount of these produced data can easily overwhelm humans that try to make sense out of it. The need for methods that will analyse the content of the produced data, identify emerging topics in it and present the topics to the users has emerged. In this work, we focus on emerging topics identification over large and dynamic data. More specifically, we analyse two types of data: data published in social networks like Twitter, Flickr etc. and structured data stored in relational databases that are updated through continuous insertion queries.

In social networks, users post text, images or videos and annotate each of them with a set of tags describing its content. We define sets of co-occurring tags to represent topics and track the correlations of co-occurring tags over time. We split the tags to multiple nodes and make each node responsible of computing the correlations of its assigned tags. We implemented our approach in Storm, a distributed processing engine, and conducted a user study to estimate the quality of our results.

In structured data stored in relational databases, top-k group-by queries are defined and an emerging topic is considered to be a change in the top-k results. We maintain the top-k result sets in the presence of updates minimising the interaction with the underlying database. We implemented and experimentally tested our approach.

Zusammenfassung

Heutzutage entstehen durch fast jede menschliche Aktion und Interaktion Daten. Fotos werden auf Flickr bereitgestellt, Neuigkeiten über Twitter verbreitet und Kontakte in Linkedin und Facebook verwaltet; neben traditionellen Vorgängen wie Banktransaktionen oder Flugbuchungen, die Änderungen in Datenbanken erzeugen. Solch eine riesige Menge an Daten kann leicht überwältigend sein bei dem Versuch die Essenz dieser Daten zu extrahieren. Neue Methoden werden benötigt, um Inhalt der Daten zu analysieren, neu entstandene Themen zu identifizieren und die so gewonnenen Erkenntnisse dem Benutzer in einer übersichtlichen Art und Weise zu präsentieren. In dieser Arbeit werden Methoden zur Identifikation neuer Themen in großen und dynamischen Datenmengen behandelt. Dabei werden einerseits die veröffentlichten Daten aus sozialen Netzwerken wie Twitter und Flickr und andererseits strukturierte Daten aus relationalen Datenbanken, welche kontinuierlich aktualisiert werden, betrachtet.

In sozialen Netzwerken stellen die Benutzer Texte, Bilder oder Videos online und beschreiben diese für andere Nutzer mit Schlagworten, sogenannten Tags. Wir interpretieren Gruppen von zusammen auftretenden Tags als eine Art Thema und verfolgen die Beziehung bzw. Korrelation dieser Tags über einen gewissen Zeitraum. Abrupte Anstiege in der Korrelation werden als Hinweis auf Trends aufgefasst. Die eigentlich Aufgabe, das Zählen von zusammen auftretenden Tags zur Berechnung von Korrelationsmaßen, wird dabei auf eine Vielzahl von Computerknoten verteilt. Die entwickelten Algorithmen wurden in Storm, einem neuartigen verteilten Datenstrommanagementsystem, implementiert und bzgl. Lastbalancierung und anfallender Netzwerklast sorgfältig evaluiert. Durch eine Benutzerstudie wird darüber hinaus gezeigt, dass die Qualität der gewonnenen Trends höher ist als die Qualität der Ergebnisse bestehender Systeme.

In strukturierten Daten von relationalen Datenbanksystemen werden Beste-k Ergebnislisten durch Aggregationsanfragen in SQL definiert. Interessant dabei sind eintretende Änderungen in diesen Listen, was als Ereignisse (Trends) aufgefasst wird. In dieser Arbeit werden Methoden präsentiert diese Ergebnislisten möglichst effizient instand zu halten, um Interaktionen mit der eigentlichen Datenbank zu minimieren.

Summary

In this work, we provide methods to identify emerging topics in big and dynamic data. We focus on two types of data, stream data obtained from Web 2.0 sources and stream data obtained from queries executed in business intelligence applications. The former are read only once, at the moment of their production, while the latter are permanently stored in a relational database. These two types of data are considered to split this work in two conceptual parts.

In the first part we identify emerging topics over Web 2.0 sources. Over the years social networks like Twitter[1], Youtube[2], Flickr[3], Tumblr[4], etc. have gained great popularity. The common characteristic of all these sites is that users create profiles and post messages, videos or images, each one accompanied by a set of tags. The use of these tags enables the representation with text even of non-textual messages, e.g. videos or images. The tags are usually carefully selected by the users to express the content or topic of each post. The sites facilitate search mechanisms over these tags that allow users to find posts annotated with specific tagsets that comply to their interests. Users can continuously query these sites for new posts and try to identify the topics that are new, i.e. topics that were not discussed in the past but are discussed now. However, we believe that it would be more convenient for them to be automatically notified about the newly discussed topics. Twitter, for example, provides users with a list of the most popular tags at any time. These tags can be used by the users to get an overview of what is currently hot in Twitter. We argue that a hot topic is not necessarily new and propose methods that identify new topics as soon as they start being discussed. The identified emerging topics could be obtainable through a public website allowing users to have access to them without the need to create a profile to yet another website.

We define topics in Web 2.0 as combinations of tags that co-exist in the published posts and define a correlation measure that reflects how strongly related the tags are. We define a topic to be emergent if the tags representing it show an unexpectedly high correlation. To estimate that, we monitor the occurrences of co-occurring tagsets over time. At each time point, we use the observed data to compute the correlation value for each tagset. In addition, we predict the correlation value of each tagset using the exponential smoothing technique. The comparison of the predicted correlation value to the computed correlation value provides us with an estimation of the degree of surprise involved with each topic. We compare our approach with TwitterMonitor [MK10], an

[1]http://twitter.com/
[2]http://www.youtube.com/
[3]http://www.flickr.com/
[4]http://www.tumblr.com/

earlier approach similar to ours that considers topics to be represented by groups of bursty tags. We conduct a user study to estimate the quality of the topics identified by the two approaches.

The quality of the identified topics is an important aspect of the problem we study in this thesis and the majority of works in this area focus on identifying precise topics. This sometimes results in overlooking the efficiency aspects of the problem. With a vast amount of produced data, methods that are able to efficiently process them in real time are necessary. In this work, we describe a detailed implementation of our approach that allows identifying the emerging topics shortly after they appear in the social networks. The implementation we propose is based on Storm[5], a distributed stream processing engine that allows to specify various operators and give directions on how they communicate with each other. The user has to provide the functionality of the operators and the general characteristics of the distributed setting, e.g. the number of the machines. Storm assigns the operators to the machines and implements the communication of them in the physical level. The general idea of our approach is to partition the tags to the various machines based on their co-occurrences. New posts are forwarded to the machines according to the created partitions. Each machine computes the correlation of each of the co-occurring tagsets found in its assigned partition. We propose various algorithms on the tags partitioning. The algorithms attempt to minimise the communication overhead in the network and balance the processing load in the machines. We present theoretic estimations on the performance of the partitioning algorithms and evaluate our approach conducting a thorough experimental study.

In the second part of this work, we focus on emerging topics over data stored in relational databases. More specifically, we focus on a data warehouse environment. In this setting, there are two types of databases: the operational database which is accepting new data and is configured to provide online transaction processing (OLTP) with indexing, concurrency control and so on, and the data warehouse database which stores summaries over the data and is configured to provide online analytic processing (OLAP) where the queries access a large portion of the stored data. In this thesis, we define a set of top-k group-by queries over the data stored in the operational database. The queries associate groups having specific characteristics to an aggregate score over a measure of interest creating a set of top-k rankings. The rankings are created with respect to the organisation of the data in the data warehouse. An emergent topic is defined to be a change in a top-k ranking. A change can either be a new group entering the ranking or an old group climbing up the positions in the ranking. In order to spot these changes, the rankings need to be maintained against all incoming updates.

We propose two algorithms that attempt to make the top-k rankings self-maintainable. These algorithms minimise the interaction with the underlying database materialising N more than the requested k groups and assigning an estimated score to each group, not present in the top-(k+N) results. The estimated scores guarantee that, at any time, the real score of each group can only be worse than its estimated score, thus a group currently in the top-k results is never missed. We exploit the star or data cube schemas found in data warehouses and share results among various top-k rankings using techniques similar

[5]http://storm-project.net/

to those used in multi-query optimisation. This allows us to further minimise the need to query the stored data. We evaluate the benefits of our approach through a thorough experimental study over the TPC-H[6] dataset using updates specifically created to highlight the benefits and drawbacks of our algorithms.

[6]http://www.tpc.org/tpch/

Contents

Chapter 1

Introduction

In 2013, Twitter alone contributed on average 190 million tweets every day and 3000 images were uploaded to Flickr every minute[1]. With all these available data, it is difficult for the users to identify what is of interest to them. This creates the necessity for methods that collect and analyse the data, providing the user with an easily conceivable summary of it. Undoubtedly, users are more interested in new information, on events that are suddenly and unexpectedly arising. Over the past years, the research community has published a number of works on identifying topics discussed on blogs, news portals and microblogs (e.g. [SHM09, BNG09, PPP11]). The focus on many of these works is mainly on identifying topics of high quality over data collected in the past, overlooking any efficiency problems. We focus on identifying efficiently and in real time emerging topics, or events, that are intelligible to the users.

User generated content in Web 2.0 is usually annotated with short descriptive text that specify the topic of a tweet, the content of a video etc. These annotations are, in general, called tags. In Twitter, specifically, they are called hashtags due to the special character # used in front of each such tag. We do not distinguish between tags and hashtags and use the terms interchangeably. We define topics to be sets of tags and compute correlation statistics over each such set. We consider an event to be a change in the correlation of the tags representing it that was not expected based on the correlation values observed during the recent past. A novel distributed implementation of general purpose stream processing framework allows us to efficiently process the vast amount of produced data and present an emerging topic to the users shortly after a few messages about this topic have been published.

We extend the event identification process to more traditional settings, i.e. business intelligence storing data in a relational database. In such a setting, we provide users with summaries over the data in the form of top-k rankings. Top-k rankings allow users to focus on the essence of the information. Events in top-k rankings are defined to be either changes in the relative positions of the elements already in the ranking or new elements, outside the ranking, replacing elements inside the ranking. To identify these events, the old data are permanently stored and considered together with the new streaming data. The bottleneck in such approaches is the continuous interaction with the underlying

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

database. We focus on developing algorithms that identify events efficiently and in real time minimising this interaction. Reducing the burden imposed to the database from the event identification procedure allows it to use its resources to execute other jobs, e.g. create indices.

1.1 Problem Statement

We consider a stream $\mathcal{D}$ of incoming data elements d_i and statistics computed over them. Each data element d_i is seen in the data stream multiple times at different time points and each occurrence of it is associated with some additional information. The data elements can either be tagsets found in new published messages in social media or sets of instances of database attributes that are affected by updates. In the case of tagsets used in social media, the additional information is the number of times the tagset is seen in the set of messages published during the recent past. In the case of sets of instances of database attributes, the additional information is the change imposed by the update to some numeric attribute associated with the specified set of instances. The additional information found in each data element occurrence is used to compute statistics about it.

Our goal is, for each received data element to interpolate the statistics that have been computed using previous occurrences. Additionally, we want to compute new statistics about each received data element that take under consideration the latest information about it. Comparing the interpolated to the computed statistics we can identify unexpected behaviour in the data elements. We want the above procedure to be performed efficiently and in an online manner, allowing for the identification of unexpected behaviours (i.e. emerging topics) within short time period after their occurrence.

Given the fast rates at which data elements are produced and the big number of elements at each time point, this is a very challenging problem. For example, on Twitter, there are 500 million tweets published on an average day [2]. Thus, approximately 6,000 tweets are produced every second with an extreme case of more than 140,000 tweets published every second observed on the 3rd of August 2013 [3].

1.2 Contributions

In this work, we make a number of contributions in the area of event detection over large and dynamic data. These contributions are summarised below.

1. We define a topic to be a set of tags and introduce a novel correlation measure that estimates how strongly related are the tags comprising a topic. Our correlation measure captures, at the same time, the importance of a tagset in the whole dataset and in the subset of the dataset that concerns different aspects of the topic represented by the tagset.

2. We introduce a scoring function that considers the decay on the interest of the users in emerging topics and incorporate this function in the final

[2]https://about.twitter.com/company
[3]http://www.internetlivestats.com/twitter-statistics/

scoring function used to estimate how interesting a topic, spotted in the past, is in the present.

3. We propose four algorithms that partition the tags to multiple nodes and make each node responsible to compute the correlations for the tags in the subset of tags it has been assigned. The algorithms partition the tags exploiting the characteristics of the dataset and optimising for different criteria.

4. We discuss a theoretic view on the performance of the aforementioned partitioning algorithms which helps ascertain existing limitations and strengths.

5. We introduce a generic framework that identifies events in a relational database which is subject to continuous changes caused by insertions in the stored relations. Events in this setting are defined as changes in the top-k results produced by group-by aggregate queries.

6. We present two algorithms that maintain the top-k results. The algorithms limit the interaction with the underlying database combining ideas derived from multi-query optimisation and view maintenance. Both algorithms guarantee that the top-k results are always exact.

1.3 Publications

The work presented in this thesis has been published in various workshops and conferences.

In [AMRW12], we have presented the initial enBlogue approach which is described in Chapter 4. In this work, we introduced the correlation measure and the scoring function used to estimate the emergency of a topic represented by a set of tags. Initially, enBlogue was conceived to run in a single machine and the efficiency aspects of the problem were attacked via an attempt to minimise the number of tracked tagsets.

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

A demonstration of this work has been published in [ASRW11].

- Foteini Alvanaki, Michel Sebastian, Krithi Ramamritham, and Gerhard Weikum. Enblogue: Emergent topic detection in web 2.0 streams. International Conference on Management of Data (SIGMOD), 2011.

In [AM13a], we have introduced an approach for efficient computations of tag correlations over dynamic data. This work came as a consequence of [AMRW12] and the goal is to propose a method that will enable the identification of emergent topics in real time. The basic idea is to partition the tags to multiple nodes and let each node compute the tag correlations only for its assigned set of tags.

- Foteini Alvanaki and Sebastian Michel. Scalable, continuous tracking of tag co-occurrences between short sets using (almost) disjoint tag partitions (Best Student Paper). ACM SIGMOD Workshop on Databases and Social Networks (DBSocial), 2013

This approach was extended with additional partitioning algorithms and published in [AM14]. We present both the initial and the extended approach in Chapter 5.

- Foteini Alvanaki and Sebastian Michel. Tracking set correlations at large scale. International Conference on Management of Data (SIGMOD), 2014.

In [AM13b], we have presented our event detection approach on top-k rankings created in a relational database system using group-by aggregate queries. In this work, described in Chapter 6, we have presented two algorithms that minimise the interaction with the underlying database. The detailed experimental evaluation has demonstrated the gains achieved by our approach.

- Foteini Alvanaki and Sebastian Michel. A thin monitoring layer for top-k aggregation queries over a database. International Workshop on Ranking in Databases (DBRank), 2013 (co-located with VLDB 2013).

1.4 Thesis Outline

This thesis is organised as follows: Chapter 2 presents an introduction to basic concepts, measures and structures used throughout our work. Chapter 3 presents related work. Chapter 4 presents an algorithm on identifying emerging topics in Web 2.0 in general and social media in particular. It also introduces a measure that can be used to estimate the interestingness of each emerging topic. Chapter 5 presents algorithms enabling implementing our emerging topic detection approach and any approach based on set correlations in a large cluster allowing for efficient computation of correlations for multiple tagsets found in documents created in fast pace. Chapter 6 presents our approach of event detection in data warehouses. Finally, Chapter 7 presents our conclusions and discusses on future work.

Chapter 2

Background

In this chapter, we present some ideas, methods, algorithms and structures that are related to this thesis and are considered fundamental in the areas of Stream Processing, Information Retrieval and Top-k Query Processing. Readers having experience in these fields would be familiar with this material. The rest of the readers would get an overview of some basic ideas and methods that will allow them to better understand the work presented in the following chapters.

2.1 Social Media

In the Web 2.0 terminology, the social media are websites that do not simply provide information but allow users to interact with each other. The means of interaction can vary significantly with the most common of them being text. Apart or in addition to text, images (e.g. Flickr[1]) or videos (e.g. Youtube[2]) are also used. Users in social media are organised in communities sharing the same interests. Social media today are very popular and a lot of research is focusing on analysing the content published on them (e.g. [ACD$^+$08, PP10]) and how the relations and interactions of users are created and evolve through and across them (e.g. [GKL08, BHKL06]).

2.1.1 Weblogs

Weblogs, or simply blogs, are a category of social media. Each weblog consists of multiple entries which are usually ordered in reverse chronological order. Every entry in a blog is called *post*. Posts can be of an arbitrary long size and of any content. In general, each post is annotated with a set of tags, assigned by the author, that describe its content. In the early days of their existence, weblogs were mostly restricted in narrations of events from the author's every day life. This justifies their initial name "personal diaries".

In the majority of cases, a single user creates the blog and is the main author of it. Multi-author blogs, however, have emerged recently. A multi-author blog is owned by a single user but has multiple authors that contribute content to it. Other users, in addition to the authors, can contribute to a blog by commenting

[1]http://www.flickr.com/
[2]http://www.youtube.com/

on the posts. The posts, in most of the blogs, are textual with links to other blogs and web pages. The linking behaviour between weblogs, i.e. blog A links to another blog B and blog B links back to blog A, creates a social network. The whole community of blogs is called *Blogosphere*.

Some blogs have been very influential through time and many politicians and campaign leaders use them to impact the public opinion of the voters. As of early 2013, more than 200 million blogs existed[3]. This vast number of weblogs creates the need for search engines specifically designed to search information on the content of the blogs. The most popular of these engines is Technorati[4]. Technorati uses the tags the authors assign to their posts to categorise the search results.

2.1.2 Twitter

Twitter[5] is another form of weblog that appeared in 2006. The difference of it to traditional blogs is that the posts in Twitter are limited to 140 characters. The limited size of allowed posts places Twitter in a blogging subcategory, called microblogging.

Posts in Twitter are called tweets and are public unless otherwise specified by their author. Tweets, similarly to blog posts, are accompanied by a number of tags that indicate their topic. These tags in Twitter are preceded by the special character # and for that they are called *hashtags*. For example, `‘‘#Pique #Puyol & #Valdes are almost all out for the season & we can’t sign new players for 2yrs. Did we not pay #FIFA subscriptions? #Barca’’` is a tweet having five hashtags (`#Pique`, `#Puyol`, `#Valdes`, `#FIFA` and `#Barca`).

In order to post a tweet one should register on the Twitter webpage. Registered users create communities/networks by subscribing to other users tweets. The user subscribing to another user is said to follow this user and is called follower. The user to whom another user subscribes is called the followee. A user automatically sees in his, so called, *timeline*, all tweets posted by the users it follows.

Users can interact with each other by replying to tweets. A tweet intended to serve as a reply to some user contains the name of this user preceded by the character @. The same formulation is used when a tweet just mentions another user. For example, `‘‘@marilys #FIFA fine #Barca 305K + 14mth ban player transfer’’` is a tweet replying to the user called *marilys* while `‘‘Dear @FIFAcom stop pleasing all those European clubs and remove the transfer ban from @FCBarcelona. #Barca #FIFA’’` is a tweet mentioning the users *FIFAcom* and *FCBarcelona*. A reply can usually be differentiated from a mention by the existence of the `@user` in the beginning of the tweet. However, this is not obligatory and it is possible to have replying tweets with the `@user` part in the middle of the text and tweets beginning with `@user` that do not intend to serve as replies. Many users can engage in exchanging replies creating conversations in Twitter.

Sending again a tweet posted by another user is an additional form of interaction called *retweeting*. Retweets are exactly the same to the original tweets but they are additionally accompanied by the name of the user who made the

[3]http://snitchim.com/how-many-blogs-are-there/
[4]http://www.technorati.com/
[5]http://twitter.com/

retweet. Below is an example resembling a retweet:

```
‘‘Retweeted by marilys
#Barca banned from signing any player for the next 14 months’’
```

2.2 Data Stream Processing

Data stream processing is a data-centric processing model. Data arrives continuously, each arriving object is processed and, depending on the application, an action is performed for each or some of the received objects. Network monitoring (e.g. [CJSS03, IPF+07]), web applications (e.g. [APL98, GDH04]), and sensor networks (e.g. [HKG+06, PPKG03, MFHH02]) are some of the multiple data stream applications.

A data stream can be modelled as a sequence of objects

$$O = (o_1, o_2, o_3, \ldots)$$

An ordering of the objects is usually assumed. The ordering is imposed by a timestamp [BBD+02] that is assigned to each object reflecting its creation time (explicit timestamp) or the time of its arrival in the system (implicit timestamp). The objects can be of various types, from simple real values to vectors or more complex objects.

Data streams have two important characteristics: they are (i) continuous and (ii) unbounded. These two characteristics pose a number of challenges on the applications processing data streams:

- Since objects arrive continuously, the processing of each object should be quick in order to finish before the arrival of the next object.
- The objects to be received next in a data stream are not known. Data can only be processed in the order it is received forbidding random accesses.
- Since data streams are unbounded, collecting the whole data before processing them is not possible. Instead, a subset of the data is selected and processed at any time.

To reduce the memory needs of data stream processing applications, many data reduction techniques have been proposed. Sketches (e.g. [FM83]), data sampling (e.g. [CMN99]) and histograms (e.g. [IP99]) are among the most common of them. Data sampling is also used to cope with the fast arrival rates of the objects. An additional technique used for the same reason is data dropping (e.g. [TcZ+03]) in which objects are discarded by the data stream.

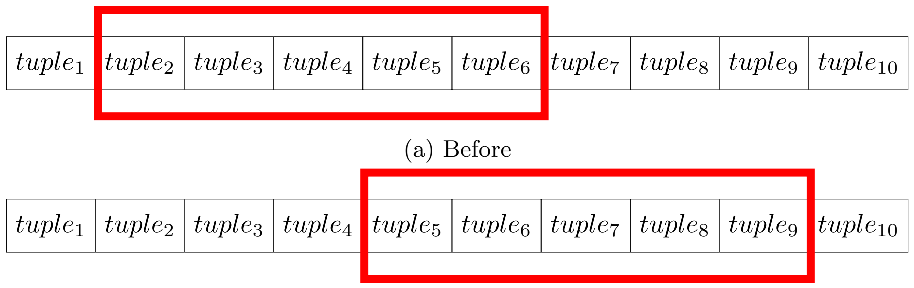
2.2.1 Sliding Window

Using a sliding window [CSA05] over an unbounded data stream is a common technique in data stream applications. A sliding window restricts the view on a subset of the data, the most recent of them, and allows performing operations only on this subset. Restricting the focus of the processing in the most recent data is a very natural method since old data are usually not interesting for users and real world applications.

Each sliding window has two characteristic values, its size $\mathcal{W}$ and the number of units w it slides every time. There are two basic types of sliding windows: *tuple-based* and *time-based* sliding windows.

Tuple-based Sliding Window

The size and the slide units of a *tuple-based* sliding window are defined with respect to the number of tuples (objects). A tuple-based sliding window of size $\mathcal{W}$ and slide units w contains at any point the $\mathcal{W}$ most recently created tuples and every time it slides it disregards the w oldest tuples and adds w new tuples. Figure 2.1 shows an example of a tuple-based sliding window with $\mathcal{W} = 5$ and $w = 3$ before (2.1a) and after (2.1b) the window slides.



(a) Before

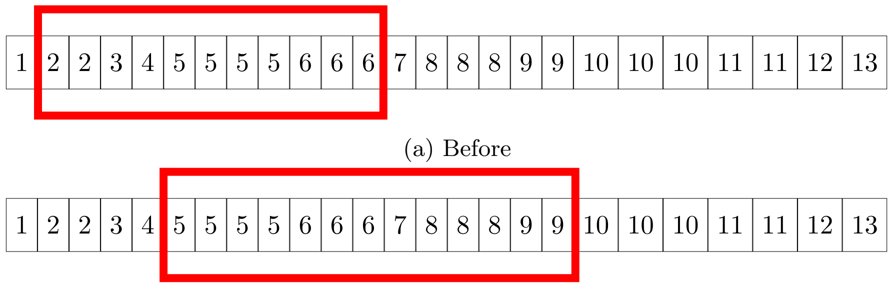
(b) After

Figure 2.1: Tuple-based Sliding Window

Time-based Sliding Window

A *time-based* sliding window is defined with respect to time. Each time-based sliding window has an upper, $time_{up}$, and a lower, $time_{low}$, time limit. At any point the window contains all objects having a timestamp within the time frame defined by the aforementioned limits. A time-based sliding window of size $\mathcal{W}$ and slide units w has an upper limit that differs from the lower limit by $\mathcal{W}$ time units, i.e. $time_{up} - time_{low} = \mathcal{W}$, and every time it slides the limits are increased by w time units, i.e $time'_{up} = time_{up} + w$ and $time'_{low} = time_{low} + w$. Figure 2.2 shows an example time-based sliding window with $\mathcal{W} = 5$ and $w = 3$ before (2.2a) and after (2.2b) the window slides.

Each number corresponds to the timestamp of the tuple/object.



(a) Before

(b) After

Figure 2.2: Time-based Sliding Window

As seen in Figure 2.2, the number of tuples inside a time-based sliding window can change after the slide of the window. Respectively, the time span of a tuple-based sliding window might be different after a slide.

2.2.2 Data Streams Mining

In data stream mining [GZK10] the goal is to use the data seen so far in order to extract information for the whole data stream and predict the characteristics of the data that will be received in the future. Common applications of data stream mining include clustering (e.g. [OMM$^+$02, AHWY03]), frequent pattern mining (e.g. [CH08, MM02]) and classification (e.g. [DH00, WFYH03]).

Data streams are ordered in time, thus the mapping from a data stream to a time series is straightforward. For this reason it is not uncommon to apply in data streams techniques that are used for the analysis of time series.

2.2.2.1 Time Series Analysis

Time series are analysed either on frequency domain or on time domain. With respect to frequency, the most common method is wavelet analysis. With respect to time, the most common method is correlation analysis.

Wavelet Analysis

Wavelet Transform [BN09] is a method for analysing the frequency components of a stream. Wavelet transform provides information regarding the frequencies comprising a stream and the time points when these frequencies exist. More formally, wavelet transform rewrites the stream in terms of an orthonormal function basis providing time-frequency representation. The simplest such basis in Harr wavelet [BN09].

Fourier [Bri88] and wavelet transform are related to each other. The main difference is that wavelets provide information in time and frequency at the same time whereas Fourier transform provides information only for the frequency but not about the time a specific frequency occurred.

Correlation Analysis

Correlation analysis [Bri88] is used to determine the statistical dependency between two time series. Two series are considered positively correlated when they show similar trends in time, i.e. both increase of decrease. Similarly, two time series are negatively correlated when they show opposite trends in time, i.e. one increases when the other decreases and vice versa. The correlation of a time series with itself at a different time point, called auto-correlation, can be used to determine repeating patterns in the stream.

2.2.3 Distributed Data Stream Processing

In a 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.

Limiting the communication needs in these applications is very important. Common techniques include reducing the number of messages exchanged among the nodes by omitting all but the absolutely necessary of them and/or reduce the size of data sent with these messages. Minimising the communication however, can potentially affect the accuracy of the results in case the nodes do not manage to obtain all the information they need, thus care should be taken so that the quality of the produced results remains within acceptable limits.

2.2.3.1 Storm

Storm[6] is a framework that provides a fault-tolerant, distributed stream processing infrastructure. From an abstract perspective, it resembles Hadoop MapReduce[7] with the difference being that MapReduce is designed for batch processing while Storm continuously processes incoming data.

Topology

What in MapReduce is called "job" is modelled in Storm as a topology. The topology is essentially a graph with nodes representing operators that are connected with each other depending on how they communicate. The communication in Storm is performed following a push-based model. Figure 2.3 shows an example topology. Storm distinguishes two types of operators, *Spouts* and *Bolts*. Spouts are sources of streams. Bolts consume any number of streams, process them and, possibly, emit new streams.

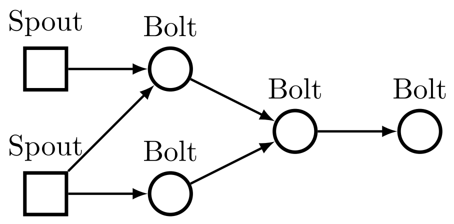


Figure 2.3: An example topology

Data Flow Specification

The operators in Storm exchange tuples, i.e. simple lists of values. Bolts consume data from Spouts or other Bolts by registering to their output streams. One of the key properties of Storm is that it allows multiple instances (called tasks in Storm) 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.

[6]http://storm-project.net/
[7]http://hadoop.apache.org/

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 a specific instance (or instances) by using the instance's unique identifier.

Parallelism

The application developer defining the topology provides at the same time *hints* on how the topology should be parallelised. Storm provides three levels of parallelism:

- **worker:** Each worker executes a subset of the topology in its own Java Virtual Machine. The developer specifies the number of workers that wants for his topology. Storm assigns to the topology the maximum possible number of workers asked. Each worker can potentially run on a different machine but this is not necessary. The number of workers that can run in each machine is part of the Storm configuration. Each worker has an incoming and an outgoing queue where messages to or from the subset of the topology that has been assigned to the specific worker are stored.
- **executor:** Each executor is a separate thread. For each operator, there is one executor defined by default. The developer can ask for multiple executors for each operator that will process data in parallel. Each executor is assigned to one of the topology's workers. Multiple executors can be assigned to the same worker. Similar to the workers, each executor has an incoming and an outgoing queue. Incoming messages are transferred from/to the worker queue to/from the appropriate executor queue.
- **task:** The task performs the actual data processing. By default, each executor is assigned one task. If more tasks than executors are defined each executor runs multiple tasks sequentially.

Figure 2.4 depicts an example assignment of executors to workers and of tasks to executors. For this example, we assume two operators, the red and the green. The developer has defined two tasks of the green operator and three tasks of the red operator. Additionally, the developer has defined two workers and four executors. The code used to define this parallelism is the following:

```
1. topology.setBolt("green", new Green(), 1).setNumTasks(2)
2. topology.setBolt("red", new Red(), 3)
3. configuration.setNumWorkers(2);
```

The developer wants multiple (2) tasks of the green operator to be assigned in one executor. For that it uses the `setNumTasks` option (cf. command 1). For the red operator, the developer wants one task per executor which is the default assignment performed in Storm if nothing else is provided (cf. command 2). The developer has no control on which and how many executors will be assigned in each worker. She can only define the number of workers that she desires (cf. command 3). Storm decides how many of the requested workers is possible to be created and how the executors are assigned on them.

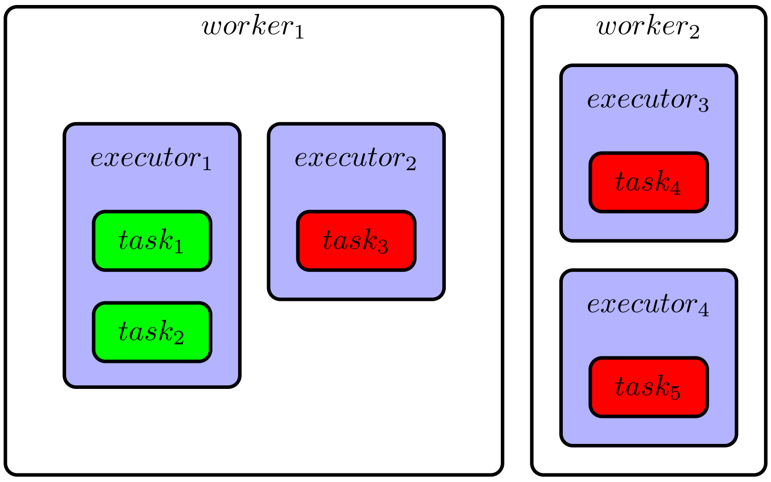


Figure 2.4: Parallelism in Storm

In the used version of Storm (v0.8.2), the number of tasks defined in a topology are static and cannot be changed while the topology is running. Executors and workers on the other hand can be reconfigured online, *rebalancing* the topology. However, redefining the number of executors is possible only when executors are less than the defined tasks. All tasks assigned to the same executor run sequentially so, in real applications that intend to run multiple tasks in parallel creating more tasks than executors does not usually make sense. However, this can be useful while testing a topology.

2.3 Information Retrieval

According to [MRS09], Information Retrieval is finding the material that satisfies an information need within large collections of information resources. The data used in Information Retrieval are usually unstructured text files or documents of limited structure, having, for example, author, subject and so on. These documents are not easily understood by a computer and as a consequence Information Retrieval techniques often return estimated or inexact results. The most common applications using Information Retrieval techniques are Web search engines. In these applications, the information need is described with a query posted by the user and the material satisfying the information need is, in its majority, a set of web pages.

2.3.1 Inverted Index

The inverted index [ZM06] is among the most basic structures used in Information Retrieval systems. It maps a term (or word) to the list of documents that contain it allowing immediate retrieval of all the documents that contain a specific term. The lists of documents associated with each term are called postings lists. The documents in these lists are usually represented by their document identifiers. The inverted index is the most efficient structure for ad-hoc queries over collections of documents [MRS09]. Depending on the selected retrieval model a postings list may contain additional information such as the number of times a term is found in each document and/or be sorted according to some score, e.g. a weight reflecting the importance of the term in each document.

Figure 2.5 shows an example of a simple inverted index created using the following documents:

- $document_1$ = "the house on the beach"
- $document_2$ = "a green big house"
- $document_3$ = "the green garden of the house"

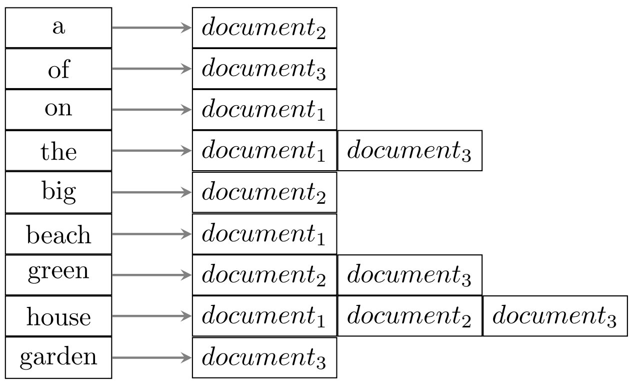

a → $document_2$
of → $document_3$
on → $document_1$
the → $document_1$ $document_3$
big → $document_2$
beach → $document_1$
green → $document_2$ $document_3$
house → $document_1$ $document_2$ $document_3$
garden → $document_3$

Figure 2.5: Inverted index example

2.3.2 Results Quality Estimation Methods

Since the semantic relevance of a query to a document is not straightforward, it is very important to be able to asses the quality of the results returned from an Information Retrieval system. *Precision* and *recall*, are two measures used for this purpose [MRS09].

Recall is a measure reflecting the fraction of the total relevant documents retrieved by the system. In order to find the total relevant documents, great manual effort is required: all documents should be examined and declared as relevant or not to the information need. For this reason, it is not always possible to know in advance the total number of relevant documents. This prohibits in many cases the use of the recall measure.

Precision is a measure reflecting the fraction of retrieved documents that are relevant to the information need. Relevant documents, as before, are judged by humans, but, in this case, the set of documents to be examined is restricted to the set of documents retrieved by the Information Retrieval system. Judging the relevance of a small set of retrieved documents, instead of the whole collection, is a relatively easy task and precision is the measure used in most cases to asses the effectiveness of an Information Retrieval system.

There are many measures to compute precision. The simplest of them is derived by the definition of the measure and is computed using the following formula:

$$precision = \frac{\#\{relevantDocuments \cap retrievedDocuments\}}{\#retrievedDocuments}$$

However, the results retrieved by the Information Retrieval systems are usually ranked according to how related they are estimated to be to the information need. To asses the quality of the results in these cases more sophisticated precision measures are used. Below, we describe two such measures.

Precision@k

Precision@k focuses on the top-k results retrieved by an information retrieval system and finds within these results the number of relevant ones. The measure is computed as the fraction of relevant results in the top-k results, using the following formula:

$$Precision@k = \frac{\#relevenatResultsInTopk}{k}$$

For example, Precision@10=0.3 means that in the top-10 retrieved results there are 3 that are found to be relevant. Precision@k is considered a binary measure, since it uses only the number of relevant results and the ranking positions of these results are not taken under consideration.

NDCG

NDCG is the normalised version of DCG [JK02]. DCG (Discounted Cumulative Gain) estimates the satisfaction of the user who submitted the query obtained from the retrieved results. This satisfaction is called gain. DCG requests high relevant results to be returned higher in the ranking based on the following two principles:

- Users are less likely to examine results low in the ranking.
- Users get more satisfaction from highly relevant results.

Using any measure of relevance, each retrieved result is assigned a score $relevance_i$, where i depicts the position of the result in the ranking. The lower in the ranking each result is placed, the more its score is discounted. The total gain from a ranking of k results is computed as:

$$DCG@k = relevance_1 + \sum_{i=2}^{k} \frac{relevance_i}{\log(i)}$$

To be able to compare various lists of ranked results, the normalised version (NDCG) of DCG is used. NDCG is computed as the ratio of the DCG computed over the obtained ranking to the DCG computed over the ideal ranking. Ideal is the ranking having all results ordered according to the user perceived satisfaction.

2.3.3 Vector Space Model

The vector space model [MRS09] is a common model used to represent text documents and queries as vectors. Assuming a collection $\mathcal{D}$ of documents, a dictionary $\mathcal{T}$ is created that contains all terms t_i that are found in these documents. Each document in the collection $\mathcal{D}$ is then represented as a vector of size equal to the size of the dictionary $\mathcal{T}$. Depending on the Information Retrieval method, the vector representation of each document can be either *binary* or *real*.

Using a binary vector representation implies that all terms are of equal importance and the interesting information is whether a term is present in a document or not. Having 0 (zero) in a position of the binary vector indicates the absence of the corresponding term from the document while having 1 (one) indicates the presence of the corresponding term in the document.

Using a real vector representation allows handling each term differently. Terms that are absent from the document are again represented with a 0 (zero) value, but terms that are present in the document are represented with a real value which reflects the importance of the corresponding term in the document. The most common measure that is used to reflect the importance of a term in a document is the $tf \times idf$ measure.

2.3.4 $tf \times idf$ measure

Assuming a collection of documents $\mathcal{D}$, $tf \times idf$ measure [MRS09] is used to assign a weigh w_i^j to each term t_i in each document d_j in $\mathcal{D}$. The weight w_i^j reflects the ability of t_i to differentiate d_j from the rest of the documents in the collection. The idea behind the $tf \times idf$ measure is that a term t_i can better differentiate a document d_j when:

- the frequency of t_i within the document d_j, $tf(t_i, d_j)$, is high
- the frequency of t_i in the whole collection D, $df(t_i, \mathcal{D})$, is low

When the above conditions are met the term t_i in the document d_j should be assigned a high weight. The weight is computed using the following formula:

$$w_i^j = tf(t_i, d_j) \times idf(t_i, \mathcal{D})$$

where $idf(t_i, \mathcal{D})$ is the inverse of $df(t_i, \mathcal{D})$.

2.4 Similarity Measures

Similarity measures are used to quantify the similarity between objects. Depending on whether the objects in question are represented using a binary or a real vector, different similarity measures should be used ([LRB09]).

When the objects are represented as real vectors, numerical similarity measures are used. It is common to define a numerical similarity measure as the inverse of a distance function. One of the most frequently used distance functions is the Euclidean distance which computes the distance between two objects $\mathbf{X} = (x_1, x_2, \ldots, x_n)$ and $\mathbf{Y} = (y_1, y_2, \ldots, y_n)$ using the following formula:

$$d(\mathbf{X}, \mathbf{Y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

The distance is usually normalised using the min-max normalisation computed from the following formula:

$$d' = \frac{d - d_{min}}{d_{max} - d_{min}}$$

In the above formula d' is the normalised distance, d is the computed distance, d_{min} and d_{max} are the minimum and maximum computed distances in the whole dataset.

When the objects are represented as binary vectors, the fundamental question is whether the objects share elements or not. For that, the measures estimating the similarity between any two binary objects $\mathbf{X}$ and $\mathbf{Y}$ are expressed as functions of the following quantities:

- $|\mathbf{X} \cap \mathbf{Y}|$: the number of attributes present in both $\mathbf{X}$ and $\mathbf{Y}$
- $|\mathbf{X} - \mathbf{Y}|$: the number of attributes present in $\mathbf{X}$ but not in $\mathbf{Y}$
- $|\mathbf{Y} - \mathbf{X}|$: the number of attributes present in $\mathbf{Y}$ but not in $\mathbf{X}$
- $|\overline{\mathbf{X}} \cap \overline{\mathbf{Y}}|$: the number of attributes absent from both $\mathbf{X}$ and $\mathbf{Y}$

Binary based similarity measures are further divided based on whether they consider the attributes absent from all objects or not. The similarity measures that only care about the attributes that are present in some of the objects are the similarity measures of type 1. The similarity measures that consider both the attributes that are present and those that are absent are the similarity measures of type 2.

Nominal data are usually represented using binary vectors.

2.4.1 Jaccard Coefficient

Jaccard coefficient [Jac12] is one the most frequently used similarity measures designed for binary data. It increases as the number of attributes present in both objects increases and decreases as the number of attributes present in either of the objects but not in the other decreases. Jaccard coefficient does not consider the number of elements absent from both objects, thus it is a binary similarity measure of type 1.

Using the quantities described above, the Jaccard coefficient between two objects $\mathbf{X}$ and $\mathbf{Y}$ is defined as:

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

However, Jaccard coefficient is more commonly computed using the following formula:

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

Jaccard is a symmetric similarity measure, i.e. $J(\mathbf{X}, \mathbf{Y}) = J(\mathbf{Y}, \mathbf{X})$. It can be computed for an arbitrary number of objects using the general formula:

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

2.4.2 Inclusion–Exclusion Principle

The inclusion–exclusion principle [Dek86] allows computing the total number of attributes present in any of two objects $\mathbf{X}$, $\mathbf{Y}$ using the number of attributes present in either of them and the number of attributes present in both of them. The inclusion–exclusion formula for the objects $\mathbf{X}$ and $\mathbf{Y}$ is the following:

$$|\mathbf{X} \cup \mathbf{Y}| = |\mathbf{X}| + |\mathbf{Y}| - |\mathbf{X} \cap \mathbf{Y}|$$

The inclusion–exclusion principle is generalised to multiple objects $\mathbf{X}_1$, $\mathbf{X}_2$, …, $\mathbf{X}_n$ by the following formula:

$$\left|\bigcup_{i=1}^{n} \mathbf{X}_i\right| = \sum_{k=1}^{n}(-1)^{k+1}\left(\sum_{1\leq i_1<\cdots<i_k\leq n} |\mathbf{X}_{i_1} \cap \cdots \cap \mathbf{X}_{i_k}|\right)$$

For example, for three objects $\mathbf{X}_1$, $\mathbf{X}_2$, $\mathbf{X}_3$ the formula is:

$$\begin{aligned}|\mathbf{X}_1 \cup \mathbf{X}_2 \cup \mathbf{X}_3| = &|\mathbf{X}_1| + |\mathbf{X}_2| + |\mathbf{X}_3| \\ &- |\mathbf{X}_1 \cap \mathbf{X}_2| - |\mathbf{X}_2 \cap \mathbf{X}_3| - |\mathbf{X}_2 \cap \mathbf{X}_3| \\ &+ |\mathbf{X}_1 \cap \mathbf{X}_2 \cap \mathbf{X}_3|\end{aligned}$$

2.5 Metrics of inequality

The metrics of inequality are usually used by social scientists to measure the inequality of the wealth distribution in the population. These methods, however, have general applicability and can be used to estimate the inequality in the values of any distribution.

Lorenz Curve

Lorenz curve [BL08] is a graphical method to represent the inequality in a distribution. Having a population of size n and a value x_i for each individual in this population, Lorenz curve is created by ordering the values x_i in increasing order so that x_1 corresponds to the smallest value and x_n to the biggest, and plotting the points:

$$\left(\frac{j}{n}, \frac{\sum_{i=1}^{j} x_i}{\sum_{i=1}^{n} x_i}\right), j = 1, 2, \ldots, n$$

The above corresponds to the cumulative distribution of the empirical probability distribution.

Considering a set of individuals with the income shown in Table 2.1, the corresponding Lorenz curve is shown in Figure 2.6.

Gini Coeffcient

The Gini Coefficient [BL08], also known as Gini index, is another measure used to estimate the inequality of the values in a distribution. It was initially introduced as a measure of the income inequality in the population, but its usage

Individual	Income	Individual	Income
1	1342	6	9120
2	2654	7	11547
3	3945	8	14321
4	5126	9	17948
5	6947	10	22168

Table 2.1: Income example

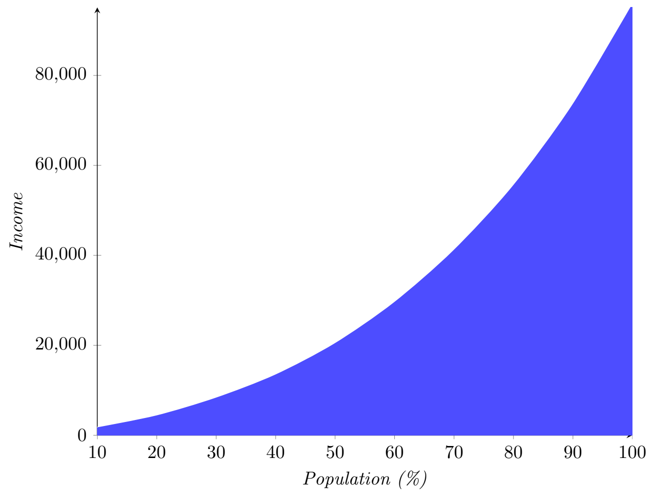


Figure 2.6: Lorenz curve example

has become much wider. Mathematically, the Gini coeffcient is defined as the ratio of the area between the line of equality (a line with angle 45 degrees) and the Lorenz curve (grey area in Figure 2.7). A small value of the Gini coefficient indicates a more balanced distribution with 0 (zero) reflecting perfect equality and 1 (one) perfect inequality.

Given a population of size n and a value x_i for each individual in this population, ordered in increasing order so that x_1 corresponds to the smallest value and x_n to the biggest one, the Gini coefficient can be computed using the formula:

$$G = \frac{1}{n} \times \left(n + 1 - 2 \times \left(\frac{\sum_{i=1}^{n} (n+1-i) \times x_i}{\sum_{i=1}^{n} x_i} \right) \right)$$

2.6 Top-k Queries

Top-k queries have applications in various domains such as similarity search over multimedia data (e.g. [CGM04, GBK00]), spatial data analysis (e.g. [RJVDN10, RJN12]), relational databases (e.g. [CG99]) and data stream processing settings (e.g. [MBP06]). The general idea is that each result satisfying the query is assigned a score using some scoring function. The k results with the best scores are selected and presented to the user. Selecting the top-k out of a possibly huge

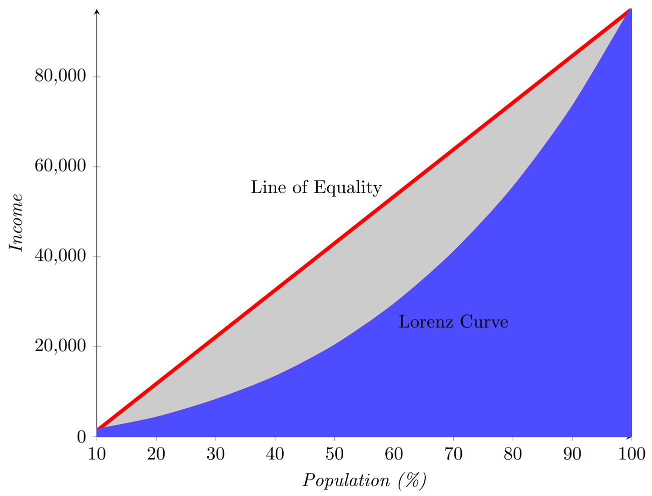


Figure 2.7: Gini coefficient example

amount of results is a very natural procedure since users are often interested in the results that show an outstanding performance. Therefore, top-k query processing has attracted a lot of research interest during the previous years.

2.6.1 Taxonomy

Top-k query processing techniques can be classified according to various criteria [IBS08], creating a taxonomy.

Query Model

The query model specifies the data objects which are considered by the top-k query processing technique to have a score. The top-k results are selected out of these data objects. According to the query model, the techniques are classified to:

- **Top-k Selection Queries:** In this model, the data objects that are assigned a score are the base tuples. The scores from multiple attributes of the same tuple might need to be combined in order to produce the final score of the tuple.

- **Top-k Join Queries:** In this model, the data objects that are considered to have a score are results of a join. Each base tuple has a score and the score of a joined result is a combination of the scores of the base tuples that produced the joined result. The top-k results are selected from the produced tuples.

- **Top-k Aggregate Queries:** In this model, the data objects that are assigned a score are groups of tuples. The base tuples are grouped together and each group is assigned a score which is an aggregation of the scores of the base tuples it contains. The group scores are then used to select the top-k groups.

Data Access Methods

The categorisation of the top-k processing techniques according to the Data Access Method depends on the extend they allow random accesses on the data. The three possible categories are (i) random accesses are allowed, (ii) controlled random access are allowed, and (iii) random accesses are not allowed.

Implementation Level

A top-k processing algorithm can be either implemented in (i) the application level or (ii) the query engine level. The techniques implemented in the application level might use specialised indices or materialised views but the main top-k processing is performed outside the query engine. The techniques implemented inside the query engine may introduce new query operators (e.g. [IAE04]).

Data and Query Uncertainty

Various categories can be created combining the data and query uncertainty levels:

- **Exact methods over certain data:** Both the method and the data are deterministic
- **Approximate methods over certain data:** The data are deterministic but the methods report approximate results in an attempt to minimise the response time.
- **Uncertain data:** The data are probabilistic and the methods are based on uncertainty models.

Ranking Function

Most of the top-k processing techniques use a **monotone ranking function** which has properties that allow efficient top-k retrieval. Few approaches (e.g. [ZHC$^+$06]) use a **generic ranking function** which they try to optimise in the presence of constraints. Finally, the skyline queries rank objects according to various attributes and select those objects that are not dominated by any other object. These queries belong to the category of queries with **no ranking function**.

2.6.2 The Threshold Algorithm (TA)

Among the most influential algorithms on top-k query processing is the Threshold Algorithm (TA) proposed by Fagin [FLN01]. TA assumes the existence of m sorted lists L_i and determines the top-k objects that have the maximum overall score in all lists executing the following steps:

1. It accesses the objects in the m lists in parallel, starting from the top objects (sorted access).

2. For any object x that is seen in one of the lists during the sorted access, it performs a random access to the rest of the lists and retrieves the corresponding scores for x.

3. It computes the final score for the object x. If this score is among the top-k scores seen so far then it remembers x.

4. Assuming l_i to be the smallest score seen in list L_i during sorted access, it computes a threshold score aggregating the scores l_i of all lists L_i. This score is the maximum possible score any unseen object can have.

5. If at least k objects with scores greater or equal to the threshold have been seen then the algorithm terminates.

As long as the aggregate function is monotone, the TA algorithm correctly identifies the top-k objects.

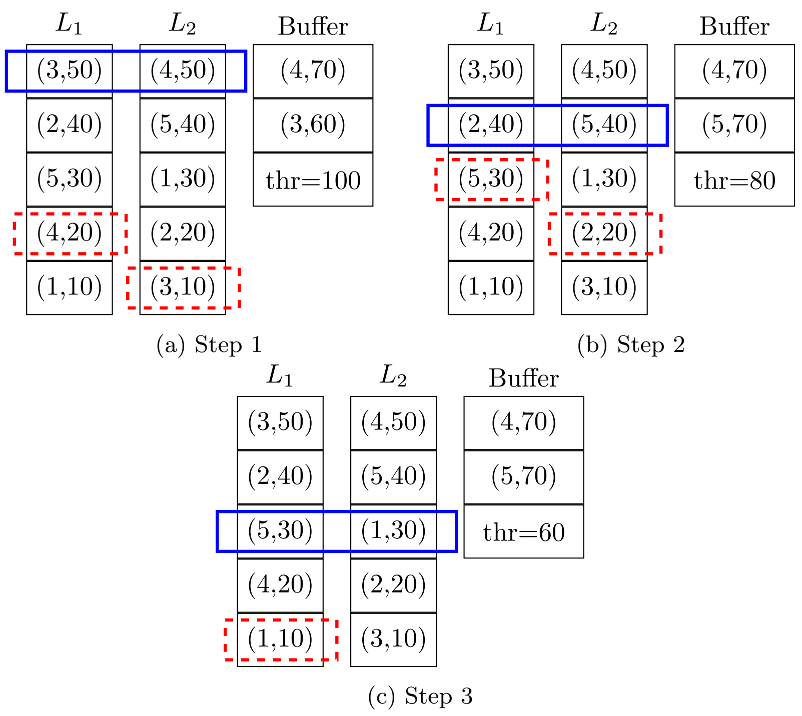


Figure 2.8: TA algorithm example

Figure 2.8 shows an example of the TA algorithm where the top-2 objects are retrieved. In the example, there are two lists, L_1 and L_2. Each list stores pairs of the form *(object_id, score)*. The final score of each object x is computed as the sum of its scores in the two lists. The blue continuous lines depict the sorted accesses while the red dashed ones the random accesses. The Buffer contains at any time the top-k objects seen so far. The algorithm terminates after three steps:

Step 1 (Figure 2.8a): The sorted access on the lists retrieves the elements $(3, 50)$ and $(4, 50)$. Summing these two scores gives the threshold value, equal to 100. A random access is performed for each object. For the object with id 3, the score obtained through random access is 10, resulting in its final score being 60. For the object with id 4, the additional retrieved score is 20 and its final score is computed to be 70. Both objects have a score that includes them in the top-2 results, thus they are stored in the Buffer. The threshold is greater than the score of the two objects and the algorithm proceeds.

Step 2 (Figure 2.8b): The sorted access in the second step retrieves the elements $(2, 40)$ and $(5, 40)$. The sum of these two scores produces the new threshold value, equal to 80. The random accesses for the two objects retrieve elements $(5, 30)$ and $(2, 20)$, resulting in the final score for the object with id 5 being 70 and the final score for the object with id 2 being 60. The final score of the object with id 5 is greater than the score of the object with id 3 that was put in the Buffer in the previous step, thus the element (5,70) replaces the element (3,60) in the Buffer. Since the threshold is still greater than the scores of the two objects currently in the Buffer, the algorithm proceeds to the next step.

Step 3 (Figure 2.8c): The sorted access of the third step retrieves only one element, that with id 1 and score 30. The other element, with id 5, has been already retrieved through a random access in the previous step. The random access for the object with id 1 retrieves a score of 10 for it, resulting in a final score of 40. The threshold in this step is calculated to be 60 and since both objects in the Buffer have a score greater than 60 the algorithm terminates. The top-2 objects found are the object with id 4 and score 70 and the object with id 5 and score 70 as well.

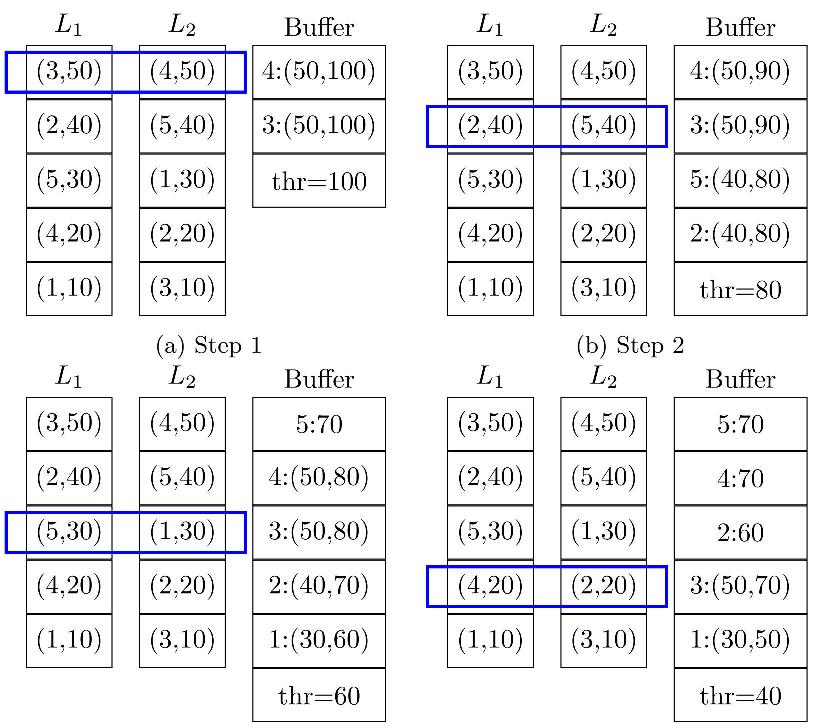
According to the taxonomy presented in Section 2.6.1, the query model assumed in TA is the top-k selection query since the elements in the various lists are considered to be attributes of a tuple. TA allows sorted and random accesses and is an exact method performed over certain data. Is uses a monotone ranking function and the implementation of it is in the application level.

2.6.3 Non-Random Access Algorithm (NRA)

A limitation of TA is that it assumes the ability to perform random accesses on the lists. However, random accesses may not be available or be very expensive. Non-Random Access (NRA) Algorithm modifies the TA algorithm to use only sorted accesses. In NRA, each object x has two scores, a lower bound score computed using the scores seen for this object in the lists L_i and an upper bound score computed using the scores seen for x in the lists L_i plus the scores l_i for all lists L_i x has not been seen. The upper bound score is updated every time a new object, retrieved from a list, causes l_i to drop. The top-k objects are selected using the lower bound scores. Objects with an the upper bound score that does not qualify for the top-k scores are discarded.

Figure 2.9 shows the same two lists show in example 2.8. NRA is now used, instead of TA, to find the top-2 objects in these lists. As seen, NRA requires one more step than TA before being able to terminate:

Step 1 (Figure 2.9a): In the first step, the element $(3, 50)$ is retrieved from list L_1 and the element $(4, 50)$ is retrieved from list L_2. Summing the



(a) Step 1 (b) Step 2

(c) Step 3 (d) Step 4

Figure 2.9: NRA algorithm example

scores of the two retrieved results in a threshold equal to 100. For the object with id 3, the lower bound is its retrieved score, i.e. 50, and its upper bound is its retrieved score increased by the minimum score in the other list L_2, i.e. 50+50=100. Similarly, the lower and upper bound scores of the object with id 4 are 50 and 100 respectively. Both objects are stored in the Buffer since their upper bound scores are not less than the threshold.

Step 2 (Figure 2.9b): In the second step, the elements $(2, 40)$ and $(5, 40)$ are retrieved. Retrieving these two new elements drops the minimum score seen in each list to $l_1 = 40$ and $l_2 = 40$. Using the new minimum scores, the threshold is computed to 80 and the upper bounds of the scores for the objects with id 3 and 4, retrieved during the previous sorted access, are updated to 90. The bounds for the new objects are computed using the same methodology used in the previous step. The result is a lower score equal to 40 and an upper score equal to 80 for both objects. All four retrieved objects have scores not less than the threshold, thus they are all stored in the Buffer.

Step 3 (Figure 2.9c): In this step, the elements $(5, 30)$ and $(1, 30)$ are retrieved. The object with id 5 is already in the Buffer since its score in list L_2 has been retrieved during a sorted access in a previous step. Retrieving the score of the object from L_1 results in computing the real score for it to

70 (its minimum score plus the new score). All other objects in the Buffer have their lower bounds updated. The bounds of the new object with id 1 are computed to be 30 and 60. The threshold is computed now to be 60. Since neither of the old nor the new objects has an upper bound less than the threshold, all objects are stored in the Buffer. Although there is an object (the object with id 5) with known real score, it cannot be output since there are other objects that have potentially a score greater than it (the objects with id 3 or 4).

Step 4 (Figure 2.9d): In the final step, the elements $(4, 20)$ and $(2, 20)$ are retrieved. Both of them are already in the Buffer and their real scores are computed to be 70 for the object with id 4 and 60 for the object with id 2. The upper bounds of the other objects in the Buffer are updated and the threshold is computed to 40. Although there are still objects in the Buffer with upper bounds not less than the threshold, two objects, 5 and 4, have real scores that cannot be exceeded by any other object. The algorithm outputs these two objects and terminates.

NRA differs from TA in the data access method. So, according to the taxonomy presented in Section 2.6.1, NRA uses the top-k selection queries model and does not allow random accesses. It is an exact method performed over certain data using a monotone ranking function and the implementation of it is in the application level.

2.7 Data Warehouses

A data warehouse is a database system that stores recent and past data collected, usually, from various sources. It forms a central repository storing precomputed summaries of the data and it is used for Online Analytical Processing (OLAP) that provides support for decision making.

2.7.1 Data Organisation

Data in a data warehouse is summarised over multiple attributes. Each such attribute is called dimension. For example, a retails chain interested in the sales for each product in each city every year will organise the data over the attributes product, place and date. These attributes are the dimensions over which the sales are summarised. Each combination of product, place and date instances along with the corresponding sales value is called a fact.

Conceptually, the organisation of data into dimensions and facts resembles a data cube (or hypercube in case of more than three dimensions). Each dimension in the cube holds the instances for one of the attributes over which a measure of interest is summarised, i.e. the attributes product, place and date of the previous example. Each cell in a data cube stores the measure of interest, i.e. the sales of the previous example. A data cube example can be seen in Figure 2.10[8]

It is common to organise dimensions in hierarchies when generalisations and specialisations of them exist. For instance, the place dimension of the above example could be generalised into countries, countries into continents and so

[8]Image taken from `http://docs.oracle.com/html/B13915_04/i_olap_chapter.htm`

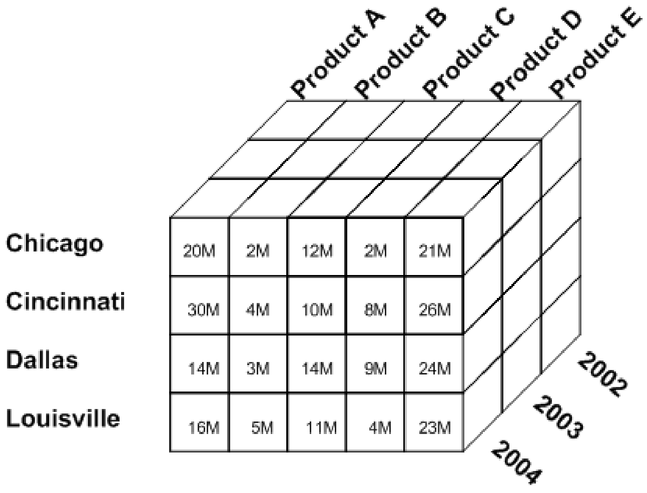


Figure 2.10: Data Cube

on. For each generalisation, a summary of the measure of interest (e.g. sales) is also computed.

Dimensions and facts are usually physically stored in different tables. The facts table references the dimension tables using a primary-foreign key relation. This physical organisation of data aligned horizontally resembles a star schema. Figure 2.11[9] shows a star schema example. When dimensions are organised in hierarchies the schema resembles more that of a snowflake as can be seen in Figure 2.12[10].

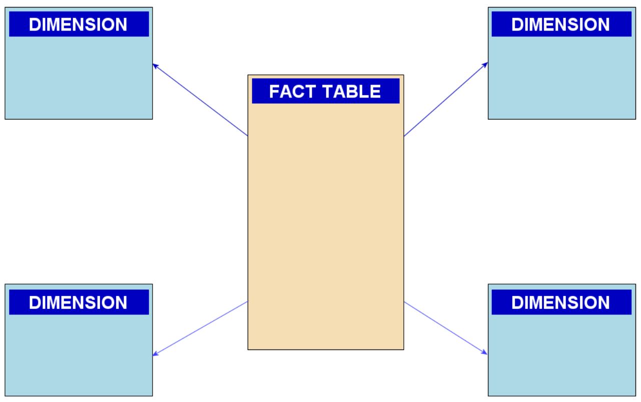


Figure 2.11: Star schema

[9]Image taken from `http://commons.wikimedia.org/wiki/File:Star-schema.png`
[10]Image taken from `http://en.wikipedia.org/wiki/File:Snowflake-schema.png`

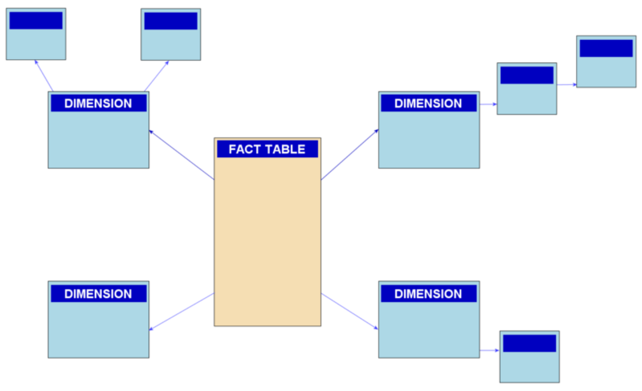


Figure 2.12: Snowflake schema

2.7.2 Extract, Transform and Load Data

Data in data warehouses is relatively static. The, so called, operational databases receive every new coming data and from time to time this data is transferred to the data warehouse. This procedure consists of three phases: Extraction, Transformation and Loading.
Extraction: During the Extraction phase, the data from the various source systems are extracted. During this phase, data are checked to verify that they meet an expected pattern. This may result in rejecting part or all of them.
Transformation: During the Transformation phase, the data are transformed in a way that complies with the organisation schema and the data types used in the data warehouse, e.g. dates are transformed from “yyyy/mm/dd” to “dd MMMM yyyy” and so on. It is during this phase that the summaries of the measures of interest are computed.
Loading: During the Loading phase, the data are actually loaded into the data warehouse.

Extract, Transform and Load (ETL) procedure may be very challenging if the data warehouse schema has not been carefully created.

Chapter 3

Related Work

The data we consider in this thesis is either sequences of messages published in social media or sequences of insertion queries applied in a relational database. In both cases, we deal with stream data and try to identify in them interesting characteristics. From that perspective, this thesis is related to works on *data stream mining* and we present a number of works that cover some approaches in this area (Section 3.1). Mining data streams created from messages published in social media is closely related to works on *topics identification* (Section 3.2). More specifically, we focus on identifying *emerging topics* discussed in social media as close to their appearance time as possible (Subsection 3.2.1). The ideas presented in this thesis on identifying events in relational databases are based on maintaining top-k rankings created using top-k aggregate queries. From that perspective, the thesis is also related to works on *top-k queries* (Section 3.3). The top-k queries we consider are continuous and their results are materialised and maintained in the presence of updates that potentially affect them. This idea is related to works on *materialised views maintenance* (Section 3.4). In the following, we present a set of representative works on all the aforementioned areas.

3.1 Data Stream Mining

A common goal in data stream mining is to find streams that change through time following similar patterns. Along this line is the work of Davood Rafiei [Raf99]. Rafiei suggests using linear Fourier Transformations to express similarity functions over data streams, e.g. moving average, shift, etc. [Raf99] proposes processing multiple such transformations at a time in order to find similarities between the data streams. The idea, in brief, is to create an R-tree [Gut84] index over the first n Fourier coefficients of each sequence and apply a group of transformations in them. The transformed sequences are checked as whether they follow the same pattern.

Similar to [Raf99], Yunyue Zhu et al. [ZS02] in their approach, named StatStream, use also the first n Fourier coefficients to approximate a data stream. More specifically, they use a time window over the stream and transform the data in the window in the frequency domain using the Discrete Fourier Transform. Afterwards, they approximate the data keeping only the first n Fourier coefficients. It is known in general that the first Discrete Fourier Transform coefficients of a time series contain most of the information [BN09]. So, approximating a stream with the first n Fourier coefficients is a common practise (popular also in data compression) that is expected to capture the structure of the data. Using this technique, Statstream is able to statistically monitor in real time up to 10,000 streams, reducing the time and space needed to process them.

Byoung-Kee Yi et al. [YSJ$^+$00] monitor multiple sequences of data in parallel. At any point in time, they can express a monitored sequences s_i as a linear combination of the old values of this sequence s_i and the new and old values of the other sequences. Using multi-variate linear regression [Was10], they can predict the current value of the sequence s_i. This allows filling values for sequences that were not received during the current time point due to for, example, some delay. They use the same technique to find unexpected values for sequences, i.e. received values that are very different from the predicted ones, which might indicate an interesting event related to the corresponding sequences. Finally, the coefficients used in the linear regression formula are used to find dependencies between sequences; a high coefficient indicates a high correlation between the predicted sequence and the sequence having the coefficient.

Richard M. Karp et al. [KSP03] propose an algorithm for finding the elements in a data stream with frequency higher than some threshold θ. The algorithm they propose is a generalisation of the majority algorithm [BM91] which finds an element that exists with frequency 50% in the data. To achieve that, it stores a single element accompanied by a counter. The counter is increased by 1 (one) every time the element is seen in the input and decreased by 1 (one) otherwise. When the counter is 0 (zero), the stored element is replaced by the element seen in the input and the counter is set to 1 (one). To find the elements with frequency higher than θ [KSP03], store $k = \lfloor \frac{1}{\theta} \rfloor$ elements each one paired with a counter. If the element currently in the input already exists in the k stored elements, its counter is increased by 1 (one) otherwise, if there are less that k stored elements, the new element is stored with its counter set to 1 (one). If there are already k stored elements, all counters are decreased by 1 (one). Elements with counter equal to 0 (zero) are removed from the stored elements.

Wei-Guang Teng et al. [TCY03] propose a method on finding itemsets in a data stream that are temporarily frequent. The general idea is to use a time-based sliding window of size W and compute the frequency of each itemset every time the window slides. Each itemset is assigned a cumulative frequency which equals its average frequency in all windows. The itemsets with cumulative frequency higher that some threshold θ are considered to be frequent. To avoid tracking all possible itemsets, Wei-Guang Teng et al. identify initially the single frequent items and combine them to create frequent itemsets. Only the itemsets that have been found to be frequent are tracked in the consequent windows and only as long as their cumulative frequency remains above the threshold θ. For space efficiency, each itemset is represented as a linear function of its cumulative frequency which losslessly can generate the separate frequencies at each time point. Itemsets that are frequent for the first time or are not frequent anymore can be considered to indicate some event.

Jeffrey Xu Yu et al. [YCLZ04] are also inspecting rapid data streams for frequent itemsets. They claim that most existing algorithms find frequent itemsets allowing false positives. This results in an explosion in the considered number of itemsets and thus in memory consumption. The authors argue that algorithms allowing false negatives instead are more appropriate. In this context, they propose an algorithm that uses Chernoff bound [Che52] to prune itemsets that are potentially infrequent.

3.2 Topics Detection

The work of Qiankun Zhan et al. [ZMC07] is among the first works on topic detection in social networks. At this time the focus was mainly on Blogs and web forums. Research using Twitter data started a bit later. The approach they propose is based on multiple levels of graph partitioning. They initially create a graph having a node for each document and an edge between two nodes depending on the content relation of the corresponding documents. They partition the graph creating groups of documents representing the same topic. Using the temporal aspect of the documents, they partition them again into groups of documents created at specific time windows. For the set of documents of each topic in the same time window, they create the dual graph which they partition based on the information flow pattern similarity between the social actors (senders, receivers). Finally, each event is represented as a graph of social actors connected through a set of documents, e.g. a blog post with its author and the people commented on it along with their comments.

Hassan Sayyadi et al. [SHM09] extract keywords from the documents published in social media and use them to create a graph. Each node in the graph is a keyword and two nodes are connected if the keywords exist in the same document and if finding one of the keywords in one document increases the probability of finding the other keyword in the same document as well. They split the graph in communities, i.e sets of keywords that are densely connected with each other, and consider each community to represent one topic. To separate the communities from each other, the authors use the betweenness centrality score. According to this, the score of an edge equals the number of shortest paths between all pairs of nodes that pass through this edge. The intuition is that edges with high scores are probably connecting nodes from one community to nodes from another community. Such edges are duplicated to allow keywords to belong to more than one communities.

Hila Becker et al. [BNG09] propose a clustering technique over documents created in the social media. According to their approach, various features of the documents and specific metrics for each of them, e.g. the cosine similarity over the documents body, or the Haversine distance for geolocation information, are used to create separate clusters. The clusters are created using a single pass algorithm with centroid similarity and at the end they are combined to create a single clustering solution where two documents belong to the same cluster in the final solution if they belong together in the majority of the separately created clusters.

Bharath Sriram et al. [SFD$^+$10] use a predefined set of categories like News, Events, Opinions etc. and they try to assign the tweets to these categories. For the categorisation they use, primarily, information extracted from the profile of the authors. Their basic motivation is the observation that authors tend to write messages with topics from a limited number of categories. In addition to the authors information, they analyse the text itself. Each category has a set of features associated with it. Extracting these features from the body of a message helps assigning the message to specific categories. For example, if a tweet contains a date and a place then it is assigned to the Events category.

Ana-Maria Popescu et al. [PPP11] use sets of tweets from various time periods (snapshots) and identify the entities mentioned in each of them. Afterwards, using Machine Learning techniques, they decide whether each snapshot regards specifically some entities or represents a generic discussion. As a final step, each entity is associated to a set of actions – sequences of the form *(entity, verb, action)* – and a set of opinions in each snapshot.

Kazufumi Watanabe et al. [WOOO11] identify events that are of small scale, i.e. they are of interest to a local community. Unless the documents contain geotagging information, they decide the place each document refers to by using the entities mentioned in it. More precisely, they bind each entity referring to a place to a real world location and they group documents according to the mentioned locations. For example, a document mentioning *Times Square* is placed in the group about Times Square in New York. Documents are further grouped based on their creation time. Documents created in the same period at the same place are likely to talk about the same local event. The co-occurring terms in each group of documents are analysed to finally decide whether the group refers to a specific event. If there is a set of terms co-occurring in many of the documents in the group then this set is considered to represent an event taking place at the specific location.

Andreas Weiler et al. [WMS12] inspect an incoming stream of tweets and spot the terms that exhibit unusually high rate. These terms are considered to represent events. They use the metadata found in the tweets to obtain additional information about the identified events. For example, they use the geotags to position the events at specific locations. Additionally, using DBPedia[1] and WordNet[2], they classify the events to categories. Using tweets from the past few hours, they enrich the events with terms that co-existed in the past with terms found in the events. Finally, they rank the events according to the number of retweets or total number of tweets.

Alan Ritter et al. [RMEC12] manually extract events from a subset of tweets. These tweets are used to train sequence models that are used later to extract events from unseen tweets. From each new tweet, they extract the entities mentioned in it and use the model they have trained to find the events described in them. They also extract from each tweet temporal expressions that are used later to decide whether the events described in the tweets are mandatory. The events extracted from the tweets are grouped into types using unsupervised classification techniques. Finally, the events are ranked according to their correlation with a specific date. They consider events strongly connected with a specific date to be more important while events evenly distributed in time are usually mandatory, for example tweets regarding lunch (*"Spaghetti for lunch today"*) are expected to be seen every day.

Milad Eftekhar et al. [EK13] extract the hashtags found in tweets and group them into tagsets based on their co-occurrences. These tagsets are organised in a lattice which is partitioned to create topics. The authors examine two types of

[1]http://dbpedia.org/
[2]http://wordnet.princeton.edu/

lattice partitioning. According to the first type, the lattice is partitioned using as a criterion the number and size of the partitions. According to the second type, a notion of weight is introduced reflecting the importance of each edge in the lattice. The partitions are created using the edge weights as a criterion. For the first partitioning type, they examine two partitioning subproblems. In the first one, the objective is to create the most possible partitions with each created partition having at least c tweets. To achieve it, they consider nodes with more than c tweets to be separate partitions and merge the remaining nodes until all partitions have at least c tweets. The objective in the second subtype is to create k partitions such that the number of tweets in each partition is maximised. The solution they propose is based on the min-max graph partitioning problem. They also examine two subproblems in the lattice partitioning using as a criterion the weight of each partition. The first subproblem is to create the k partitions with the maximum possible total weight. The second subproblem is to create k partitions in a way that the minimum weight is maximised. For both subproblems, they remove from the lattice the edges with a negative weight and get all remaining connected components. For the first problem, they return the k components with the maximum weight. For the second problem, the k components with the maximum weight are further partitioned in a way that maximises their weights.

Al the above works aim at identifying qualitative topics. They focus on data collected over previous days and analyse them in an offline manner. Such approaches might be of great interest to market analysts but we believe that are of limited interest to individuals or news agencies. They are particularly of no use to organisations or countries interested to prevent or limit the effects of, for example, natural phenomena or diseases. For such applications, online approaches for topics detection and more specifically, emerging topics detection in real time are needed.

3.2.1 Emerging Topics Detection

Many works exist on identifying emerging topics discussed in social media. Within these works, one can find various definitions for a topic. In our work, we define topics using the textual annotations and/or the entities found in the documents. More precisely, each topic is represented by a set of correlated tags. We track the correlation of the tagsets over time and identify emerging topics to be those topics represented by a tagset having an unexpected increase in its correlation. The idea of capitalising on the textual annotations to identify emerging topics is not new. However, existing works track over time single tags. The single tags that show an unexpected increase in their "interestingness" are then used to create tagsets that are considered to represent emerging topics. A sample of these works, along with works that identify emerging topics using alternative definitions for the topics, is presented here.

Nilesh Bansal et al. [BK07] present BlogScope, a system that allows users to find events discussed in the Blogosphere. More specifically, a set of bursty keywords, i.e. keywords that exhibit sudden increase in their popularity, is computed every day. A user initiates a query over blog posts by selecting any of these keywords. BlogScope searches blog posts to find keywords correlated to the selected keyword. It considers two keywords to be correlated if they frequently appear in the same posts. The maximum set of correlated keywords that are bursty is considered to represent an event and is presented to the user. BlogScope identifies the places where each event was interesting using the location of the authors that published the posts. Additionally, the time period during which the keywords were bursty is used to find the time when the event was interesting. Users can subscribe to keywords that reflect their interests and get notification about related emerging topics.

Toshimitsu Takahashi et al. [TTY11] propose a method for emerging topic detection based on links dynamically created between the users of a social network. The links are created through explicit mentions, retweets, replies or direct messages, all called simple mentions. They consider a stream of incoming messages and a window over it. For each message in the window, they sample messages of the same user created during the past T time intervals. They use these messages to create a model of the users mention behaviour which they use afterwards to detect a mentions anomaly in the new message. They aggregate the anomaly scores of all messages in the window obtaining a general trend. Using a change point technique based on the Sequentially Discounting Normalised Maximum Likelihood Coding [UYTI11], they detect changes in the statistical dependence of mentions. They issue an alarm if the score of the change point exceeds some threshold. The threshold is automatically selected using a method of dynamic threshold optimisation.

Shiva Prasad Kasiviswanathan et al. [KMBS11] propose a method of emergent topic identification in social media using dictionary learning. They define an emergent topic to be a topic that appears from many different sources and is different form topics that have been already seen in the past. In their approach, each document is modelled as a vector using the tf$\times$idf measure. Initially a set of documents is used to learn a dictionary of k atoms. The atoms are selected in a way that all the documents can be represented as a sparse liner representation of them. Every new documents is checked against this dictionary. If it cannot be represented as a linear representation of the k atoms with low error then the document is considered to be novel. They use the novel documents to learn a new dictionary of k_1 atoms. Each atom represents a topic and each novel document is assigned to the atom in which it has the most dominant representation resulting in a clustering of documents to topics.

Ankan Saha et al. [SS12] propose a method to identify emerging topics using non-negative matrix factorisation. Following the same technique to [KMBS11], they model each document as a vector using the tf$\times$idf measure. They use the documents to create a document-term matrix. This matrix is factorised resulting in two new matrices W and H. Each column in W and each row in H correspond to one topic. Matrix H represents the distribution of terms to topics while matrix W represents the distribution of topics to documents. Only the last k lines of H correspond to emerging topics. The other lines correspond to the topics found up to the previous time points. This existence of previous topics allows them, apart from identifying new topics, to track, at the same time, the evolution of old topics. To distinguish real topics from noise, they use the assumption that real topics will have a rapid increase in the number of documents associated with them.

Anish Das Sarma et al. [DSJY11] consider events to be represented by dynamic relationships among entities. They assume entities to be known in advance or obtained using an entity extraction tool. They use a time window over the incoming documents and create a temporal profile for each entity. This profile contains the time windows at which the entity was bursty. Entities that are bursty at the same time window are considered to be related to each other. The strength of the relation is measured using the point wise mutual information score [Tur01]. To detect all the entities involved in an event, they create an Entity Dynamic Relation (EDR) graph, i.e. a graph having one vertex for each entity and one edge between two vertices when a dynamic relation between the corresponding entities exists. They consider connected components in the EDR graph to represent different events. Adding in the graph information about past time windows allows them to track the evolution of events, in the sense of the involvement of entities in them, by considering connected components over a number of consecutive time windows.

Michael Mathioudakis and Nick Koudas [MK10] present TwitterMonitor, a system that identifies emergent topics over the Twitter stream. The paper is a demo description and many details regarding their approach are omitted. In general, TwitterMonitor is based on the common idea that a topic is a set of bursty keywords that occur frequently in the same tweets. The identification of emergent topics is performed in three steps:
Identify Bursty Keywords: Based on queue theory, they have developed an algorithm, `QueueBurst`, that identifies keywords that suddenly occur in high frequency. The authors claim that `QueueBurst` is able to identify bursty keywords in real time, performing only one pass over the data. Furthermore, the algorithm is adjustable against faulty bursts and spam.
Identify Bursty Topics: Bursty keywords, identified in the previous step, are grouped into disjoint sets forming topics. For this process, they use the tweets received during the previous minutes and find on them bursty keywords that appear together in relatively many tweets.
Analyse Bursty Topics: As a final step, for each group of bursty keywords, they check the tweets related to it for additional, non-bursty, entities. Moreover, when possible, they provide external links to news portals discussing the stories identified by TwitterMonitor to be emergent.

3.3 Top-k Queries Over Data Streams

Many works exist on processing top-k queries in a traditional RDBMS environment. Most of these works are based on the family of threshold algorithms by Ronald Fagin et al. [FLN01]. Despite all the available work on efficiently creating top-k rankings (see the survey of Ilyas et al. [IBS08] for an overview) the top-k aggregate rankings having group-by conditions, which we consider, have not attracted enough attention. To the best of our knowledge the only existing approaches are [LCCCI06] and [YMH08].

Chengkai Li et al. [LCCCI06] assume that they know a-priori the number of tuples in each group or an upper limit about it and propose a methodology very close to NRA algorithm; the maximum potential score of each group is computed using the real score of the seen tuples and the maximum possible score of the unseen tuples. When retrieving a new tuple, they always get a tuple from the group with the maximum possible score. If tuples within the groups are ordered, the tuple with the maximum score is selected, dropping the maximum possible score for all unseen tuples and resulting on faster identification of the top-k groups.

Man Lung Yiu et al. [YMH08] also propose an algorithm based on NRA. They assume the tuples in a relation are stored in decreasing score order. Additionally, similarly to [LCCCI06], they assume the number of tuples per group is known a priori. For every retrieved tuple, they update the maximum possible score of all groups. They prune groups that have a maximum possible score not inside the top-k ones. They also propose an algorithm that does not assume any ordering on the tuples. The basic idea is to randomly select a number of tuples and use them to guess the top-k groups. For these groups, the exact scores are computed while for all other groups a hash table is used. The algorithm is focusing on memory usage rather than time efficiency since the relation has to be scanned multiple times until the top-k groups are identified. Every scan however, processes less tuples since buckets that have very low scores are discarded.

Both the above works focus on algorithms processing data that are relatively static. Additionally, they compute top-k queries on demand. In this thesis, we continuously monitor top-k rankings and try to notify the users for changes in them instead of waiting for the users to initiate the queries. We also consider more dynamic environments, i.e. quick, continuous updates affect the data.

Kyriakos Mouratidis et al. [MBP06] compute and maintain the top-k elements in a data stream with valid tuples restricted by a sliding window. They index the elements in the window in a grid based on the values they have in their attributes and they select the boxes in the grid that can have elements qualifying for the top-k results. Any monotonic function can be used to detect these boxes. Only new elements added in boxes that contain elements which potentially belong in top-k are inserted or deleted from the top-k list. If it is the case that not enough elements exits in the selected boxes the query is re-evaluated, i.e. new boxes are found. More than the top-k elements can be stored to avoid such cases using a k-skyband instead. The algorithms are trivially extended to elements with more than two attributes.

Parisa Haghani et al [HMA10] assume a stream of documents and a set of queries expressed as profiles defined as sets of tags with a weight on each tag. They propose a method to maintain a separate list with the top-k most relevant documents for each profile. The method they propose is very similar to the TA algorithm. They create for each tag a list of profiles sorted on the weight of this tag on each profile. For each incoming document, they get the lists of profiles for all the tags found in the document and compute the score of the document for each profile. The threshold is computed on the m lists with the highest minimum scores, where m is the maximum number of tags defined in any profile. To accelerate the procedure, they alternatively propose to maintain sorted lists of groups of profiles for each tag where the profiles in each group will be unsorted.

Rui Zhang et al. [ZKOS05] compute group by aggregate queries over data streams. Their main idea is based on Gigascope [CJSS03], a stream database. According to Gigascope, the aggregate computation is split in a two levels hierarchy. Received elements are hashed into buckets in the first level. If the bucket contains elements belonging to the same group with the new element, its aggregate is updated. Otherwise, the group is moved to the second level and a new group, the one of the received element, is created in the bucket. At specific time intervals, all groups are moved from the first level to the second. [ZKOS05] assumes multiple queries that differ only on the group by condition and proposes processing them together creating other groups, called phantoms, in order to have less groups to maintain. The answers to initial groups should be derived by the phantom ones. They derive a cost model on the phantoms creation and maintenance which allows to decide on one of multiple possible phantom combinations.

Ahmed Metwally et al. [MAEA05] propose a method that identifies the top-k most frequent elements in a data stream especially in cases when the distribution of frequencies is skewed. The idea is to monitor N elements at any time. If a received element is monitored then the corresponding counter is updated otherwise, the element with the minimum counter is removed and the new element is put in its place. The counter of the new element is set equal to the counter of the removed element increased by one. The intuition is that frequent elements will never be in the last position of the monitored elements and thus they will never be replaced.

3.4 Materialised Views Maintenance

Maintaining materialised views has been a topic of research. José A. Blakeley et al. [BLT86], in this early work, propose criteria that allow identifying whether an update (insertion or deletion) affects the content of a view, i.e. it is relevant to the view. They consider views created using selections, projections, joins or all of these together and propose for each of them a methodology that allows to update them identifying the tuples that should be added or deleted. This way of updating the views is called delta or differential update.

Yue Zhuge et al. [ZGMHW95] extend the methods proposed in [BLT86] to handle multiple concurrent updates. In such a case using differential updates might result in faulty materialised views. Assume for example multiple insertions. After each of them, the tuples that should be added to the view are computed. If the updates happen more frequent than the estimation of the new tuples then the same tuples could be added multiple times. To avoid that, they propose the use of compensate queries which identify and delete tuples added multiple times.

Ashish Gupta et al. [GJM96] introduce the notion of self-maintainable views, i.e. views that can be maintained using only the changes caused in the underlying relations and the contents of the view itself. Similar to the previous works, the views that they examine are created using selections, projections and/or joins. They define a number of theorems which identify whether a view is self-maintainable or not based on its characteristics.

Sunil Samtani et al. [SKM99] propose, instead of maintaining each materialised view separately, to share computations and tuples when possible. They propose creating intermediate views, called auxiliary views, and use them as shared repositories between multiple materialised views. This way, each materialised view has to be derived/computed using the tuples in the auxiliary views. This causes a delay in the update of the views but decreases the space requirements since it avoids storing duplicate tuples.

Along the same lines is the work of Hoshi Mistry et al. [MRSR01]. They assume a number of permanently materialised views and propose materialising extra views in an attempt to share update computations between various materialised views. The additional views can be permanent or transient in case their continuous maintenance is very expensive.

Ke Yi et al. [YYY$^+$03] focus on materialised top-k views and attempt to make them self-maintainable. They propose, instead of materialising the top-k results, to materialise the top-k' results, where, of course, $k' > k$. According to their approach, k' is initially set to k_{max}. For every update or insertion that causes a tuple to enter the top-k' results k' is increased by one. For every update or deletion that causes a tuple to be removed from the top-k' results k' is decreased by 1. When $k' < k$, k' is reset to k_{max} and the view is recomputed. Depending on the skewness of the data and the updates, the proposed technique can significantly delay re-computations while materialising only a small subset of the data.

Our work on identifying events over top-k rankings created in a relational database could use many of the techniques proposed in the area of materialised view maintenance. However, in the case we study we cannot really benefit from all these techniques since we materialise only a small portion of all possible results when the majority of these approaches assumes fully materialised results. We benefit, though, from the ideas presented in [YYY$^+$03]

Chapter 4

enBlogue: Emergent Events Identification in Social Media

The user generated content published in the Web 2.0 contains information about events happening around the world, in a specific country or in a specific neighbourhood. This makes it a very valuable resource of news information. However, the big amount and fast rate that this content is generated at can easily overwhelm users. We propose a method which identifies emerging topics analysing short messages such as tweets and blogs. We achieve this by capitalising on the textual annotations that are usually found in these messages. A demonstration paper of this work has been published at the ACM International Conference on Management of Data (SIGMOD 2011) [ASRW11] and the full paper regarding it has been published at the ACM International Conference on Extending Database Technology (EDBT 2012)[AMRW12].

We focus our discussion on Twitter due to its very massive nature, but our method is not limited to it. We define topics to be represented by combinations of tags, i.e. tagsets. Consider for example the tweet `''#Bieber getting a #tatoo of Selena #Gomez was a little uncesisaary''` (sic) commenting on Justin Bieber's new tattoo being inspired by his former lover Selena Gomez. The tagset {*Bieber, Gomez, tattoo*} could be used to represent this topic. Additionally, or alternatively, to the tags explicitly used by the users, named entities extracted from the body of the messages can be used as topic indicators. Using the named entities extracted from the above tweet, the tagset representing the topic would be {*Bieber, Selena, Gomez, tattoo*}.

One should not confuse popular or hot topics with emergent topics. Hot topics are the topics that gather a lot of interest, i.e. many documents are commenting on them. Emergent are the topics that attract unusual interest compared to the interest they have gathered in the past. Emergent topics are very likely not to be popular at the time they are identified, but they might become popular after short time as more users become aware of them and start commenting on them. We try to identify these topics before they become popular.

Since Barack Obama was announced to be a candidate US president, the tag

Obama on Twitter has been a very popular one, i.e. many messages are usually annotated with the hashtag *#Obama*. The tag *Mandela* on the other hand, became popular after the death of Nelson Mandela on the 5th of December 2013. Finding messages annotated with both tags was relatively uncommon until the 10th of December 2013. At this day, Barack Obama attended the memorial service held for Nelson Mandela. The popularity of this pair of tags lasted for a few days and dropped again to (almost) zero.

The above example suggests that tracking the popularity of single tags is not always enough to identify emerging topics. This is especially true for tags that are in general popular. For not popular tags, it is likely, although not certain, to spot a burst when a related event occurs. Figure 4.1 depicts an example on how the number of documents related to two tags t_1 and t_2 , a popular one and a less popular one, and the number of documents related to both of them change over time. In this figure, one can potentially identify two events, one related to the tag t_2 and another related to the tagset $\{t_1, t_2\}$. The two events are not aligned in time thus, tracking only single tags, i.e. t_1 and t_2 separately, allows identifying events related to each of them but is not enough to detect the event related to both of them, i.e. the tagset $\{t_1, t_2\}$. This principle can be generalised in tagsets of bigger size, i.e. tracking the tagset $\{t_1, t_2\}$ is not enough to detect events related to the tagset $\{t_1, t_2, t_3\}$ and so on.

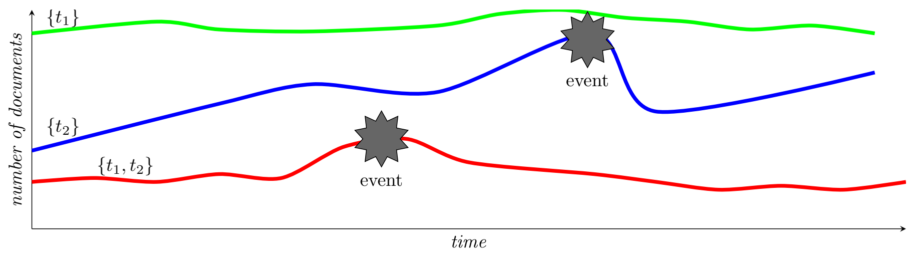


Figure 4.1: Tags over time

The bigger the size of a tagset representing a topic, the more specific the topic becomes. This might be useful allowing users to better understand the real event behind the tagset representation. For example, on the 11th of December 2013 an increase observed in the documents annotated with the tag *#selfie*. Looking at this single tag is very unlikely to guess the specifics of the event. Luckily the messages were also annotated with the tags *Obama* and *Mandela* commenting on the selfie shot by Barack Obama during the memorial service for Nelson Mandela. On the other hand, very specific events tend to be volatile, i.e. disappear very quickly, and are more likely to be created by spammers.

4.1 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 $s_i = \{t_1, t_2, \ldots\}$ and accompanied by a timestamp tm_i imposing an ordering on the documents. We use a time-based sliding window of size $\mathcal{W}$ over the documents to limit the focus to the most recent of them. The window slides every w time units. We want, immediately after sliding the window, to identify and report

$\mathcal{W}, w$	The size of the sliding window, The units the windows slides every time
$\mathcal{S}, s_i$	A set of tagsets, A tagset/A topic
$\mathcal{TG}, t_i$	A set of tags, A tag
min	Minimum number of documents to qualify a tag for further analysis
T_i	A set of documents ids annotated with tag t_i
d_i	A document id
tm_i	The timestamp of document d_i
ρ	Number of previous values for computed and predicted correlation, and popularity stored for each tagset (topic)
$\mathcal{CR}^{s_i}, cr_j^{s_i}$	Set of ρ previous computed correlations for the tagset s_i, Computed correlation for the tagset s_i at timepoint j
$\mathcal{CR}_{pr}^{s_i}, cr_{pr_j}^{s_i}$	Set of ρ previous predicted correlations for the tagset s_i, Predicted correlation for the tagset s_i at timepoint j
$\mathcal{POP}^{s_i}, pop_j^{s_i}$	Set of ρ previous popularity values for the tagset s_i, Popularity values for the tagset s_i at timepoint j

Table 4.1: Notations used in the chapter

the set of the k most interesting emergent topics mentioned in the documents currently in the window.

4.2 Computational Model

We represent each document $d_i \in \mathcal{D}$ using a triple of the form: (tm_i, d_i, s_i). We consider that the tags in s_i have been selected to reflect the topic of d_i. All sets of tags s_j, where $s_j \subset s_i$, reflect some aspect of the topic of d_i. The bigger the size of a tagset s_j, i.e. the more the tags it contains, the more specific becomes the description of the topic.

Consider, for example, the document `‘‘#Obama says U.S. engagement with #China will not come at the expense of #Japan’’` obtained through Twitter. In Twitter, each tweet represents a document. One tagset extracted from this tweet is s_1 = {`#Obama`}. This tagset indicates that the topic of the tweet is related to the president of the US. Another tagsets is s_2 = {`#Obama, #Japan`} which reveals that the topic of the tweet is related to both Barack Obama and the country of Japan. In total, seven different tagsets can be extracted from the above tweet (s_1 = {`#Obama`}, s_2 = {`#Japan`}, s_3 = {`#China`}, s_4 = {`#Obama, #Japan`}, s_5 = {`#Obama, #China`}, s_6 = {`#Japan, #Chine`}, s_7 = {`#Obama, #Japan, #China`}). Each one of them will indicate a topic related to the tweet.

The computational model we consider is the following: We extract all tags from the documents within the bounds of the sliding window and combine them to create all sets of co-occurring tags. For each set of co-occurring tags, we compute a correlation measure. We track the tagsets' correlations over time and identify unexpected bursts in them. We consider an unexpected burst in the correlation to indicate an emergent topic represented by the tagset exhibiting the burst.

4.3 Approach

We consider a stream $\mathcal{D}$ of triples (tm_i, d_i, s_i), where s_i is a set containing tags that describe the topic of the document d_i. We impose a time-based sliding window over $\mathcal{D}$ that restricts the focus to the most recent triples. We propose an approach to identify emerging topics using the triples currently in the window. The approach consists of the following steps:

1. **Seed tags selection:** Seed tags are a subset of all the tags in the current window. They can be used to limit the tagsets considered in the following steps to those having at least one seed tag. Seed tags can be determined based on different criteria, such as popularity or volatility.

2. **Correlation Tracking:** Using the triples currently in the window, we compute for each considered set of tags a correlation measure.

3. **Shift Detection:** We inspect the temporal changes in the correlations computed in the previous stage in order to identify temporal bursts in them.

4.3.1 Identifying Seed Tags

In the presence of excessive amounts of tagsets, seed tags can be used to limit the tagsets considered for further processing to those having at least one seed tag. From all the tags found in the current window, we select the k most popular of them to be used as seed tags. The rationale is that for an emergent topic to be interesting at least one of the tags representing it should be “hot” by itself.

The algorithm used to select the seed tags is shown in Algorithm 1. The input to this algorithm are the triples (tm_i, d_i, s_i) present in the current window. These triples are used to compute the popularity of all tags in the window (Algorithm 1, line 8) of which the top-k most popular ones are used as seed tags (Algorithm 1, lines 7 - 13). A *min* parameter is additionally provided. This parameter defines the minimum number of documents a tagset should appear in, in order to be considered for further processing. In Twitter, it is not uncommon to have pairs of tags that exist only in a very few documents, i.e. 1 or 2. To get rid of these pairs, we use the *min* parameter. In this context, we disregard from seeds tags those that do not appear in at least *min* documents.

4.3.2 Measuring Tag Correlations

For any set of co-occurring tags $s_i = \{t_1, t_2, \ldots\}$, we need to compute how strongly connected the tags are. We need a measure that will allow us to estimate whether the tags represent a topic or they are randomly put together (probably by a spammer). Generally speaking, a good measure should reflect:

- **Local Importance:** How popular is the topic in the community of users interested in any of its aspects

Algorithm 1: Identify Seed Tags

```
Input: Set of triples $\mathcal{TP} = \{(tm_1, d_1, s_1), (tm_2, d_2, s_2), \ldots, (tm_n, d_n, s_n)\}$,
       Integer $min$, Integer $k$
Result: Set of $k$ tags
/* Find all tags currently in the window */
$\mathcal{TG} = \{\}$ // total set of tags in the current window
foreach $(tm_i, d_i, s_i) \in \mathcal{TP}$ do
    $\mathcal{TG} = \mathcal{TG} \cup s_i$
end foreach
/* Find the k most popular tags currently in the window */
$\mathcal{TG}_{topk} = \{\}$ // top-k popular tags in the current window
Find $T_p$ : $t_p$ the least popular tag in $\mathcal{TG}_{topk}$
foreach $t_j \in \mathcal{TG}$ do
    $T_j = \{d_i | t_j \in s_i \wedge (tm_i, d_i, s_i) \in \mathcal{TP}\}$
    if $(|T_j| > |T_p|$ && $|T_j| > min)$ then
        replace $t_p$ with $t_j$ in $\mathcal{TG}_{topk}$
        update $T_p$
    end if
end foreach
return $\mathcal{TG}_{topk}$
```

- **Global Importance:** How popular is the topic in the whole community

Consider two tags t_1 and t_2. Each of these tags is accompanied by a set T_i containing all document ids of the documents in the current window that are associated with tag t_i. *A document d_j is associated with a tag t_i if the tag has been explicitly used to indicate the topic of the document d_j or if tag t_i has been extracted from the body of the document d_j using an annotation extractor mechanism.* Out of all the documents associated with either tag t_1 or tag t_2 a subset is associated with both t_1 and t_2. To estimate the local importance of the topic represented by the tagset $\{t_1, t_2\}$, one should compute the similarity of the sets T_1 and T_2. The more similar these sets are, the more important the topic $\{t_1, t_2\}$ is to the users interested in any aspect of it. One of the most common measures for the similarity of two sets is the Jaccard coefficient.

The global importance of a topic is estimated by the general interest of all users in it. The measure we use to compute it is the frequency of the documents associated with the tagset in the whole collection. The product of the two measures estimating the global and the local importance of a topic is used to measure the correlation of the tags representing the topic.

For any set of tags $s_i = \{t_1, t_2, \ldots\}$, the correlation is computed using the Formula 4.1:

$$corr(t_1, t_2, \ldots) := \overbrace{\frac{|\bigcap_i T_i|}{|\bigcup_i T_i|}}^{local\,importance} \times \overbrace{\frac{|\bigcap_i T_i|}{N}}^{global\,importance} \tag{4.1}$$

We know that $0 \leq \frac{|\bigcap_i T_i|}{|\bigcup_i T_i|} \leq 1.0$ and $0 \leq \frac{|\bigcap_i T_i|}{N} \leq 1.0$ thus, $0 \leq corr(t_1, t_2, \ldots) \leq 1.0$

4.3.3 Shift Detection

A topic is considered to be emergent when its behaviour deviates from what is expected, similar to [MP03]. The more the observed behaviour deviates from the expected behaviour, the more emergent the topic is.

The behaviour of a topic is said to be expected if it can be predicted from its previous behaviour. The prediction of the behaviour of a topic is attempted with the use of exponential smoothing [Bro63]. Exponential smoothing is a forecasting technique that uses a weighted moving average of past data as the basis for the forecast. The process gives greater weights to most recent observations and smaller weights to observations in the more distant past. The reason is that the future value is likely to be more dependent upon the recent past. Equation 4.2 depicts the exponential smoothing formula:

$$\hat{v}_t = av_{t-1} + (1-a)\hat{v}_{t-1} \tag{4.2}$$

where v_{t-1} is the previously observed value and $\hat{v}_{t-1}$ is the previously predicted value. a is a smoothing parameter defining the importance of the previously observed value and the importance of the previously predicted value in the prediction of the current value.

Definition 1. *A topic represented by a set of tags $s_i = \{t_1, t_2, \ldots\}$ is emergent if the computed correlation value of the tags $t_1, t_2, \ldots$, represented as $v_t^{s_i}$, is larger than its predicted correlation value, represented as $\hat{v}_t^{s_i}$.*

The difference between the computed and the predicted correlation values of a topic represented by a tagset s_i is called prediction error. Using the prediction error to estimate the emergence of a topic results in overestimating the emergence of the tagsets found in the current window for the first time (the predicted correlation value is zero). At the same time, any tagset having a current correlation value higher that its previous, non-zero, correlation value might be overlooked. To avoid this we use instead the relative prediction error, defined as

$$er_t^{s_i} = \frac{v_t^{s_i} - \hat{v}_t^{s_i}}{v_t^{s_i}} \tag{4.3}$$

4.3.4 Scoring

The relative prediction error detects the topics that are emergent, but it cannot detect out of them the ones that are more interesting. A good measure for the interestingness of a topic is its popularity measured as the frequency of the tagset representing the topic. We measure the score of an emergent topic represented by the tagset s_i using Equation 4.4.

$$sc_t^{s_i} = \frac{er_t^{s_i}}{|\ln(pop_t^{s_i})|} \tag{4.4}$$

where $pop_t^{s_i}$ is the popularity of the topic at the t timepoint, $0 < pop_t^{s_i} \leq 1.0$.

We choose to divide by the absolute value of the natural logarithm of the popularity and not just to multiply with the popularity because the natural logarithm has the property of dampening the effect popularity has on the final score. This means that if a not so popular topic s_1 and a popular topic s_2 have very different relative errors with $er_t^{s_1}$ being greater than $er_t^{s_2}$ then it is more difficult for s_2 to be considered more emergent than s_1 just because it is more popular.

Example: Consider four topics s_1, s_2, s_3 and s_4 and their scores sc_1, sc_2, sc_3 and sc_4 respectively. Assume that the relative error of the topic s_1 is related to the relative error of the topic s_2 by $er_t^{s_1} = 1.010 \times er_t^{s_2}$ and the relative error of the topic s_3 is related to the relative error of the topic s_4 by $er_t^{s_3} = 1.100 \times er_t^{s_4}$. Considering the relative errors of the topics the relative rankings of them are $sc_t^{s_1} > sc_t^{s_2}$ and $sc_t^{s_3} > sc_t^{s_4}$. In order for the popularities of the topics to be able to affect those rankings they should be related as shown by the formulas $pop_t^{s_2} = 1.047 \times pop_t^{s_1}$ and $pop_t^{s_4} = 1.520 \times pop_t^{s_3}$. In the simple case, where the relative error is just multiplied with the popularity, a relation between the popularities given by the formulas $pop_t^{s_2} = 1.011 \times pop_t^{s_1}$ and $pop_t^{s_4} = 1.101 \times pop_t^{s_3}$ suffices to change the relative ranking of the topics.

The previous example shows that by using the natural logarithm of the popularity it is more difficult to affect the ranking of the topics. Moreover an increase in the factor relating the relative errors of 8.9% (from 1.010 to 1.100) needs an increase in the factor relating of popularities of 45.2% (from 1.047 to 1.520) for the ranking to be affected. In the simple case this increase in the factor of popularities is just 8.9% (from 1.011 to 1.101), the same as the increase in the ratio of relative errors.

Thus, using the natural logarithm of the popularity lessens the influence of it in the final score. Since we are interested in emergent topics and not in hot ones, this is a desired result. We avoid the situation that the overall score depends too much on the popularity, but at the same time a big difference in the popularity, compared to the difference in the relative errors, is able to change the ranking of the topics.

4.3.5 Score Smoothing

Naturally, if a topic's behaviour does not change much with time, the capability of predicting the next value improves and the topic is not considered emergent anymore. Intuitively though, we can say that a user does not loose interest in a topic from one moment to another. For example, in case of a big scandal almost every newspaper has an article about it in the first day. The following days the scandal is not a surprise anymore, but the newspapers have articles about it since it is still of some interest to the users. Hence, the interestingness of one day, e.g. the first day the event occurred, should carry over to other days, with a dampening factor, obviously.

This intuition is confirmed by the observation made in articles from the New York Times archive. We used this source as an example because we believe that the newspaper editors have a good understanding of how long an event is interesting to the consumers. We discovered that very often the number of articles referring to one specific event decreases through time by a factor of

$$penalty(\Delta t) := e^{-\lambda \Delta t}$$

where Δt is the distance in time from the moment when the most articles for the topic were written. After testing a few dozens of topics, we obtained on average a value of $\lambda = 0.38$. This means that an interestingness score of yesterday gets dampened by a factor of $e^{-0.38} = 0.68$, the score obtained two days ago by a factor of 0.46, and so on.

Fang Wu and Bernardo A. Huberman in [WH07] conducted a similar study on the decay of the novelty of a story published in a website. Similar to our results, they found that the attention attracted by a story or the interestingness of a story fades following an exponential law with decreasing rate $e^{-0.4*(\tau-t)}$ every hour.

Dampening each previous score by the appropriate factor, we get a set of scores for each topic. One score, dampened, for each point in the past and one score for the current point which is obviously not dampened as $\Delta t = 0$. The biggest of these scores is finally assigned to the topic, giving a chance to old "surprises" to influence a topic's ranking position today.

$$score_{\tau}^{s_i} := max_{t \leq \tau} \left\{ \frac{er_t^{s_i}}{|\ln(pop_t^{s_i})|} * e^{-\lambda*(\tau-t)} \right\}$$

The dampening factor found from the study of the New York Times articles seems to produce nice results when applied to blog posts but not when applied to tweets. This is obviously due to the fact that topics discussed in Twitter have a fast refresh rate, while topics discussed in blogs have a refresh rate that resembles that of newspapers. The dampening factor $e^{-0.38\Delta t}$ has a half life of 1.8 evaluations. Since topics in Twitter change very often, we are obliged to evaluate new emergent topics more frequent, e.g. every hour instead of one day which is the case for newspapers. This means that with the use of the above dampening factor a topic looses its half score after the second hour. This is too fast and a smaller dampening factor is needed. By experimentally testing various dampening factors, we concluded that the factor $e^{-0.2\Delta t}$, which has half life of 3.4 evaluations, is more appropriate for Twitter.

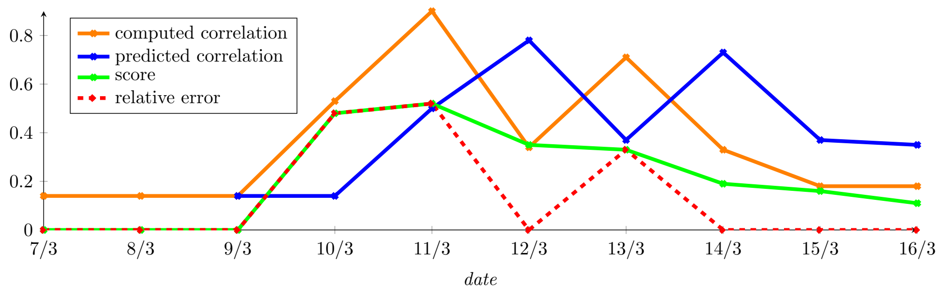


Figure 4.2: Interesting shift in correlation of two tags.

Since the number of previous values that we can store for each topic is limited, the final score may not be the largest of all the observed scores of this topic but just the greatest of the scores that we store. In practice, this is not a restriction as the dampening factor de-facto erases former scores after a couple of time units in any case. Figure 4.2 shows an illustration of the score derivation. The red dashed line depicts the change in the relative error. At each timepoint, the relative error represents the surprise related to the topic. The green line depicts the final score assigned to the topic at each time point. The green line declines over time and the changes in it are much smoother than the changes in the read line. This behaviour fits better the change in the interest of the users to emergent topics.

Algorithm 2 shows the procedure followed when computing the score of a topic represented by the tagset s_i. Three sets are given as input to this algorithm. A set $\mathcal{CR}^{s_i}$ having $\rho + 1$ values; the correlation values computed during the previous ρ evaluations along with the correlation value computed during the current evaluation. A set $\mathcal{CR}^{s_i}_{pr}$ having the correlation values predicted during the previous ρ evaluations and a set $\mathcal{POP}^{s_i}$ having the popularity values computed during the previous ρ and the current evaluation.

The first step of the algorithm is to predict a correlation value for the topic (Algorithm 2, Line 2). It achieves it using the correlation values computed and predicted during the previous evaluation in the exponential smoothing equation (Equation 4.2). Afterwards, the algorithm iterates over the ρ previous and the current correlation values and for each one computes a score for the topic. Each score is dampened by an appropriate factor which for the current score is 1 since $\Delta t_{\rho+1} = 0$, i.e. the current score is not dampened. The final score returned by the algorithm is the biggest of the smoothed scores (Algorithm 2, Lines 4 - 15).

Algorithm 2: Score a Tagset s_i

```
   **Input**: Set of computed correlation values $\mathcal{CR}^{s_i} = \{cr_1^{s_i}, cr_2^{s_i}, \ldots, cr_{\rho+1}^{s_i}\}$
   Set of predicted correlation values $\mathcal{CR}^{s_i}_{pr} = \{cr_{pr_1}^{s_i}, cr_{pr_2}^{s_i}, \ldots, cr_{pr_\rho}^{s_i}\}$
   Set of popularity values $\mathcal{POP}^{s_i} = \{pop_1^{s_i}, pop_2^{s_i}, \ldots, pop_{\rho+1}^{s_i}\}$
   **Result**: Double $score$
1  $score = 0$
2  **Predict** current correlation $cr^{s_i}_{pr_{(\rho+1)}}$
3  $\mathcal{CR}^{s_i}_{pr} = \mathcal{CR}^{s_i}_{pr} \cup cr^{s_i}_{pr_{(\rho+1)}}$
4  **foreach** $cr_j^{s_i} \in \mathcal{CR}^{s_i}$ **do**
       /* number of evaluation since this value was computed */
5      $\Delta t_j = (\rho + 1 - j)$
6      $er_j = \frac{cr_j^{s_i} - cr_{pr_j}^{s_i}}{cr_j^{s_i}}$
7      **if** *Twitter Dataset* **then**
8          $score_j = \frac{er_j}{|\log(pop_j^{s_i})|} \times e^{-0.2 \times \Delta t_j}$
9      **else**
10         $score_j = \frac{er_j}{|\log(pop_j^{s_i})|} \times e^{-0.38 \times \Delta t_j}$
11     **end if**
12     **if** $score_j > score$ **then**
13         $score = score_j$
14     **end if**
15 **end foreach**
16 **return** $score$
```

4.4 Implementation

We have implemented a full-fledged prototype system, called enBlogue. The implementation is done in Java 1.6 and follows the standard concepts of a push-based architecture for stream processing. At the data source level, it consists of several wrappers that either consume live streams or replay existing datasets for experiments. Data is represented in an array n-tuple format, consumed by stream operators, and pushed along producer-consumer edges in query-processing plans. The sliding window we use is time-based.

Figure 4.3 shows the workflow. Every block in the illustration represents an operator. Each operator receives data from the previous operator, processes them and pushes the results to the next operator.

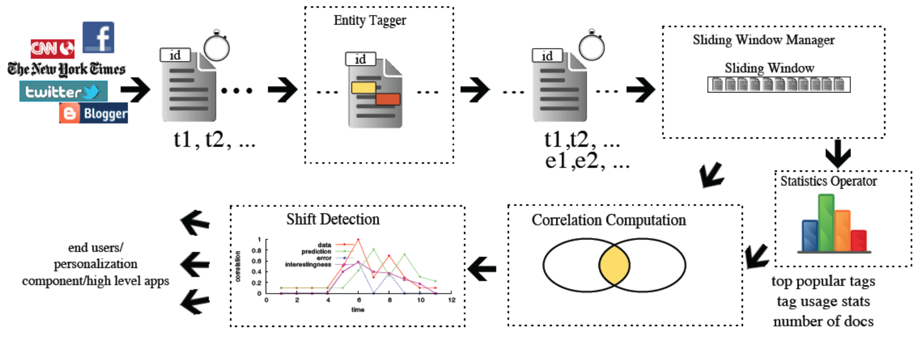


Figure 4.3: Workflow Illustration

Entity Tagger

The first step in the workflow is to obtain the set of tags representing the topic of each document. Some of these tags have been used by the creator of the document for this reason and are extracted from the document's metadata. Additional tags, e.g. names of persons are found in the body of the document. We have implemented a preprocessor operator, the Entity Tagger, that extracts these tags. The Entity Tagger uses an automatic entity extractor tool implemented in our group and finds in the document's body entities like persons, organisations and places. These entities are added in the set of tags that represent the topic of the document and are treated afterwards as common tags. For each incoming document, the Entity Tagger outputs a triple of the form (tm_i, d_i, s_i).

Sliding Window Manager

The Sliding Window Manager is responsible to decide which triples *(*tm_i*,* d_i*,* s_i*)* belong in the sliding window at any time. The sliding window is represented by its upper and its lower time bounds. The distance of these bounds equals the size $\mathcal{W}$ of the window and depends on the application. For the Twitter, for example, a reasonable valued of $\mathcal{W}$ could be 1 hour. The Sliding Window Manager blocks the arriving triples that have a timestamp smaller than the upper limit of the window. Triples arriving out of the order, i.e. having a timestamp smaller than the lower limit of the window, are discarded. When a triple with timestamp greater than the upper limit of the window is received, the Sliding Window Manager releases all the tuples to the rest of the operators. We call this moment *evaluation point* and the period between two evaluation points *evaluation period*. At the evaluation point, the Sliding Window Manager slides the window by w time units updating its upper and lower bounds. Additionally, it discards all triples with timestamp smaller than the current upper limit of the window.

The Entity Tagger and the Sliding Window Manager are continues operators in the sense that they process each incoming document as soon as it arrives in the system. The other operators, that receive the triples the Sliding Window Manager releases, operate only during each evaluation phase. Their execution starts as soon as they receive input from an operator preceding them in the workflow and ends at the moment they release their results to the operators following in the workflow. It is during the evaluation phase that the main steps of our approach, described in Section 4.1, are executed.

Statistics Operator

The Statistics operator comes in the workflow immediately after the Sliding Window Manager. It gets the triples released by the Sliding Window Manager and outputs the seed tags of the current evaluation. The seed tags are found using the Algorithm 1 presented in Section 4.3.1.

Correlation Computation Operator

The Correlation Computation operator takes the triples release by the Sliding Window Manager and the seed tags $\mathcal{TG}_s$ computed by the Statistics operator and finds all tagsets that represent some topic. *A tagset represents a topic if all its tags co-exist in at least min documents and at least one of its tags is a seed tag.*

Algorithm 3 outlines this process. Due to efficiency problems, in the current implementation only tagsets of size 2 (pairs) are considered. For each tag t_i, Algorithm 3 computes the set of documents T_i that are associated with it (Algorithm 3, Line 6). Using these tags and the seed tags, the algorithm creates all possible pairs that contain at least one seed tag. Only the pairs having at least *min* documents associated with them are further considered. For each such pair, the *Correlation Computation* operator computes and stores its popularity and Jaccard coefficient (Algorithm 3, Lines 9 - 15).

Apart from the tagsets representing topics according to the documents seen in the current evaluation, there are also tagsets representing topics identified in previous evaluations. The Correlation Computation operator updates also their current popularity and correlation values

The complete algorithm executed by the Correlation Computation operator is shown in Algorithm 4. For each topic found in the current evaluation (Algorithm 4, Line 2), the Correlation Computation operator computes its current popularity and correlation value and adds the new values to the tuples stor-

Algorithm 3: Find New Topics

```
Input: Set of triples $\mathcal{TP} = \{(tm_1, d_1, s_1), (tm_2, d_2, s_2), \ldots, (tm_n, d_n, s_n)\}$,
    Set of tags $\mathcal{TG}_s = \{t_1, t_2, \ldots, t_k\}$, Integer $min$
Result: Set of tuples $\mathcal{NEW} = \{(s_1, pop^{s_1}, jac^{s_1}), \ldots, (s_l, pop^{s_l}, jac^{s_l})\}$
/* Find all tags currently in the window */
$\mathcal{TG} = \{\}$ // total set of tags in the current window
foreach $(tm_i, d_i, s_i) \in \mathcal{TP}$ do
    $\mathcal{TG} = \mathcal{TG} \cup s_i$
end foreach
/* Find the documents associated with each tag currently in
   the window */
foreach $t_j \in \mathcal{TG}$ do
    $T_j = \{d_i | t_j \in s_i \wedge (tm_i, d_i, s_i) \in \mathcal{TP}\}$
end foreach
/* Create all tag pairs that have at least one seed tag */
$\mathcal{NEW} = \{\}$ // tag pairs considered for further processing
foreach $t_i \in \mathcal{TG}_s$ do
    foreach $t_j \in \mathcal{TG}$ do
        /* Store only the tag pairs that co-exist in at least
           min documents */
        if $|T_i \cap T_j| \geq min$ then
            $\mathcal{NEW} = \mathcal{NEW} \cup (\{t_i, t_j\}, |T_i \cap T_j|, \frac{|T_i \cap T_j|}{|T_i \cup T_j|})$
        end if
    end foreach
end foreach
return $\mathcal{NEW}$
```

ing the previous ρ values for them (Algorithm 4, Lines 2 - 12). The current popularity and correlation values for the topics found in any of the previous ρ evaluations but not in the current one are set to 0 (Algorithm 4, Lines 13 - 19). Topics that were not found in the current and any of the previous $\rho - 1$ evaluations are not considered further.

Shift Detection Operator

The Correlation Computation operator uses a number of threads to find the topics. Each thread is assigned a subset of the seed tags and checks all pairs that have one of the seed tags it has been assigned. After all threads have finished, the results are forwarded to the Shift Detection operator. This operator computes the score for each topic executing the Algorithm 2 described in detail in Sections 4.3.3 and 4.3.5.

4.4.1 Diversification

Our main approach ends with the Shift Detection operator. However, in order to increase the users satisfaction, we have added one additional operator, the

Algorithm 4: Update topics information

```
Input: Set of triples $\mathcal{TP} = \{(tm_1, d_1, s_1), (tm_2, d_2, s_2), \ldots, (tm_n, d_n, s_n)\}$,
    Set of tags $\mathcal{TG} = \{t_1, t_2, \ldots, t_k\}$,
    Set of tuples
    $\mathcal{PR}_{old} = \{(s_1, \mathcal{POP}^{s_1}, \mathcal{CR}^{s_1}, \mathcal{CR}^{s_1}_{pr}), \ldots, (s_l, \mathcal{POP}^{s_l}, \mathcal{CR}^{s_l}, \mathcal{CR}^{s_l}_{pr})\}$,
    Integer $min$
Result: Set of tuples $\mathcal{PR}_{new} =$
        $\{(s_1, \mathcal{POP}^{s_1}, \mathcal{CR}^{s_1}, \mathcal{CR}^{s_1}_{pr}), \ldots, (s_l, \mathcal{POP}^{s_l}, \mathcal{CR}^{s_l}, \mathcal{CR}^{s_l}_{pr})\}$
1  $\mathcal{PR}_{new} = \{\}$
2  $\mathcal{NEW} = FindNewTopics(\mathcal{TP}, \mathcal{TG}, min)$
   /* Check each topic found in the current window            */
3  foreach $(s_i, pop, jac) \in \mathcal{NEW}$ do
4      if $\nexists(s_j, \mathcal{POP}^{s_j}, \mathcal{CR}^{s_j}, \mathcal{CR}^{s_j}_{pr}) \in \mathcal{PR}_{old} : s_j = s_i$ then
           /* Set previous values to 0                           */
5          $(pop^{s_i}_k, cr^{s_i}_k, cr^{s_i}_{pr_k}) = (0, 0.0, 0.0), \ \forall k, 0 < k < \rho$
6      else
           /* Remove oldest values                               */
7          $\mathcal{PR}_{old} = \mathcal{PR}_{old} \setminus (s_i, \mathcal{POP}^{s_i}, \mathcal{CR}^{s_i}, \mathcal{CR}^{s_i}_{pr})$
8          $(pop^{s_i}_k, cr^{s_i}_k, cr^{s_i}_{pr_k}) = (pop^{s_i}_{k+1}, cr^{s_i}_{k+1}, cr^{s_i}_{pr_{k+1}}), \ \forall k, 0 < k < \rho$
9      end if
10     $(pop^{s_i}_\rho, cr^{s_i}_\rho, cr^{s_i}_{pr_\rho}) = (pop, jac \times \frac{pop}{|\mathcal{TP}|}, 0.0)$ // Set current values
11     $\mathcal{PR}_{new} = \mathcal{PR}_{new} \cup (s_i, \mathcal{POP}^{s_i}, \mathcal{CR}^{s_i}, \mathcal{CR}^{s_i}_{pr})$
12 end foreach
   /* Check each old topic not found in the current window      */
13 foreach $(s_i, \mathcal{POP}^{s_i}, \mathcal{CR}^{s_i}, \mathcal{CR}^{s_i}_{pr}) \in \mathcal{PR}_{old}$ do
14     if $\exists j \in [2, \rho] : pop^{s_i}_j > 0$ then
15         $(pop^{s_i}_k, cr^{s_i}_k, cr^{s_i}_{pr_k}) = (pop^{s_i}_{k+1}, cr^{s_i}_{k+1}, cr^{s_i}_{pr_{k+1}}), \ \forall k, 0 < k < \rho$
16         $(pop^{s_i}_\rho, cr^{s_i}_\rho, cr^{s_i}_{pr_\rho}) = (0, 0.0, 0.0)$
17         $\mathcal{PR}_{new} = \mathcal{PR}_{new} \cup (s_i, \mathcal{POP}^{s_i}, \mathcal{CR}^{s_i}, \mathcal{CR}^{s_i}_{pr})$
18     end if
19 end foreach
20 return $\mathcal{PR}_{new}$
```

Diversification operator. The *Diversification* operator gets the emergent topics identified by the *Shift Detection* operator and creates groups of them that refer to the same event.

Definition 2. *Two topics refer to the same event if their corresponding tagsets co-exist in 80% of the documents.*

Grouping the topics referring to the same event is necessary to avoid having in the output very many similar results. The description of a topic after the grouping is the union of the tags of all the topics placed in the group.

Additional Operators

The data stream design principles followed in our approach makes the addition of new operators a straightforward procedure. For example, in addition to the *Diversification* operator, a *Personalisation* operator could be also added. The *Personalisation* operator can use standard IR techniques such as Language Models or methods based on tf×idf scores [MRS09] to select from the set of the results only those that satisfy the users preferences by computing scores for topics based on the scores of the associated documents.

User Interface

The Web-based user interface of enBlogue provides real-time monitoring and user notifications in a push-based manner (i.e. without the user having to continuously poll the server for updates on emergent topic rankings). This has been implemented using AJAX technology, more specifically, the push-based variant offered by the open-source *Ajax Push Engine (APE)*[1]. APE includes a Javascript framework for real-time data streaming to Web browsers, without any installations on the client side.

4.5 Implementation of an Alternative Approach

We compare our approach against TwitterMonitor, a work of Mathioudakis and Koudas [MK10] (see also Section 3.2). The general idea of it is summarised in the following two steps:

1. Find bursty tags
2. Group bursty tags in disjoint sets

Grouping bursty keywords to topics is very similar to our approach. However, creating disjoint sets of tags is very restrictive. To our experience, multiple events related with the same keyword may be concurrently emergent. TwitterMonitor does not consider this possibility. Additionally, Mathioudakis et al. assume that traditional news portals have already reported on the events identified by TwitterMonitor and try to link on them. This may be true for really big events but our perspective is that users, especially on Twitter, report on events before they become popular enough to be addressed by traditional news media.

[MK10] is a demonstration paper and provides limited information regarding the proposed approach. We tried to use the main principles on which they based their approach and in case we could not decide, based on the authors description, we took decisions similar to those we took in our approach.

We find bursty tags using Algorithm 5. The process is very similar to the one followed in our approach when detecting shifts (Algorithm 4), but instead of using tagsets, Algorithm 5 uses tags and instead of the correlations between tags it uses the popularities of tags. Similarly to our approach, for each tag t_i the algorithm stores its previous ρ popularity values $\mathcal{POP}^{t_1}$ and its previous ρ predicted popularity values $\mathcal{POP}^{t_1}_{pr}$.

[1] http://www.ape-project.org/

Algorithm 5: Find Bursty Tags in TwitterMonitor

```
Input: Set of triples $\mathcal{TP} = \{(tm_1, d_1, s_1), (tm_2, d_2, s_2), \ldots, (tm_n, d_n, s_n)\}$,
    Set of tuples $\mathcal{PR} = \{(t_1, \mathcal{POP}^{t_1}, \mathcal{POP}^{t_1}_{pr}), \ldots, (t_l, \mathcal{POP}^{t_l}, \mathcal{POP}^{t_l}_{pr})\}$,
    Integer $min$
Result: Set of bursty tags
        $\mathcal{ST} = \{(t_1, score^{t_1}), (t_2, score^{t_2}), \ldots, (t_m, score^{t_m})\}$
1  $\mathcal{PR}_{new} = \{\}$
2  $\mathcal{ST} = \{\}$
   /* Find all tags currently in the window */
3  $\mathcal{TG} = \{\}$ // set of all tags in the current window
4  foreach $(tm_i, d_i, s_i) \in \mathcal{TP}$ do
5      $\mathcal{TG} = \mathcal{TG} \cup s_i$
6  end foreach
   /* Check all tags found in the current window */
7  foreach $t_i \in \mathcal{TG}$ do
8      $T_i = \{d_j | t_i \in s_j \wedge (tm_j, d_j, s_j) \in \mathcal{TP}\}$
9      if $|T_i| \geq min$ then
10         if $\nexists (t_j, \mathcal{POP}^{t_j}, \mathcal{POP}^{t_j}_{pr}) \in \mathcal{PR} : t_j = t_i$ then
               /* Set previous values to 0 */
11             $(pop^{t_i}_k, pop^{t_i}_{pr_k}) = (0.0, 0.0), \ \forall k, 0 < k < \rho$
12         else
               /* Remove oldest values */
13             $\mathcal{PR} = \mathcal{PR} \setminus (t_i, \mathcal{POP}^{t_i}, \mathcal{POP}^{t_i}_{pr})$
14             $(pop^{t_i}_k, pop^{t_i}_{pr_k}) = (pop^{t_i}_{k+1}, pop^{t_i}_{pr_{k+1}}), \ \forall k, 0 < k < \rho$
15         end if
16         $pop^{t_i}_\rho = \frac{|T_i|}{|TP|}$
17         predict $pop^{t_i}_{pr_\rho}$
18         $\mathcal{PR}_{new} = \mathcal{PR}_{new} \cup (t_i, \mathcal{POP}^{t_i}, \mathcal{POP}^{t_i}_{pr})$
19         $score^{t_i} = \frac{pop^{t_i}_\rho - pop^{t_i}_{pr_\rho}}{pop^{t_i}_\rho}$ // Estimate the burst of the tag
20         $\mathcal{ST} = \mathcal{ST} \cup \{(t_i, score^{t_i})\}$ // Store the tag and its score
21     end if
22 end foreach
   /* Check all tags not found in the current window */
23 foreach $(t_i, \mathcal{POP}^{t_i}, \mathcal{POP}^{t_i}_{pr}) \in \mathcal{PR}$ do
24     if $\exists j \in [2, \rho] : pop^{t_i}_j > 0$ then
25         $(pop^{t_i}_k, pop^{t_i}_{pr_k}) = (pop^{t_i}_{k+1}, pop^{t_i}_{pr_{k+1}}), \ \forall k, 0 < k < \rho$
26         $(pop^{t_i}_\rho, pop^{t_i}_{pr_\rho}) = (0.0, 0.0)$
27         $\mathcal{PR}_{new} = \mathcal{PR}_{new} \cup (t_i, \mathcal{POP}^{t_i}, \mathcal{POP}^{t_i}_{pr})$
28         $\mathcal{ST} = \mathcal{ST} \cup \{(t_i, 0.0)\}$ // Store the tag
29     end if
30 end foreach
31 set $\mathcal{PR} = \mathcal{PR}_{new}$
32 return $\mathcal{ST}$
```

For each tag associated during the current implementation with more than *min* documents (Algorithm 5, Line 9), the algorithm checks whether the tag has been found in any of the previous ρ evaluations. In case it has not, it sets zeroes to all previous predicted and computed popularity values (Algorithm 5, Line 11). The real and the predicted popularity values are computed for each tag and a score is assigned to each of them (Algorithm 5, Lines 16 - 20). The score equals the relative error of the predicted popularity compared to the real

Algorithm 6: Group Bursty Tags in TwitterMonitor

```
Input: Set of triples $\mathcal{TP} = \{(tm_1, d_1, s_1), (tm_2, d_2, s_2), \ldots, (tm_n, d_n, s_n)\}$,
    Set of tags $\mathcal{ST} = \{(t_1, score^{t_1}), (t_2, score^{t_2}), \ldots, (t_m, score^{t_m})\}$,
    Integer $min$
Result: Set of top-k topics $\mathcal{SC} = \{(s_1, score, s_2, \ldots, s_k\}$
/* Find all tags currently in the window                    */
$\mathcal{TG} = \{\}$ // set of all tags in the current window
foreach $(tm_i, d_i, s_i) \in \mathcal{TP}$ do
    $\mathcal{TG} = \mathcal{TG} \cup s_i$
end foreach
/* Find pairs of co-existing tags                            */
$\mathcal{NEW} = \{\}$
foreach $t_i \in \mathcal{TG}$ do
    $T_i = \{d_l | t_i \in s_l \wedge (tm_l, d_l, s_l) \in \mathcal{TP}\}$ foreach $t_j \in \mathcal{TG}$ do
        $T_i = \{d_l | t_i \in s_l \wedge (tm_l, d_l, s_l) \in \mathcal{TP}\}$ if $|T_i \cap T_j| \geq min$ && $t_i \neq t_j$
        then
            $\mathcal{NEW} = \mathcal{NEW} \cup \{t_i, t_j\}$
        end if
    end foreach
end foreach
/* create undirected graph G                                 */
foreach $t_i \in \mathcal{TG}$ do
    create node $n_i$ in $G$
end foreach
foreach $\{t_i, t_j\} \in \mathcal{NEW}$ do
    create edge $e_{(i,j)}$ in $G$
end foreach
/* store each connected component                            */
$\mathcal{SC} = \{\}$
foreach Connected Component $cc_i \in G$ do
    $s_i = \{t_j \in cc_i\}$
    $pop^{s_i} = |\bigcup_j (T_j : t_j \in s_i)|$
    set $score^{s_i} = \frac{\sum_j score^{t_j} : t_j \in s_i}{|s_i|} \times \frac{1}{|ln(pop^{s_i})|}$
    $\mathcal{SC} = \mathcal{SC} \cup (s_i, score^{s_i})$
end foreach
sort $\mathcal{SC}$ in descending score order
return top-k of $\mathcal{SC}$
```

popularity, $score^{t_i} = \frac{pop^{t_i} - pop_{pr}^{t_i}}{pop^{t_i}}$. The biggest the error, the more bursty the tag is considered.

To all tags that have been found during any of the previous $\rho - 1$ evaluations but not in the current one, a popularity value of 0 is assigned as their current real popularity value. The score of these tags is also set to zero. (Algorithm 5, Lines 23 - 30).

In the second stage, the bursty tags are grouped into disjoint sets. The disjoint sets are created using Algorithm 6. The input to this algorithm is the set of triples $\mathcal{TP}$ currently in the window and the set $\mathcal{ST}$ containing the bursty tags along with their scores as computed in the previous stage. The bursty tags are organised in a graph G where each tag represents a node and there is an edge between two tags if the tags coexist in at least *min* documents (Algorithm 6, Lines 13 - 18). Each connected component in G is considered to represent a topic. We assign a score to each topic adapting Formula 4.4 to

$$sc_t^{s_i} = \frac{\sum_j score_t^{t_j} : t_j \in s_i}{|s_i|} \times \frac{1}{|ln(pop_t^{s_i})|} \tag{4.5}$$

In our approach, in each evaluation the relative error between the predicted and the computed correlation values of a tagset is considered an indicator of how emergent the topic represented by the tagset is. In TwitterMonitor each tag is considered separately and then tags are grouped into tagsets. Each tag has its own score which indicates how bursty it is. We consider the average score of all tags grouped together to indicate how emergent the topic represented by them is. Similar to Formula 4.4, the final score assigned to each topic is the average score divided by the popularity of the tagset in the current window.

4.6 Experimental Evaluation

We used our prototype implementation to conduct a series of experiments. All measurements were performed on a server with two quad-core 2.4 GHz Intel Xeon processors, 48 GB of RAM, and a 2 TB RAID-5 disk. For repeatability, the datasets were replayed from files that contain the raw data.

4.6.1 Datasets

Blog dataset: We have obtained the ISWCM Spinn3r blog dataset [BJS09] consisting of 44 million blog posts created in the time period from August 1st to October 1st, 2008. Each blog post has a set of categories assigned to it. We use these categories as tags. Examples of categories are *Election 2008* and *Economics*, or *Entertainment* and *Sports*. For the experiments, we used the blog posts from September 2008.

Twitter dataset: We have access to the "fire hose" stream of Twitter, delivering 10% of all Tweets (in general, all status updates). Tweets contain hashtags, such as #egypt and #revolution which we use as tags. For the experiments, we use the tweets from 02.07.2011 to 15.07.2011.

4.6.2 Algorithms

We compare the following algorithms:

enBlogue (eB): This is our approach for emergent topic detection, described in this chapter. For the naming of the different configurations, we will refer to our algorithms by mentioning the number of seed tags used every time. For example our algorithm using all tags as seeds will be referred as eB-100%, our algorithm using 20% of all tags as seeds will be referred as eB-20% and so on. For efficiency reasons, besides using seed tags to restrict the total number of considered tagsets, we have also restricted topics to be represented by pairs of tags. Processing pairs of tags instead of tagsets of higher cardinality allows us to process all input data in real time. As we discussed in the introduction of this chapter, this would affect the emerging topics that we identify. However, the user study we conducted (Section 4.7) shows that our approach is nevertheless able to produce results of higher quality in comparison to results produced by an alternative approach. At the end, users are presented with sets of tags as a result of the Diversification operator.

TwitterMonitor (TM): The approach by Mathioudakis and Koudas [MK10], described in Section 4.5, is our competitor.

The window size for the Twitter dataset is set to $\mathcal{W} = 1h$ and for the Blog dataset to $\mathcal{W} = 6h$. We did not see any influence on the runtimes when varying the number of topics shown to the user and we set it to $k = 20$.

4.6.3 Runtime

We measure and report the runtime cost of the methods. The values reflect the average time spent at each evaluation phase. They do not include the pre-processing costs like named-entity tagging, but they include the post-processing cost of diversification.

In general, the runtime depends on three factors:

1. **initial pairs:** In enBlogue, every seed tag is initially paired with each other tag. This results in the creation of tag pairs which we call initial pairs. Similarly, the initial pairs in TwitterMonitor are created by matching each bursty tag with each other tag. Whether the tags in each pair co-occur in any document is verified in a subsequent step. This means that there might be initial pairs that do not represent topics (only co-occurring tags can represent topics).

   Even for the simple case, where only tag pairs are considered, creating all possible pairs and subsequently eliminate those consisting of non co-existing tags adds a big processing overhead. For this, in a later implementation, when we focus on improving the efficiency of our approach we skip the creation of the initial pairs and we consider, from the beginning, only the sets of co-occurring tags. (cf. Chapter 5).

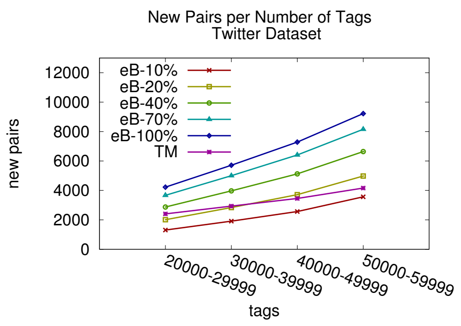
2. **new pairs:** In enBlogue, new pairs are the pairs of tags that represent a topic during the current evaluation. These pairs are identified by *Correlation Computation* operator and are a subset of the initial pairs. All new

pairs are checked during the shift detection procedure (Section 4.3.3) and during the scoring procedure (Sections 4.3.4 and 4.3.5). The number of new pairs depends on the number of initial pairs.

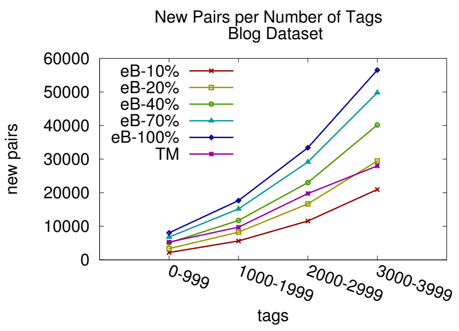
In TwitterMonitor, new pairs are the pairs of bursty tags connected with an edge in the graph G created in the current evaluation.

3. **old pairs:** In enBlogue, old pairs are the pairs that represented a topic during any of the previous $\rho - 1$ evaluations but are not representing a topic in the current evaluation (i.e. the sets of new and old pairs are disjoint). Although not found during the current evaluation, the old pairs are still stored and processed during the scoring procedure (Sections 4.3.4 and 4.3.5). The number of old pairs depends on the number of the new pairs on the previous $\rho - 1$ evaluations.

   In TwitterMonitor, old pairs are the pairs of bursty tags connected with an edge in the graph G created in any of the previous $\rho - 1$ evaluations but not in the current one.



(a) Twitter Dataset

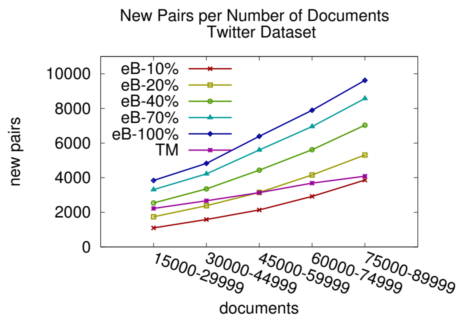


(b) Blog Dataset

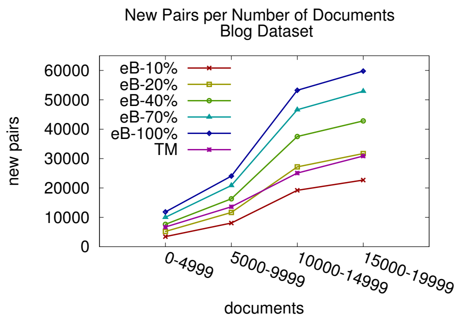
Figure 4.4: Number of new pairs for varying number of total tags

The plots in Figure 4.4 show the dependency of the number of new pairs to the number of total tags. To compute the values, we have grouped the evaluations according to the number of tags that were found in them. We average the values of all evaluations put in the same group. Figure 4.4a shows the results for the Twitter dataset and Figure 4.4b shows the results for the Blog dataset. Comparing the two plots, an interesting observation regarding the characteristics of the two datasets can be made. One can notice that in the Twitter dataset many more tags, compared to the Blog dataset, are found. However, the number of new pairs in the Twitter dataset is much lower than the number of new pairs in the Blog dataset. This is due to the fact that blog posts are longer in size and, thus each blog post contains more tags compared to the number of tags contained in a single tweet. The entities are used as additional tags which means that each blog post has bigger tagsets, i.e. it is associated with more tag pairs. In Twitter tagsets are smaller so, not many pairs are created from each tagset. Additionally, in Twitter many of the co-existing tags are not found in at least *min* documents and thus do not qualify for further consideration (i.e. to become new pairs). This observation reveals the dynamics of the two datasets with Twitter being much more noisy compared to the Blog dataset.

The number of tags used as seeds in the enBlogue variations is a portion of the total tags, and the number of initial pairs are quadratic on the number of seeds. Since the new pairs are a subset of the initial pairs, we expect the number of them to increase with the increase of the percentage of tags used as seeds. Additionally, we expected the number of new pairs to increase with increasing number of tags. For TwitterMonitor, there is no direct analogy between the number of tags and the number of bursty tags. However, we would expect the number of bursty tags to increase as the number of tags increases. The plots in Figure 4.4 confirm our expectations.



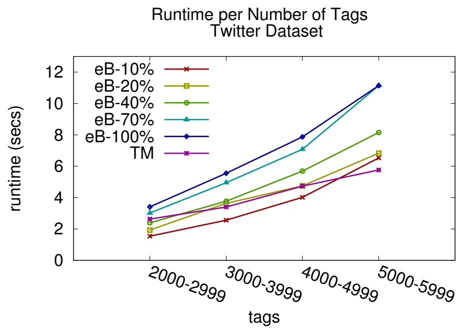
(a) Twitter Dataset



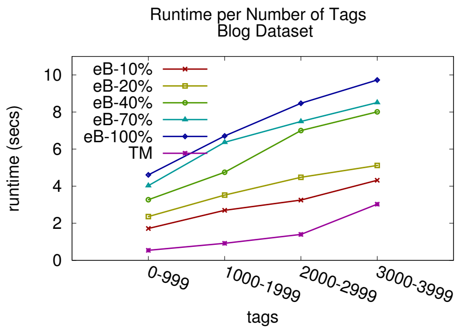
(b) Blog Dataset

Figure 4.5: Number of new pairs for varying number of total documents

The plots in Figure 4.5 show the dependency of the number of new pairs on the number of documents. An increase in the number of documents causes an increase in the number of tags thus, the change in the number of new pairs with increasing number of document shows the same trends to those seen for increasing number of tags.



(a) Twitter Dataset

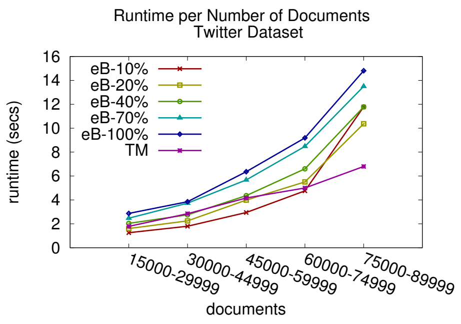


(b) Blog Dataset

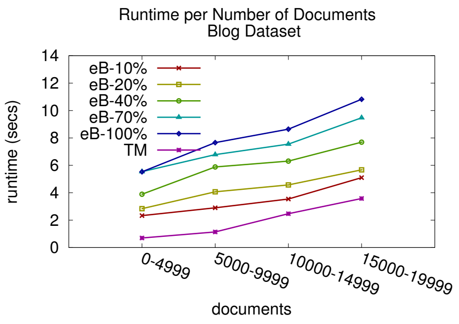
Figure 4.6: Runtimes for varying number of total tags

The plots in Figure 4.6 show how the number of total tags affects the runtime of the algorithms. As the number of tags increases, the number of initial and new pairs increases as well, resulting in higher runtimes. An increase in the number of documents affects the runtimes in the same way (Figure 4.7) since more documents contain more tags. An increase in the number of documents affects the runtimes in one more way. It causes an increase in the size of document-sets associated with each tag. This means that more time is needed to compare the sets of document-ids associated with each tag. Such comparisons are needed when the initial pairs are checked and when the correlations for the new pairs are computed.

During the experiments, we measured that the initial pairs in TwitterMonitor were, on average, 60% of the maximum number of initial pairs. The maximum number of initial pairs is the number of pairs created when 100% of the tags are used as seeds.



(a) Twitter Dataset



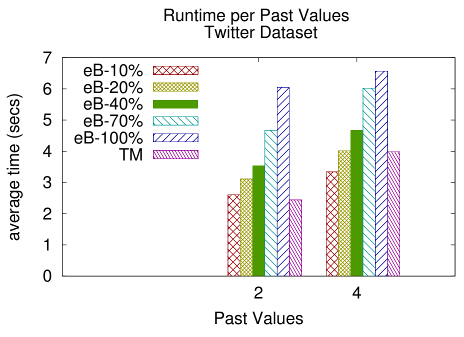
(b) Blog Dataset

Figure 4.7: Runtimes for varying number of total documents

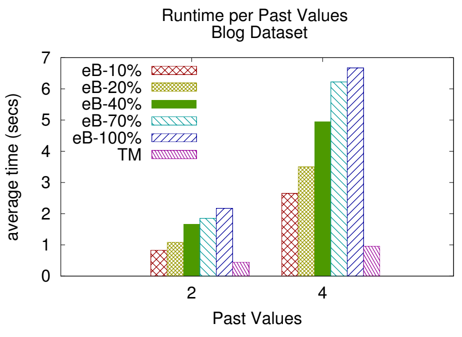
Considering only the number of initial pairs, one would expect TwitterMonitor to be slower than eB-10%, eB-20% and eB-40% in all cases. However, this is not true as can be verified by the plots in Figure 4.6 and Figure 4.7. To the contrary of our expectations, TwitterMonitor in many cases has to process fewer new pairs in each evaluation even from eB-20% (cf. Figure 4.4). This happens because many of the new pairs considered in the enBlogue variations do not consist of tags that are both bursty. There pairs are never consider in TwitterMonitor. TwitterMonitor is faster than enBlogue for one more reason; it does not consider old pairs during the scoring procedure since the notion of carrying topics from the past does not exist in it.

The plots in Figure 4.8 show the effect on the runtime of the number ρ of the previous values used for the prediction of the correlation value, in case of enBlogue, and the popularity value in case of TwitterMonitor. In enBlogue, an increase in ρ causes an increase in the number of the old pairs. In TwitterMonitor, an increase in ρ causes an increase in the number of tags that are found to be bursty. In all cases an increase in ρ causes an increase in the average runtime.

Since the characteristics of the two datasets are different, the change in the ρ affects the runtime in each of them in a different way. In the Twitter dataset an increase in ρ affects more the number of tags found to be bursty while the number of old pairs is affected less. This causes a rapid increase in the runtime of TwitterMonitor. In the Blog dataset, an increase in ρ affects more the number of old pairs while the number of bursty tags is affected less. This causes a greater increase in the runtime of the five variations of the enBlogue compared to the increase caused in the TwitterMonitor.



(a) Twitter Dataset

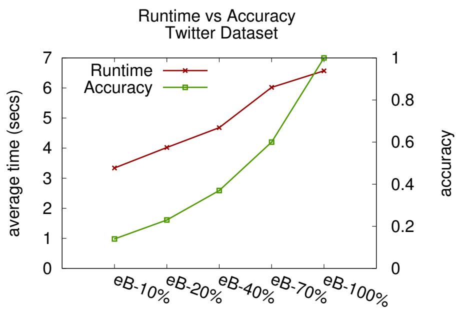


(b) Blog Dataset

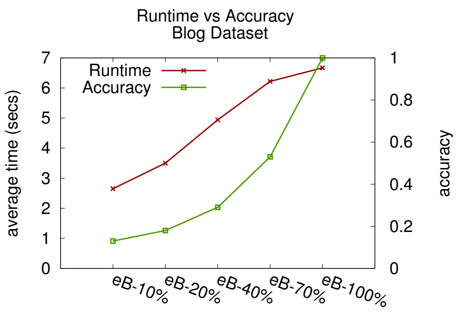
Figure 4.8: Average Runtime per past values used for the prediction of the correlation and popularity values

4.6.4 Runtime and Relative Accuracy

When we run enBlogue using a specific amount of seed tags the resulting emergent topics are only approximated. We consider as baseline the results returned by enBlogue when all tags are used as seeds. The relative accuracy for each enBlogue variation is computed comparing the results returned by the specific variation to the results returned by the baseline.



(a) 2 Past values



(b) 4 Past values

Figure 4.9: Average Runtime and Relative Average Accuracy varying the percentage of seeds

In Figure 4.9, it is shown the effect of the percentage of tags used as seeds on the runtime and the average relative accuracy. In this plot we see that the decrease in runtime is not proportional to the decrease in the percentage of seeds. For example, when using 40% of the tags as seeds the runtime is 4.68 secs (cf. Table 4.2). Decreasing the seeds by 50% (resulting in 20% of tags used as seeds) causes a reduction in runtime of only 14% (to 4.02 secs). This is because the percentage of seeds affects by the same percentage the number of initial pairs but not the number of new and old related pairs.

In Figure 4.9, one can also see that a small decrease in the percentage of seeds (e.g from 70% to 40%) causes a big decrease in the relative accuracy. This is due to the fact that by using a smaller percentage of seed-tags there are some

	Twitter		Blog	
	Runtime	Accuracy	Runtime	Accuracy
eB-10%	3.34	0.14	2.65	0.13
eB-20%	4.02	0.23	3.50	0.18
eB-40%	4.68	0.37	4.94	0.29
eB-70%	6.02	0.60	6.22	0.53
eB-100%	6.57	1.00	6.67	1.00

Table 4.2: Average Runtime and Relative Average Accuracy varying the percentage of seeds

pairs that cannot be found in the results (the pairs that do not have at least one of the selected seed-tags). This lack in pairs affects also the groups of tags, created during the diversification procedure, and is responsible for the reduced relative accuracy.

It is worth mentioning that it is not clear whether the results with 100% seeds are more of user interest or not. Since we have chosen the seeds to be the most popular tags, there is the possibility that the results using a lower percentage of seeds are more interesting than the results with the 100% seeds. However, the percentage of seeds might be of importance since with a small number of seeds there is the danger of having low diversity in the results.

4.7 User Study

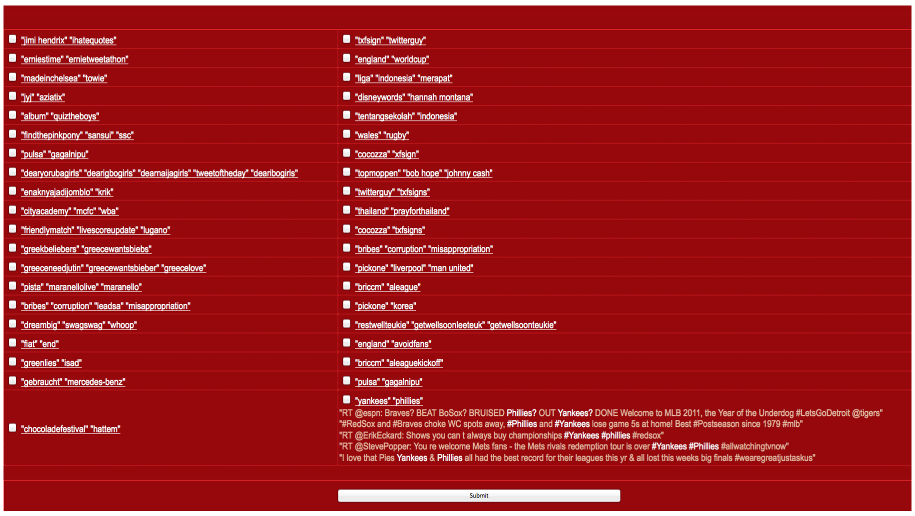
To evaluate the performance of our approach with respect to quality, we have conducted a user study. We set up a website showing 40 emergent topics, 20 topics detected by enBlogue and 20 topics detected by TwitterMonitor. The results were put in random order so that it would not be apparent which approach had detected each of them. Each topic was accompanied by an HTML checkbox. Figure 4.10 shows an example of this page.

We asked colleagues to participate in the study which had the following task description:

Every now and then, check the results published on our website and select the checkbox on the topics that you find to be emergent

We employed the study using live Twitter data. We did not see a viable way to perform such a study on offline (i.e. months or years old) data as it turned out to be very difficult for the users to go mentally back in time and check if a detected event was indeed noteworthy at that time. In particular for events that are not in the same scale with big occasions like the Olympic games, huge hurricanes, US elections, and so on, this is almost impossible.

Since tags alone are sometimes hard to map to a real world event, we provided for each topic a set of tweets containing the tags representing the topic (cf. Figure 4.10). The users could see this sample by clicking on the tagset. For example, on 16.09.2011 12:00 GMT the topic “dolphin Australia” was identi-



"jimi hendrix" "ihatequotes"	"txfsign" "twitterguy"
"erniestime" "ernietweetathon"	"england" "worldcup"
"madeinchelsea" "towie"	"liga" "indonesia" "merapat"
"jyj" "aziatix"	"disneywords" "hannah montana"
"album" "quiztheboys"	"tentangsekolah" "indonesia"
"findthepinkpony" "sansui" "ssc"	"wales" "rugby"
"pulsa" "gagalnipu"	"cocozza" "xfsign"
"dearyorubagirls" "dearigbogirls" "dearnaijagirls" "tweetoftheday" "dearibogirls"	"topmoppen" "bob hope" "johnny cash"
"enaknyajadijomblo" "krik"	"twitterguy" "txfsigns"
"cityacademy" "mcfc" "wba"	"thailand" "prayforthailand"
"friendlymatch" "livescoreupdate" "lugano"	"cocozza" "txfsigns"
"greekbeliebers" "greecewantsbiebs"	"bribes" "corruption" "misappropriation"
"greeceneedjutin" "greecewantsbieber" "greecelove"	"pickone" "liverpool" "man united"
"pista" "maranellolive" "maranello"	"briccm" "aleague"
"bribes" "corruption" "leadsa" "misappropriation"	"pickone" "korea"
"dreambig" "swagswag" "whoop"	"restwellteukie" "getwellsoonleeteuk" "getwellsoonteukie"
"fiat" "end"	"england" "avoidfans"
"greenlies" "isad"	"briccm" "aleaguekickoff"
"gebraucht" "mercedes-benz"	"pulsa" "gagalnipu"
"chocoladefestival" "hattem"	"yankees" "phillies" "RT @espn: Braves? BEAT BoSox? BRUISED **Phillies?** OUT **Yankees?** DONE Welcome to MLB 2011, the Year of the Underdog #LetsGoDetroit @tigers" "#RedSox and #Braves choke WC spots away, **#Phillies** and **#Yankees** lose game 5s at home! Best #Postseason since 1979 #mlb" "RT @ErikEckard: Shows you can t always buy championships **#Yankees #phillies** #redsox" "RT @StevePopper: You re welcome Mets fans - the Mets rivals redemption tour is over **#Yankees #Phillies** #allwatchingtvnow" "I love that Pies **Yankees** & **Phillies** all had the best record for their leagues this yr & all lost this weeks big finals #wearegreatjustaskus"

Figure 4.10: User Study: Sample Tweets

fied. Looking solely on these tags it is hard to understand the specifics of the topic. However, looking at the sample of tweets one could realise that the tagset represented the event of discovering a new species in Australia.

The events we derive from Twitter depend on the events around the world and the interests of the Twitter users. Most of the time, they are of small scale but when big events are happening enBlogue detects and reports them. For instance, on 16.09.2011 12:00 GMT enBlogue detected events such as "Assad Syria" and "Lybia Niger Gadhafi". Nevertheless, a lot of small events are interesting and worth being shown. Interestingly, and as a support of the whole approach, for a lot of the events we discovered there were no media information immediately available, only some minutes/hours later.

The user study was conducted during the last two weeks of September 2011. During this period, we recorded 80 non-redundant evaluations. To identify redundant evaluations, we used the IP address of the user who did the evaluation and the timestamp of the ranking she evaluated. Evaluations from the same IP address regarding the same ranking (identified by its timestamp) were not considered again.

We measure the precision of the results returned by enBlogue and Twitter-Monitor (TM) using the precision@k. A precision value of x at k means that a fraction of x events out of k have been considered to be noteworthy. Users were asked to select noteworthy events out of 20 events presented for each algorithm. We measured the precision at the events reported in the top-1 position from each algorithm, at the events reported in the top-2 positions from each algorithm and so on. The results are shown in Figure 4.11

We observe that enBlogue clearly outperforms TwitterMonitor. For the time points for which the users had evaluated the events in our study, enBlogue identified, on average, 2.5 out of 20 noteworthy events per hour. On the contrary, TwitterMonitor identified on average 0.8 out of 20 events. Note that the 20 reported events were not filtered (except for a simple keyword filter aiming at eliminating porn related Tweets).

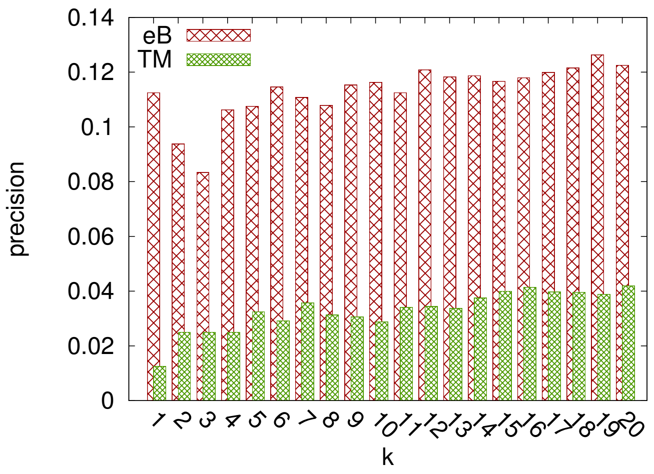


Figure 4.11: Precision@k values

For completeness, we have also calculated the NDCG values, reported in Table 4.3. The measure used to estimate the relevance of a retrieved result is binary, i.e. the result is either relevant or not thus, the ideal ranking is the one having all r relevant results, out of the top-k ones that were retrieved, in the top-r positions.

For both precision@20 and NDCG@20 values we have computed the paired t-test and Fisher's randomised significance test. Fisher's randomised significance test, computed for 100,000 permutations, reported a p-value of 0 for both the precision@20 and the NDCG@20. The paired t-test reported a p-value of 3.5×10^{-13} for NDCG@20 and 6.5×10^{-25} for precision@20.

Table 4.4 shows sample results of the events detected from enBlogue on three consecutive days (28^{th}, 29^{th}, 30^{th}) in September 2011. As we can see, enBlogue discovered quite many interesting events at those days. Including the alliance plans between Microsoft and Samsung, the killing of an Anwar al-Awlaki, a member of Al-Qaeda, by US military forces, the case of Michael Jackson's personal physician Conrad Murray, the Hollywood actor Sean Penn visiting the Tahrir place in Cairo, Egypt, and the scandal of Manchester City's Carlos Tevez, refusing the exchange during a game in the European Soccer Champions League.

4.8 Summary

In this chapter, we presented enBlogue, our approach regarding event detection in social media. We defined a topic to be represented by a set of tags and proposed a measure to estimate the correlation between these tags based on the local and global importance of them. An emergent topic (or event) has been defined to be a topic demonstrating unexpected behaviour, i.e. having a correlation value greater than the correlation value predicted using exponential smoothing. We performed experiments testing our approach using our implemented prototype. We compared enBlogue against TwitterMonitor, an alternative approach for event detection focusing specifically on Twitter. A user study was conducted

	Precision@k		**NDCG@k**	
k	**eB**	**TM**	**eB**	**TM**
1	0.112	0.012	0.112	0.012
2	0.094	0.025	0.094	0.025
3	0.083	0.025	0.089	0.026
4	0.106	0.025	0.115	0.028
5	0.108	0.032	0.123	0.033
6	0.115	0.029	0.137	0.033
7	0.111	0.036	0.151	0.041
8	0.108	0.031	0.160	0.041
9	0.115	0.031	0.177	0.044
10	0.116	0.029	0.191	0.045
11	0.112	0.034	0.198	0.053
12	0.121	0.034	0.220	0.056
13	0.118	0.034	0.229	0.059
14	0.119	0.038	0.239	0.068
15	0.117	0.04	0.246	0.073
16	0.118	0.041	0.258	0.077
17	0.120	0.040	0.270	0.078
18	0.122	0.040	0.285	0.081
19	0.126	0.039	0.300	0.083
20	0.122	0.042	0.304	0.090

Table 4.3: Precision@k results and NDCG@k results achieved in the user study by the competing algorithms.

in order to assess the appeal of the events identified by enBlogue to the users. The results of the study have been very promising and further optimisations, like personalisation, are expected to increase the user perceived satisfaction even more. For efficiency reasons, we restricted topics on tagsets of size 2. In the next chapter, we will describe an implementation that will allow us to withdraw this restriction.

28.09.2011		
{lfc, liverpool, ynwa}	{arshavin, rosicky, sagna}	{orioles, red sox}
{dana, danafacts}	{microsoft, samsung}	{europe, soteu}
{intel, samsung}	{fifa, tevezexcuses}	{detroit, tedx}
{bieberfacts, justin bieber}	{bahrain, twitition, u.s. ambassador}	
{messi, fcblive, mascherano, barca, puyol, abidal, xavi}		
{anelka, cfc, ivanovic, kalou, drogba, romeu}		
{nadarkhani, iran, irani, yousef}		

29.09.2011	
{redsox, shocked, stunned, seasonover}	{ownacolour, unicef}
{nadarkhani, iran, yousef}	{fact, healthcare reform}
{enoughisenough, occupysf, occupywallstreet}	
{bahrain, egypt, usa}	{carlos tevez, manchester city}
{bahrain, syria}	{conrad murray, michael jackson}
{bologna, occupywallstreet, ows}	
{real madrid, kaka, realmadrid}	
{bologna, occupywallstreet}	{celtic, udinese}
{nationalcoffeeday, peetscoffee}	

30.09.2011		
{in america, occupywallstreet}		{bahrain, u.s.}
{libertysquare, armenia, opposition, rally, yerevan}		
{redsox, terry francona}	{arsenal, spurs}	{motegi, motogp}
{derby, liverpool}		{anonymous, antisec}
{rugby, samoa, southafrica}	{egypt, noscaf}	{sean penn, tahrir}
{israel, awlaki, alqaeda, yemen}		{assad, syria}
{awlaki, obama}	{manutd, mufc}	{boston, terry francona}

Table 4.4: Sample of the events detected by enBlogue and marked as relevant by at least one of the user study participants

Chapter 5

Distributed Jaccard Computation

In Chapter 4, we described our approach on identifying emergent topics over dynamic data streams as Twitter. We defined topics to be represented by tagsets, but, for efficiency reasons, we restricted ourselves to tag pairs. Additionally, we defined seed tags to be the top-k most popular tags and restricted considered tagsets to those having at least one seed tag. In this chapter, we present an approach that allows to withdraw all the above restrictions and to compute the Jaccard coefficient for a large number of sets of co-occurring tags efficiently. The general idea is to distribute the computations of the coefficients to multiple machines (nodes). Each node is assigned a subset of the total tags and computes, in parallel with the other nodes, the Jaccard coefficients for its assigned set of tagsets. The Jaccard coefficient, presented in detail in Section 2.4.1, is the main ingredient of the measure we introduced in Chapter 4, used to asses the strength of the correlation between the tags in a tagset. However, our approach is not limited to the Jaccard coefficient. It can distributively compute any measure that is estimated using set operations.

The initial idea of this chapter has been published at the ACM SIGMOD Workshop on Databases and Social Networks (DBSocial 2013) [AM13a]. The extended version of it has been published at the ACM International Conference on Management of Data (SIGMOD 2014) [AM14].

Consider the following set of tagsets. Each tagset is accompanied by the number of documents associated with it.

- s_1={`#beer, #pizza, #soccer`}: 32
- s_2={`#munich, #bavaria, #soccer`}: 17
- s_3={`#bavaria, #munich`}: 28
- s_4={`#beer, #pizza`}: 41
- s_5={`#beach, #sunny`}: 19
- s_6={`#friday, #sunny`}: 23
- s_7={`#friday, #sunny, #beach`}: 5

We want to assign each tagset s_j to one of the available machines. Each machine (or node) will be responsible to compute the Jaccard coefficients for the tagset it has been assigned. To do that, each node n_i should compute two counters for each tagset s_j it has been assigned:

- one *intersection* counter carrying the number of documents annotated with *ALL* tags in s_j
- one *union* counter carrying the number of documents annotated with *ANY* of the tags in s_j

To reduce the total number of counters that should be computed we use the Inclusion-Exclusion principle. According to the Inclusion-Exclusion principle, presented in detail in Section 2.4.2, the counter for the union of a tagset can be computed using the counters for the intersections of all subsets derived from the tagset. For example, for the set s_1={`#beer, #pizza, #soccer`} the counter for the union can be computed as:

$$\begin{aligned}|T_{\#beer} \cup T_{\#pizza} \cup T_{\#soccer}| &= |T_{\#beer} \cap T_{\#pizza} \cap T_{\#soccer}| \\ &- |T_{\#beer} \cap T_{\#pizza}| \\ &- |T_{\#beer} \cap T_{\#soccer}| \\ &- |T_{\#pizza} \cap T_{\#soccer}| \\ &+ |T_{\#beer}| + |T_{\#soccer}| + |T_{\#pizza}|\end{aligned}$$

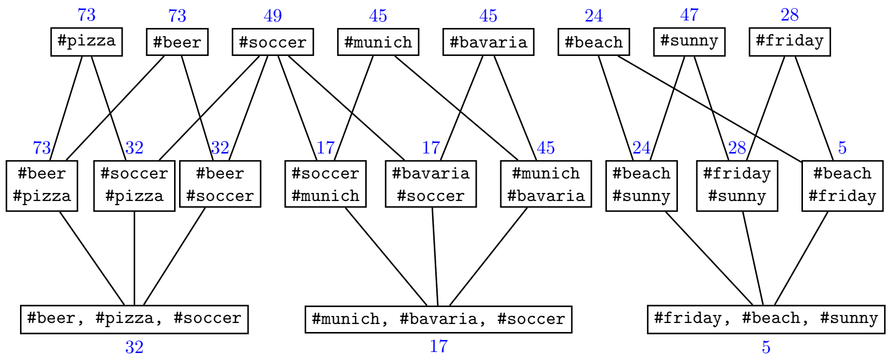


Figure 5.1: Example of a tagsets lattice

Assigning the tagsets to the nodes is not trivial. The lattice of Figure 5.1 shows the dependencies derived from the previously introduced set of tagsets. Each vertex in the lattice corresponds to a set of co-occurring tags (or tagset). The number in each vertex depicts the number of documents associated with the corresponding tagset. There is a path from a vertex n_i to a vertex n_j when the computation of the Jaccard coefficient of the vertex n_i, lying in the lower level, requires the counter of the vertex n_j, lying in any of the above levels, i.e. there is a dependency between the two vertices n_i and n_j and the corresponding tagsets. Considering again the tagset s_1={`#beer, #pizza, #soccer`}, in order to compute its Jaccard coefficient using the Inclusion-Exclusion principle the counters for the tagsets {#beer, #soccer}, {#beer, #pizza}, {#pizza, #soccer}, {#beer}, {#pizza} and {#soccer} are required (see above).

One possible way to partition the tagsets to the nodes is according to the dependencies they have with each other. Alternatively, the tagsets could be partitioned according to the number of documents associated with each of them.

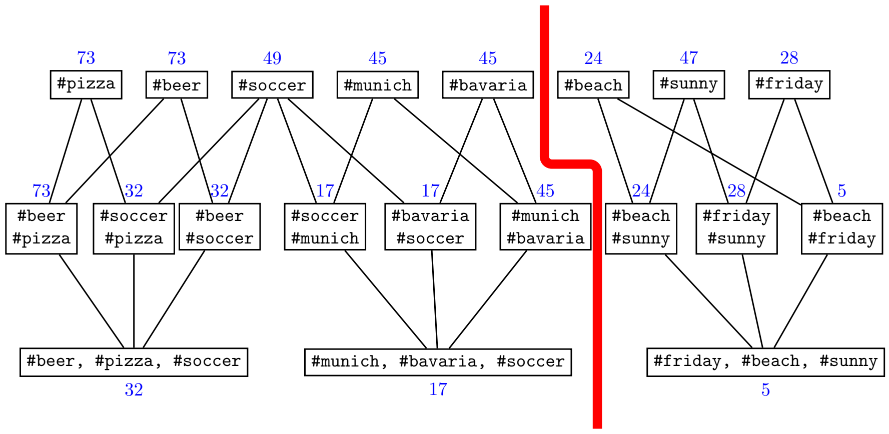


Figure 5.2: Partitioning according to dependencies

Partitioning the tagsets according to the dependencies results in two independent sets of tagsets as shown in Figure 5.2. Assuming the relative number of documents associated with each tagset remains the same through time, the partitioning shown in Figure 5.2 creates two unbalanced partitions with respect to the number of documents each node should process. The node assigned the left partition will have to process 118 documents and the node assigned the right partition will have to process 47 documents, i.e. the load distribution is 72% and 28% respectively.

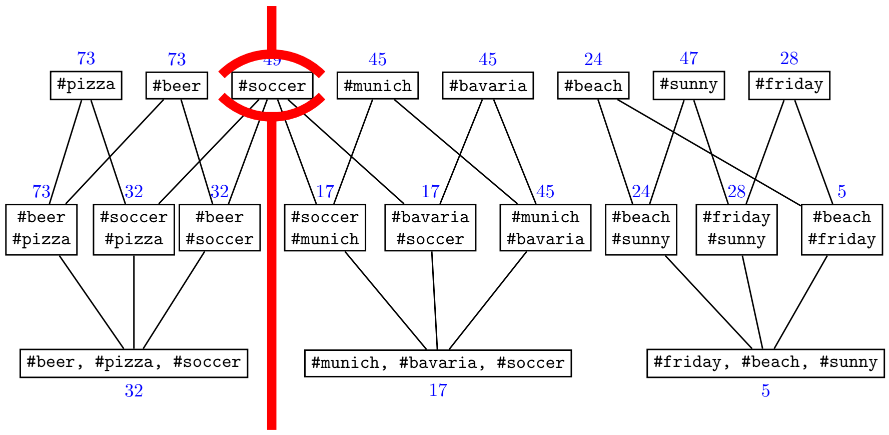


Figure 5.3: Partitioning according to load

Partitioning the tagsets according to the number of documents related to each tagset creates more balanced partitions with respect to the number of documents that need to be processed by each node. However, balancing the load between the nodes may result in the same tagsets assigned to multiple nodes. An example of such a partition is shown in Figure 5.3. The load in the two partitions is relatively balanced with the node assigned the left partition processing 90 documents and the node assigned the right partition processing 124 documents i.e. the load distribution is 42% and 58% respectively. At the same time, the tagset {`#soccer`} is assigned to both partitions since both partitions have tagsets that dependent on the tagset {`#soccer`}. This means that all documents associated with the tag `#soccer` should be forwarded to both nodes causing a communication overhead in the system.

In this chapter, we propose a family of methods that:

- Assign each set of co-occurring tags to at least one node.
- Assign the tagsets to the nodes in a way that most tagsets are assigned to at most one node.
- Assign the tagsets to the nodes in a way that balances the load seen by each node.

$\mathcal{TG}$, 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
$\mathcal{W}, w$	The size of the sliding window, The units the windows slides every time

Table 5.1: Notations used in the chapter

5.1 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 $s_i = \{t_1, t_2, \ldots\}$ from a global set of tags $\mathcal{TG}$ and it is represented by a triple of the form:

$$(tm_i, d_i, s_i)$$

The timestamp tm_i reflects the creation time of the document, the d_i is a unique identifier assigned to the document and the s_i is a set of tags representing the topic of the document. The tags may have been assigned to the document explicitly by its creator or may have been extracted from the body of the document using an annotation mechanism.

We have at our disposal k machines that are independent from each other. We want to assign the tags in $\mathcal{TG}$ to the k machines (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. At regular time intervals, the machines compute and report the Jaccard coefficients for all sets of co-occurring tags present in the set of tags they have been assigned.

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 the users, and the relative popularity of the assigned tagsets changes. These changes deteriorate the quality of the partitions in terms of balanced processing load and low communication overhead. This should be detected and new partitions should be created that fit the current data.

5.2 Approach

We propose a practical online solution that:

(i) computes partitions based on the recently observed tags and their co-occurrences.

(ii) introduces updates to computed partitions to account for new tags and new tag co-occurrences.

(iii) monitors the quality of the partitions to trigger their re-computation.

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 information flow among the tree operators is shown in Figure 5.4

5.2.1 Calculator Operator

Each Calculator C_i is assigned a subset of tags and creates one counter for each set of co-occurring tags in this subset. C_i receives all documents with tags in

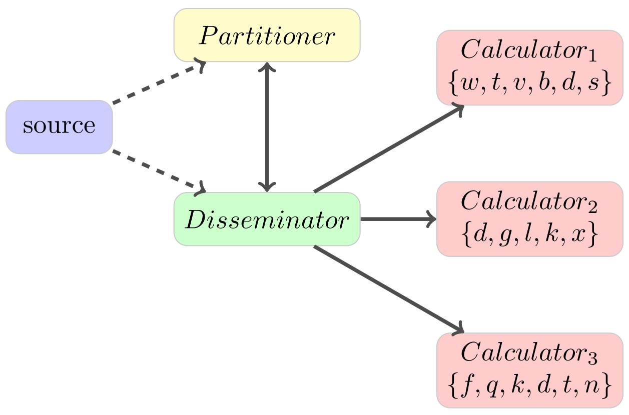

Partitioner
source
Disseminator
$Calculator_1$ $\{w, t, v, b, d, s\}$
$Calculator_2$ $\{d, g, l, k, x\}$
$Calculator_3$ $\{f, q, k, d, t, n\}$

Figure 5.4: Information flow

its assigned subset and for each received document it updates the appropriate counters. At any time, the counter for a tagset s_j reflects the number of documents associated with the tagset s_j that C_i is aware of.

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. [EK13]) and (ii) not all possible tag combinations are used.

5.2.2 Partitioner Operator

The Partitioner operator receives the incoming documents and decides how to partition the tags. The partitioning algorithm, used by the Partitioner to split the tags to the Calculators, should take care to split them in such a way 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 5.3 discusses in detail the algorithms we use for the partitioning.

5.2.3 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 then 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 [HM03] 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 $\{C_j | C_j \textit{ is counting tag } t_i\}$ 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 of the document is performed to obtain the Calculators that should receive this document.

5.3 Partitioning Algorithms

In this section we present a set of algorithms to partition the tags in the various nodes. Any partitioning algorithm 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 needs, i.e. minimise the number of messages sent from the Disseminator to the Calculators.

This problem of partitioning the tags to the nodes can be modelled as a *graph partitioning problem.* Graph G has one vertex v_i for each set of co-occurring tags s_i. 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. Considering the set of co-occurring tags introduced at the beginning of the chapter, the graph derived from it is depicted in Figure 5.5.

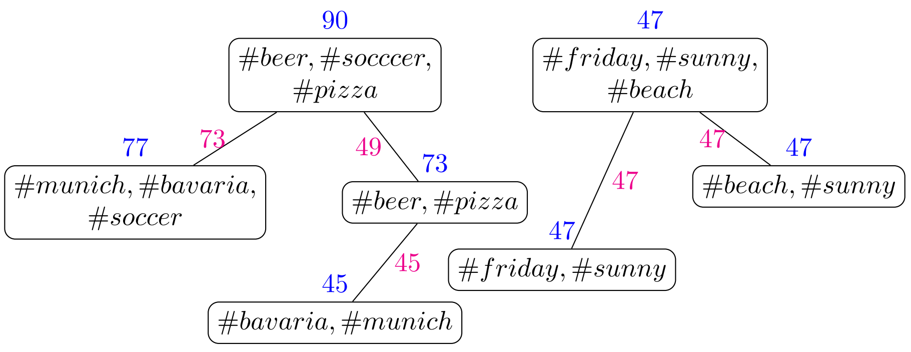


Figure 5.5: Tagsets graph example

For the kind of data we consider in this work, i.e tagsets of low cardinality occurring in social media messages like Twitter 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 (Section 5.3.1). Additionally, we make use of algorithms based on the Set Cover [CLRS09] problem (Section 5.3.2).

5.3.1 Disjoint Sets Algorithm

In social media like Twitter, users annotate their messages with tags that reflect the topics discussed on them. Tags describing the same topic are found in the same documents, creating sets of tags that are disjoint from each other. Organising the tags in a graph with each tag creating a vertex and an edge between any two co-occurring tags, results in a graph with multiple connected components. Algorithm 7 is based on this observation. Initially, all sets of tags that form connected components, i.e. *disjoint sets* of tags, are identified (Algorithm 7, Lines 3-7). The identified disjoint sets are subsequently merged into k sets/partitions, where k is the number of available machines (Algorithm 7, Lines 11-25).

Algorithm 7: Disjoint Sets Algorithm

```
   Input: Set of sets of documents $\mathcal{T} = \{T_1, T_2, \ldots\}$
       Set of tags $\mathcal{TG} = \{t_1, t_2, \ldots\}$
       Integer $k$
   Result: Set $\mathcal{PR}$ of $k$ tag-partitions
 1 $\mathcal{DS} = \{\}$
 2 $\mathcal{PR} = \{\}$
   /* Find Disjoint Sets of Tags */
 3 while $\mathcal{TG} \neq \emptyset$ do
 4     Find $ds_j = \bigcup_i t_i$ disjoint set of tags
 5     $\mathcal{DS} = \mathcal{DS} \cup \{ds_j\}$
 6     $\mathcal{TG} = \mathcal{TG} \setminus ds_j$
 7 end while
 8 foreach $ds_j \in \mathcal{DS}$ do
 9     $l_j = |\bigcup_i T_i|, t_i \in ds_j$
10 end foreach
   /* Merge them into $k$ partitions */
11 while $k > 0$ and $\mathcal{DS} \neq \emptyset$ do
12     $ds_i = argmax_{ds_j} l_j$
13     $pr_k = ds_i$
14     $l_{pr_k} = l_i$
15     $\mathcal{PR} = \mathcal{PR} \cup \{pr_k\}$
16     $\mathcal{DS} = \mathcal{DS} \setminus \{ds_i\}$
17     $k = k - 1$
18 end while
19 while $\mathcal{DS} \neq \emptyset$ do
20     $ds_j = argmax_{ds_m} l_m$
21     $pr_i = argmin_{pr_m} l_m$
22     $pr_i = pr_i \cup ds_j$
23     $l_i = l_i + l_j$
24     $\mathcal{DS} = \mathcal{DS} \setminus \{ds_j\}$
25 end while
26 return $\mathcal{PR}$
```

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 7, Line 9). As long as there are more disjoint sets to be assigned, the one with the biggest load is selected (Algorithm 7, Line 20) and assigned to the partition with the lowest current load (Algorithm 7, Line 21), attempting to balance the load in the partitions.

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 and 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 partition that is very big, the 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. In this thesis, we investigate the performance of the individual algorithms.

5.3.2 Set-Cover Based Algorithms

The algorithms presented in this section treat the creation of k partitions as a *Set Cover Problem* over the input tagsets. The general Set Cover Problem 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* [KMN99] 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 consider a hard limit on the budget. Instead, we try to minimise the final cost of the cover.

Algorithm 8 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 8, Line 3) is added to the set of selected tagsets.

The cost c_i of each set s_i is defined differently depending on whether the algorithm optimises for the communication overhead or the processing load.

- **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.

- **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 sets of 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 m^{th} iteration of

Algorithm 8: Set-Cover Based Algorithm

```
Input: Set of sets of documents $\mathcal{T} = \{T_1, T_2, \ldots\}$
    Set of co-occurring tag-sets $\mathcal{S} = \{s_1, s_2, \ldots\}$
    Integer $k$
Result: Set $\mathcal{PR}$ of $k$ tag-partitions
1  $\mathcal{CV} = \{\}$ // Set of already covered tags
2  while $k > 0$ and $\mathcal{S} \neq \emptyset$ do
3      $s_i = argmin_{s_j} c_j$ and $argmax_{s_j} |s_j \setminus CV|$
4      $pr_k = s_i$
5      $\mathcal{PR} = \mathcal{PR} \cup \{pr_k\}$
6      $\mathcal{S} = \mathcal{S} \setminus \{s_i\}$
7      $\mathcal{CV} = \mathcal{CV} \cup s_i$
8      $k = k - 1$
9  end while
10 Assign Remaining Tag-sets using Algorithm 9 or Algorithm 10
11 return $\mathcal{PR}$
```

the algorithm, C contains $m - 1$ sets and we will select the m^{th} 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 defined as $|pl_{op} - pl_n|$.

To the initial k partitions created using Algorithm 8 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 (Algorithm 9), in each iteration, the set with the most not covered tags, having the least total tags is selected (Algorithm 9, Line 3). The selected tagset is added to the partition sharing with it the most tags having the least load (Algorithm 9, Line 4).

Algorithm 9: Set-Cover Based Algorithm - Focusing on Network Communication

```
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      $s_i = argmax_{s_j} |s_j \setminus CV|$ and $argmin_{s_j} |s_j|$
4      $pr_i = argmax_{pr_j} |s_i \cap pr_j|$ and $argmin_{pr_j} \sum_{s_k \in pr_j} l_k$
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 $\mathcal{PR}$
```

When optimising for the load distribution (Algorithm 10), in each iteration, the set with the most load, having the least already covered tags is selected (Algorithm 10, Line 3). The selected tagset is added to the partition having the least load sharing the most tags with the selected tagset (Algorithm 10, Line 4).

Algorithm 10: Set-Cover Based Algorithm - Focusing on Load

```
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 $\mathcal{PR}$
```

We introduced the idea of dividing Jaccard computations in multiple nodes in [AM13a]. In that work, we treat the selection of the k first tagsets as a *Maximum Coverage Problem without budget*. The assignment of the remaining tagsets to the partitions is performed using as a criterion the number of tags shared among the tagset and the partition (Algorithm 11). In each iteration, a random set is selected (Algorithm 11, Line 2) and added to the partition with which it shares the most tags (Algorithm 11, Line 3) We compare experimentally this initial algorithm with the new ones. For the selection of the k first tagsets used to initialise the k partitions, we use Algorithm 8 setting the cost of each tagset to zero.

Algorithm 11: Set-Cover Based Algorithm - Initial

```
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 $\mathcal{PR}$
```

5.4 Theoretical Expectations

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 for equally sized tag partitions.

5.4.1 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 [ER60]. 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 [ER60] 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 is 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).

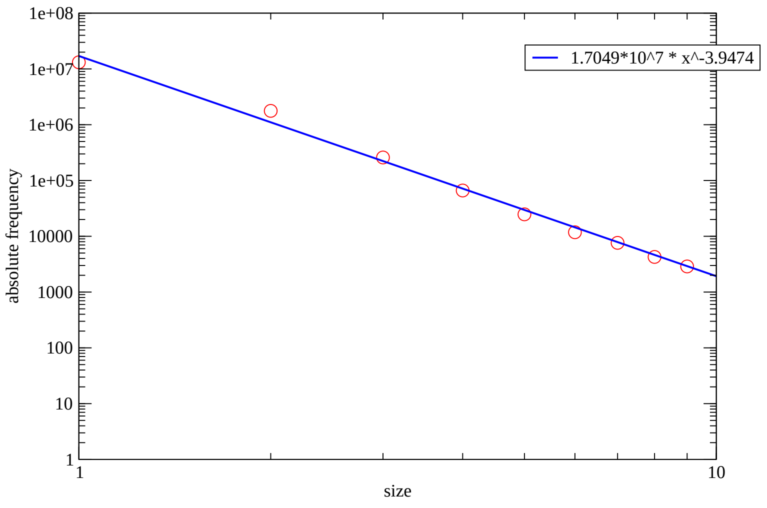


Figure 5.6: Tagsets size distribution (log-log scale)

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 the randomly selected day of the 28^{th} of January 2012. Figure 5.6 depicts the measured number of tweets for each size of tagset in log-log scale. The results show that the number of tags used to annotated the tweets follows Zipf's law, i.e. no tags at all is the most popular case, one tag the second most popular case, and so on, with skew parameter $s = 4$.

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. Tweets that are annotated with the same set of tags are viewed, in our setting, as duplicate tweets and are not considered since they do not add any additional edges.

In the 15 million tweets received on the 28^{th} of January 2012, we found around 700,000 distinct ones containing about 600,000 distinct tags, i.e. each distinct tweet had more than one distinct tags. The 700,000 distinct tweets correspond to a 10% sample of tweets. Taking the full stream under consideration, we assume a total of 7 million distinct tweets in one day or approximately, 4860 distinct tweets every minute. We do not consider the best case in which the distinct tags increase 10 times in the full stream, since we believe this is not realistic. On the contrary, we consider more restrictive cases. For example, in a 5 minutes window, we assume that half of the tweets have 1 distinct tag and the other half of the tweets share tags with the first half of the tweets. This means that, in 24300 tweets there are 12150 distinct tags. Assuming $m_{max} = 8$ tags we get $np = 0.61$. Considering a 10 minutes window, an increase in the total number of tags is expected, but at the same time we expect a decrease in the distinct tags per tweet. For that, we assume that one third of the tweets have one distinct tag. This means that, in 48600 tweets there are 16200 tags. Assuming again $m_{max} = 8$ tags, we get in this case $np = 0.92$, which is still smaller than 1.

The above model computes the number of edges (i.e. co-occurring tag pairs) in the worst case in which tags are randomly and independently assigned to the tweets. However, users do not randomly annotate their tweets. They rather select tags from topic specific vocabularies that reflect the semantics of the published content. In the sample of tweets we obtained on the 28^{th} of January 2012, we measured about 560,000 distinct pairs of tags. Extrapolating it to the whole stream we get about 5.6 million tag pairs in the whole day. Assuming a uniform distribution of pairs during the day, we get about 39,000 tag pairs in a time span of 10 minutes. This gives $np = 0.13$ for $m_{max} = 8$ instead of $np = 0.92$ computed using the theoretic model. Of course, $np = 0.13$ is very optimistic, since tagsets are not expected to increase proportionally in the whole stream, but, in any case, the real value of np is expected to be much lower than 0.92.

Consequently, as long as users select tags from exclusively topic-specific disjoint vocabularies, graph G degenerates to a set of connected components as many as the number of 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, as described by Erdős and Rényi's model. 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.

5.4.2 Communication

We consider k partitions created over n tagsets. Each tagsets s_i has m tags randomly selected from a vocabulary of size v. We further assume that each partitions contains $\frac{n}{k}$ randomly selected tagsets. We are interested in deriving the expected number of partitions that contains any of the tags found in a tagset s_i. This will give us the expected degree of communication between the Disseminator and the Calculators. Obviously, the setting we assume is suboptimal since tags are considered independent and their co-occurrences in tagsets are not considered.

The probability that a partition pr_j has common tags with a tagset s_i is equal to the probability that s_i overlaps with at least one of the $\frac{n}{k}$ tagsets s_j assigned to pr_j.

$$P[pr_j \cap s_i \neq \emptyset] = 1 - (P[s_j \cap s_i = \emptyset])^{\frac{n}{k}}$$

The probability that, given a set s_i, we have another set s_j that does not overlap with it is computed as:

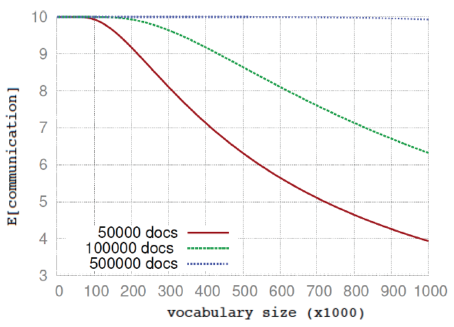
$$P[s_j \cap s_i = \emptyset] = \frac{\binom{v-m}{m}}{\binom{v}{m}}$$

The expected number of partitions sharing tags with a tagset s_i, which gives as the expected communication, is then computed as:

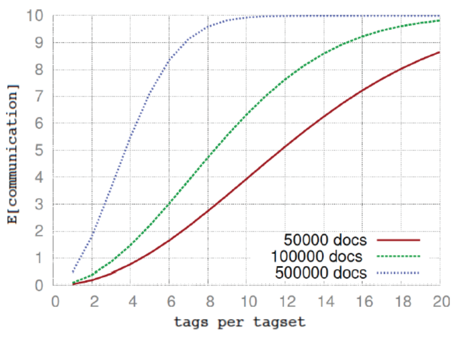
$$E[communication] = k \times \left(1 - \left[\frac{\binom{v-m}{m}}{\binom{v}{m}} \right]^{\frac{n}{k}} \right)$$

Figure 5.7 depicts the change in communication as computed using the above model. Considering the same number of documents, increasing the number of tags in the vocabulary, for given number of tags per tagset, decreases the probability of having two tagsets sharing tags (Figure 5.7a). Increasing the number of tags per tagset, for given vocabulary size, increases the probability of having two tagsets sharing tags (Figure 5.7b). In both cases, increasing the number of considered documents, increases the probability of having two tagsets sharing tags. This means that, for datasets with the same vocabulary and set size, increasing the number of documents causes an increase in the communication.

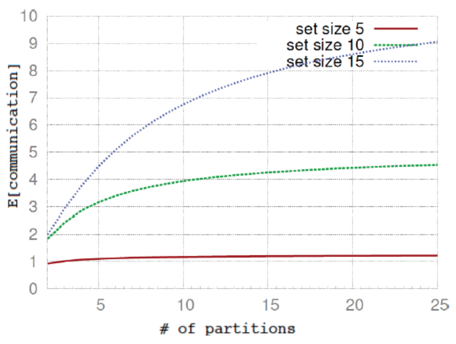
Assuming the same number of tags in each tagset, increasing the number of partitions, for given number of documents, increases the probability of splitting two tagsets that have common tags. This results in an increase in communication (Figure 5.7c). Increasing the number of documents, for a given number of partitions, increases the probability to find two tagset with common tags. This, again, results in an increase in communication (Figure 5.7d). In both cases, considering bigger sets of tags increases the communication even more since it increases the probability of assigning the same tag to more than one tagsets.



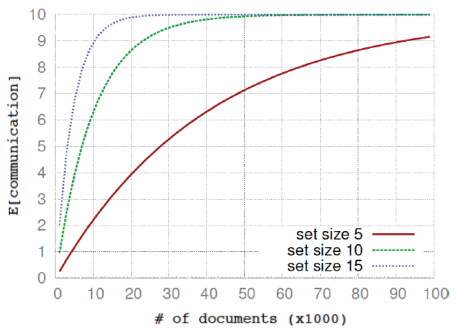
(a) Varying the vocabulary size



(b) Varying the number of tags per Tweet



(c) Varying the number of partitions



(d) Varying the number of documents

Figure 5.7: Expected communication

5.5 Operators and Topology

For the implementation of our approach, we have used Storm[1], a distributed stream processor engine described in detail in Section 2.2.3.1. The topology we create inside Storm is shown in Figure 5.8. The numbers inside the circles indicate whether there is one (1) or multiple (n) instances/tasks created for each operator. A different type of arrow is used to demonstrate each different kind of information flow.

[1]http://storm-project.net/

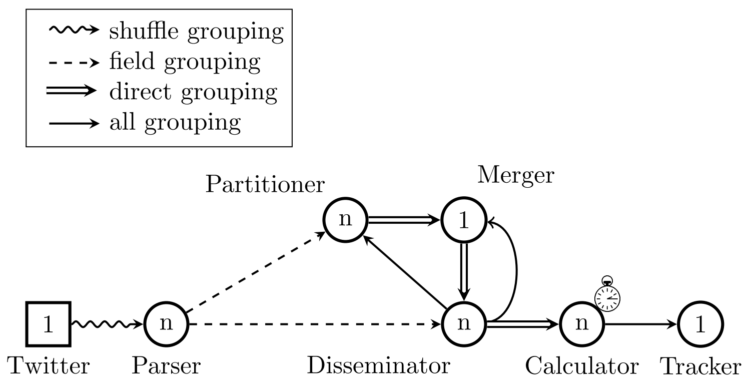


Figure 5.8: Topology

5.5.1 Parser

The Source (Spout) in our implementation produces a stream of tweets, using either live data obtained through Twitter's streaming API or, for repeatability of experiments, data 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 its authors to annotate it. This tagset can be enriched with named entities, location, or sentiment, extracted from the bodies of the messages and interpreted as additional tags. For each incoming tweet d_i, the Parser emits a tuple of the form (tm_i, d_i, s_i), where tm_i is the time of the arrival of d_i in the system. Disseminator and Partitioner operators both register to Parser.

5.5.2 Partitioner

The Partitioner operator is responsible to create the tag partitions using one of the algorithms presented in Section 6.2. To accelerate the procedure of partitioning, multiple instances of this operator can be created. Each instance of the Partitioner receives tuples from 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. [KS09]) of size $\mathcal{W}$ over the incoming tagsets. Conceptually, this window can be time-based (e.g. capturing 5 minutes of tweets) or count-based (e.g. 10,000 tweets), as described in more detail in Section 2.2.1. When the Partitioners are asked to create partitions they use the tagsets currently within 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 $(\{pr_1, pr_2, ..., pr_k\})$.

5.5.3 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.

Merger can be viewed as another Partitioner. It receives tagsets and outputs tag partitions. The tagsets it receives 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 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 5.6.

5.5.4 Disseminator

The Disseminator receives the partitions from the Merger and uses them to create an index from tags to Calculators (Section 5.2). The Disseminator receives also tuples (tm_i, d_i, s_i) from the Parser using *field grouping* on the whole tagset s_i and notifies the appropriate Calculators for their arrival. More specifically, for each received tagset s_i the Disseminator searches its index (Section 5.2) 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 C_j. For example, suppose that $s_i = \{a, b, c\}$ and Calculator C_1 is assigned the tags a, b, c and Calculator C_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.

Disseminators have two more, very important, responsibilities:

1. They identify in the input the tagsets not reflected in the partitions and inform the Merger about them.

2. 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 these are provided in Section 5.6.

5.5.5 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. To compute the Jaccard coefficients, the Calculators should know the cardinalities of the intersections of the co-occurring tags and, thus this is the information stored by them.

Consider, for example, that Calculator C_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.

5.5.6 Tracker

The Tracker operator receives the Jaccard coefficients emitted by the Calculators and uses them to perform further computations. Such a Tracker could be our prototype enBlogue which uses the Jaccard coefficients to identify emergent topics. When 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:

$$(s_i, J(s_i), CN(s_i))$$

where $J(s_i)$ is the Jaccard coefficient for the tagset s_i and $CN(s_i)$ is the value of the counter for s_i used during the computation of $J(s_i)$. 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.

5.6 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 5.4.1) while the set-cover based algorithms suffer from large amounts of redundant communication (Section 5.4.2). 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.

5.6.1 Evolving Partitions

Every time a Disseminator receives a tagset s_i, it checks whether each subset s_i^j is encapsulated in at least one notification. If this is not true there is no Calculator assigned s_i^j thus, the Jaccard coefficient for it cannot be computed. To enable the computation of the Jaccard coefficient, the Disseminator asks from the Merger to perform a *Single Addition* for s_i^j, i.e. to add s_i^j to the best possible partition.

Disseminators can tune the frequency of the *Single Additions* by asking for the additions of tagsets that are seen in the input at least sn times. Setting sn too low makes the system sensitive in spam tweets which introduce tags or create new co-occurrences that last for a short time. Setting it too high will result in missing new topics in their creation since there will be no Calculator computing the Jaccard coefficient for the new relations.

Note that the reception of a tagset s_i might cause a *Single Addition*, not necessarily for s_i but for a subset of it. Assume for example that $sn = 2$ and that Disseminator D_1 receives two tagsets $s_1 = \{a, b, c\}$ and $s_2 = \{a, b, d\}$. Assume also that there is no Calculator assigned $\{a, b\}$. With the reception of s_2 the tagset $\{a, b\}$ has been seen twice without being found in some Calculator. The limit has been reached and D_1 asks from the Merger to add it to some partition. This is not the case for neither of the tagsets $s_1 = \{a, b, c\}$ and $s_2 = \{a, b, d\}$.

When multiple Disseminators are used, the limit of sn cannot be strictly enforced. Consider again the tagsets $s_1 = \{a, b, c\}$ and $s_2 = \{a, b, d\}$. There is no guarantee that both of them will be received by the same Disseminator. In the worst case, a tagset might be seen in the input $sn' = (sn - 1) \times \#Disseminators + 1$ times before a Single Addition for it is asked. Receiving messages from the Parser using *field grouping* on the whole tagset ensures that when multiple documents use the same tagset s_i and not variations of it, then a Single Addition for this tagset will be performed as soon as s_i is seen in sn documents. This case is the most frequent case in our setting. In any case, setting $sn = 1$ can always guarantee that all new tagsets are considered independently of the number of Disseminators.

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. Subsequent receptions of the tagset are then forwarded to the Calculator selected by the Merger.

5.6.2 Partition Quality Monitoring

Every time a new partition has been created, the Merger notifies the Disseminators about it sending to each of them a tuple of the form:

$$(partitions, avgCom, maxLoad)$$

partitions are used by the Disseminators to create their index. *avgCom* and *maxLoad* contain the average communication and the maximum load of any of the created partitions respectively 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. When this is not true, the Disseminators ask from the Partitioners to create new partitions. Partitioners recompute partitions as soon as they receive a request from at least one Disseminator.

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 fraction of the most notifications sent to a single Calculator to the total sent notifications. 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 the threshold decreases the number of repartitions allowing for worse performance with respect to communication overhead or processing load.

5.6.3 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.

5.7 Experimental Evaluation

We have implemented the proposed operators in Java 1.7. For the experiments we used a cluster of 26 Linux (3.12.0) servers[2], 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.

5.7.1 Dataset

We obtained access to **Twitter's streaming API** at the "garden hose" level, that is, 10% of Twitter's public tweets and status updates. We have saved the Tweets we observed on September 5, 2013 and replay 6 continuous hours of them for our experiments.

5.7.2 Algorithms

In all experiments, we compare four algorithms:

DS: Disjoint Sets Algorithm creates partitions using disconnected communities of tags

SCC: Set Cover Based Algorithm that creates partitions optimising primarily the communication among the Disseminators and the Calculators and secondary the processing load in the Calculators.

SCL: Set Cover Based Algorithm that creates partitions optimising primary the processing load on the Calculators and secondary the communication among the Disseminators and the Calculators.

SCI: The Initial Set Cover Based Algorithm that, similarly to SCC, optimises primarily the communication among the Disseminators and the Calculators and secondary the processing load in the Calculators.

5.7.3 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, resulting in increased communication needs. 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.

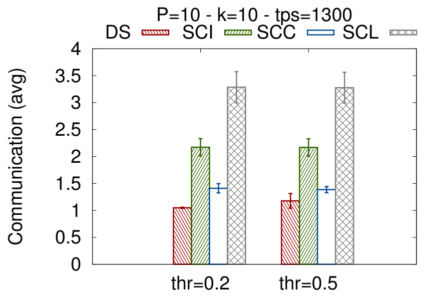
[2]We would like to thank the Database Architectures group at CWI Amsterdam for providing access to their computing cluster

Repartition Threshold *thr*: Repartition threshold defines the maximum percentage of change in the average communication or maximum load 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 average communication (or maximum processing load) is 50% worse than the average communication (maximum processing load) when the partitions were computed. Similarly, a threshold of 0.2 means that the average communication (or maximum processing load) is 20% worse than the average communication (maximum processing load) when the partitions were computed.

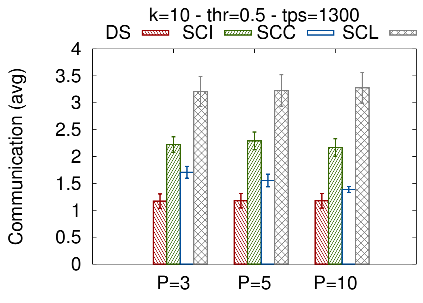
5.7.4 Experimental Results

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.*

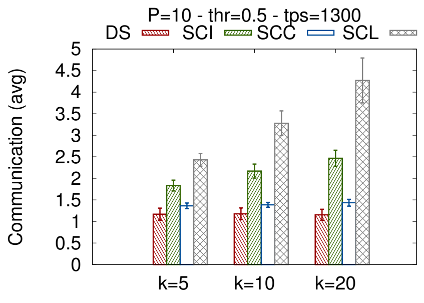
All configurations use one Parser and one Disseminator. The Disseminator asks for a Single Addition when a tagset is seen in the input without being found in any Calculator 3 times. The statistics used to estimate the quality of the partitions (average communication and maximum load) are computed for every 1000 tweets for which there was a notification sent. Calculators report



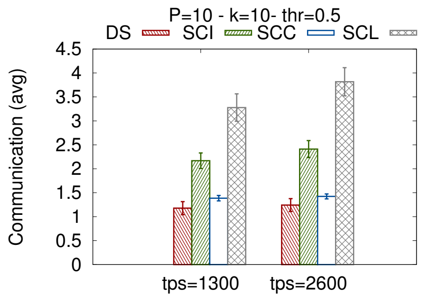
(a) Varying threshold



(b) Varying Partitioners



(c) Varying partitions



(d) Varying tweets rate

Figure 5.9: Communication

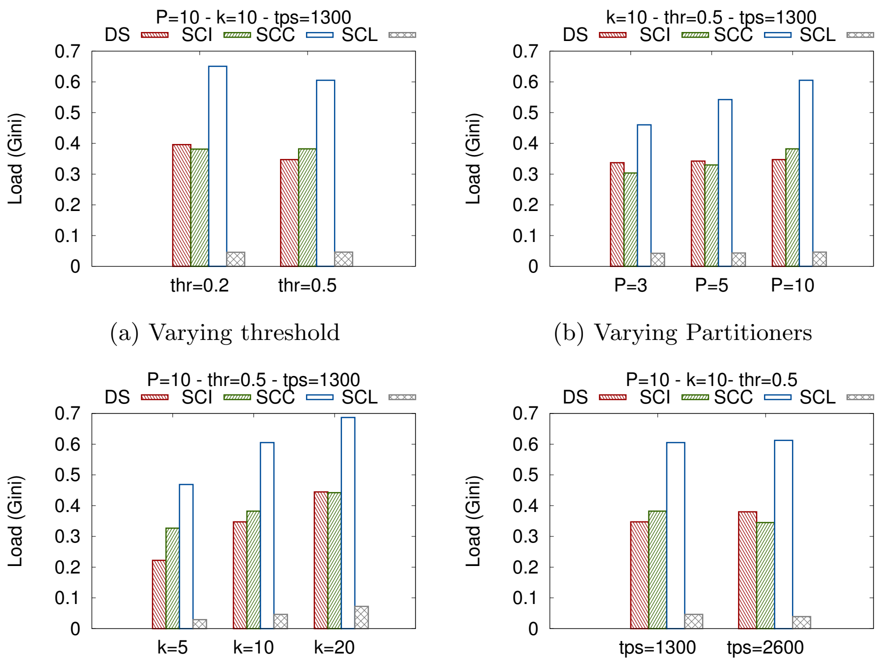
the Jaccard coefficients every 5 minutes. The Partitioners create the partitions using the set of tweets seen in the previous 5 minutes.

5.7.4.1 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 5.9 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 5.9c). 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.

5.7.4.2 Processing Load



(a) Varying threshold (b) Varying Partitioners

(c) Varying partitions (d) Varying tweets rate

Figure 5.10: Processing Load

We define the Processing Load of a single Calculator C_i to be the fraction 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 (see also Section 2.5). The plots in Figure 5.10 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 the most is the number of partitions (Figure 5.10c). 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 5.10b). 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 5.10b suggests that the careful selection of the next tagset, although keeps communication low (cf. Figure 5.9b) cannot support load balance.

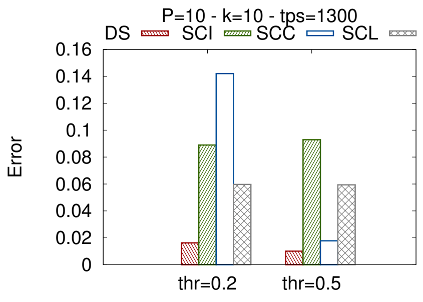
5.7.4.3 Jaccard Accuracy

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, 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, resulting in missing completely the information about this tagset.

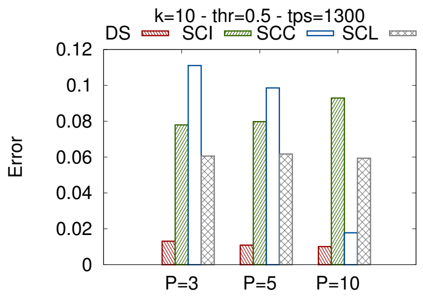
To measure the error in accuracy and the total loss of coefficients, we implemented an approach with a single Calculator which gets all tagsets and computes the Jaccard coefficients having full knowledge of the data. We use the results of this approach as our baseline. Since a tagset is added when seen at least 3 times, the baseline 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 5.11, we report on the average error of these Jaccard coefficients.

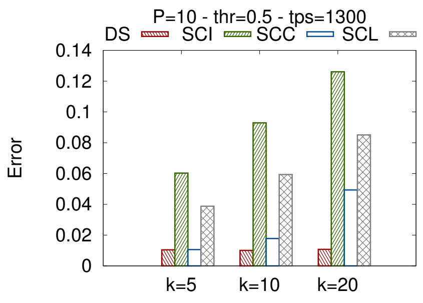
In general, DS is the algorithm computing the most accurate coefficients. It is interesting that an increase in the number of Partitioners causes a significant reduction in the error of SCC (Figure 5.11b). Additionally, an increase in the tweets input rate decreases the error in all algorithms but SCC (Figure 5.11d). Both these cases are related to the number of repartitions (see plots in Figure 5.12). When a repartition happens, it might be the case that tagsets assigned to some Calculator before the repartition are assigned to another Calculator after it. This causes multiple coefficients for the same tagsets being reported with none of them being accurate.



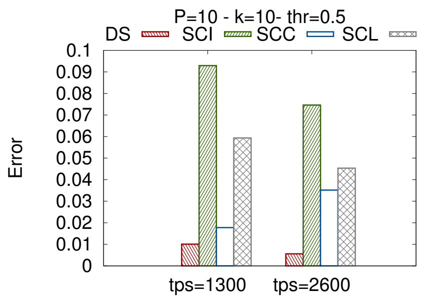
(a) Varying threshold



(b) Varying Partitioners



(c) Varying partitions



(d) Varying tweets rate

Figure 5.11: Error for tagsets seen more than 3 times

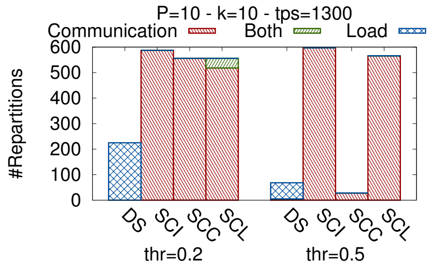
5.7.4.4 Number of Repartitions

The number of repartitions depicts how Single Additions affect the quality of the partitions. A repartition is triggered when either the communication or the processing load exceed the threshold. In some cases, both measures had been found to have exceeded the threshold. The plots in Figure 5.12 show the effect of the various parameters on the number of repartitions. As expected, DS has 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 5.12a) . This contradicts our expectations and suggest that the average communication for these algorithms is easily affected by the Single Additions.

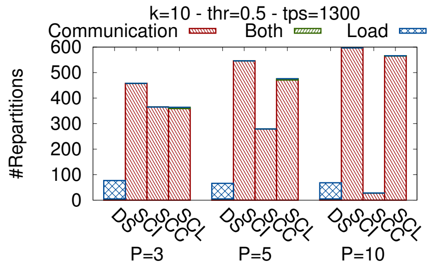
5.7.4.5 Evolution of Partitions

The plots in Figures 5.13 and 5.14 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.

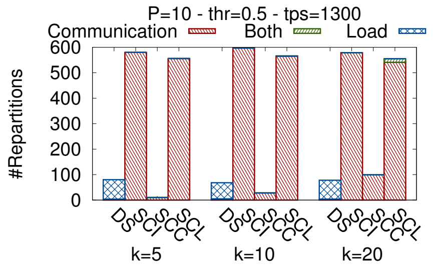
The plots regarding DS clearly show the effect of the Single Additions and



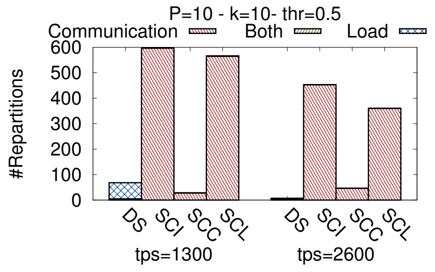
(a) Varying threshold



(b) Varying Partitioners



(c) Varying partitions



(d) Varying tweets rate

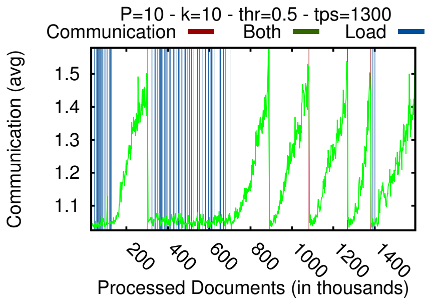
Figure 5.12: Number of Repartitions

the repartitions to the communication (Figure 5.13a) and the processing load (Figure 5.14a). As long as there are repartitions happening, the communication stays low while there is one Calculator having more load than 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.

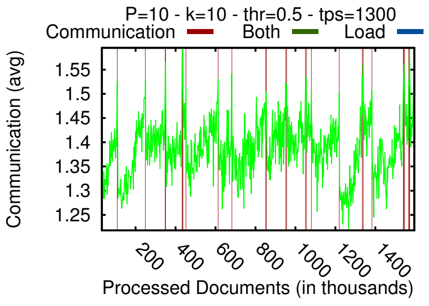
Similar results are seen for SCC in Figures 5.13b and 5.14b. For SCL and SCI, the results are not that clear since there are very many repartitions (there is approximately one repartition every 2750 processed documents). However, for SCL, Figure 5.14c clearly shows that the processing load is balanced through the whole experiment. The load for SCI (Figure 5.14d) is not balanced, but one can see that it is better balanced than the load for SCC (Figure 5.14b).

5.7.4.6 Connectivity of Tagsets

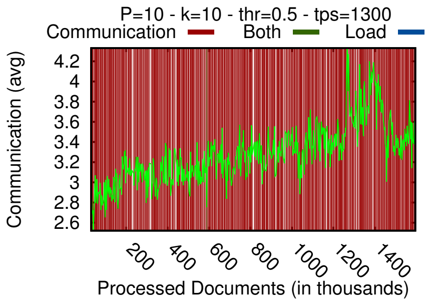
In Figure 5.15, we report some statistics regarding the Twitter dataset as they are fundamentally related to the problem we consider in this work and, in particular, relevant for the performance of the DS algorithm. For the measurements, we used the same tweets we used for the experiments. Over them, 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.



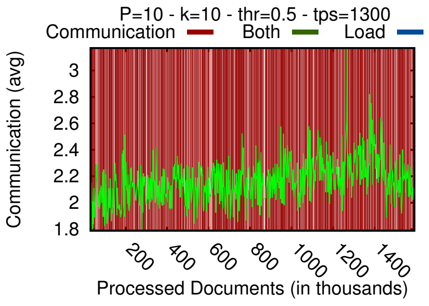
(a) DS Communication



(b) SCC Communication

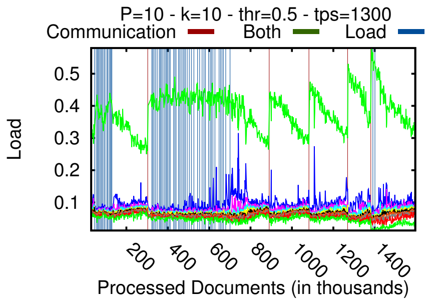


(c) SCL Communication

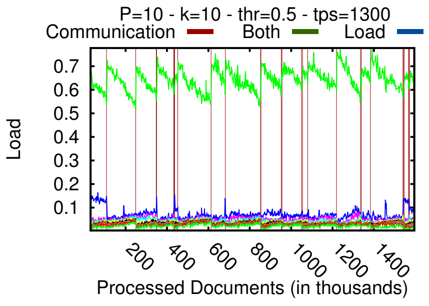


(d) SCI Communication

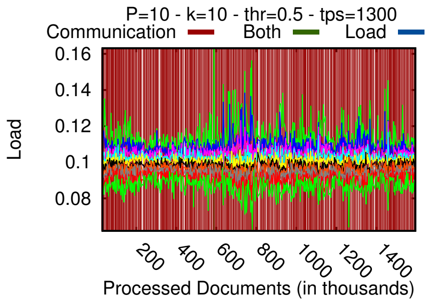
Figure 5.13: Communication over Time



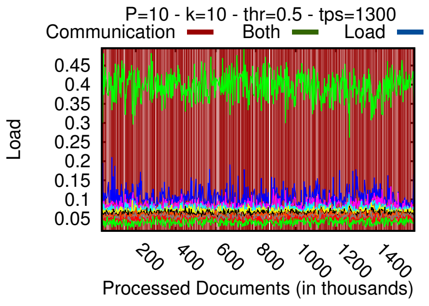
(a) DS Load



(b) SCC Load

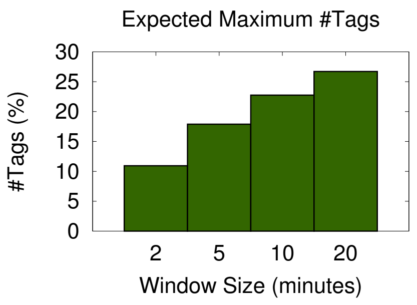


(c) SCL Load

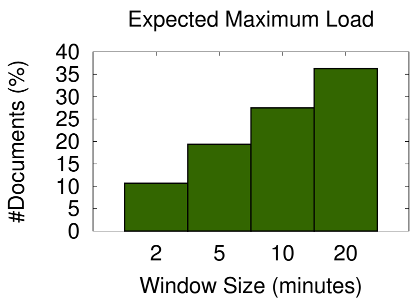


(d) SCI Load

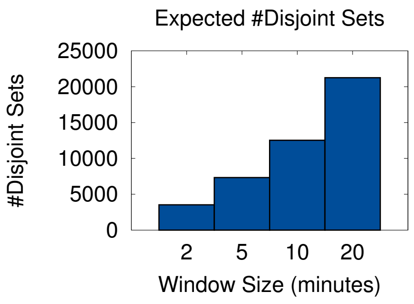
Figure 5.14: Processing Load over Time



(a) Maximum size of connected tagset per round



(b) Maximum load of connected tagset per round



(c) Number of connected tagsets per round

Figure 5.15: Tagsets connectivity and load

5.7.5 Discussion

DS has been shown experimentally to give the most accurate results. Since it creates partitions using disjoint sets, there are fewer tagsets assigned to multiple nodes, thus there are fewer Jaccard coefficients reported multiple times. It is this characteristic of DS however, that can make the use of it impossible. When not enough disjoint sets are found or a few disjoint sets dominate the dataset, SCC can be used instead. For some configurations, SCC achieves accuracy results close to those achieved by DS. Additionally, since SCC is a greedy algorithm, it is simpler and of lower complexity to DS which needs to have a global view of the data in order to find the disjoint sets. Both DS and SCC focus on communication resulting in very unbalanced processing load. In case balancing load is important, SCL should be used. With small changes, SCL can handle also cases in which not all nodes are equal and each of them can handle different load. For those cases, SCL should know the characteristics of each node. Finally, SCI achieves communication worse than SCC but better that SCL and processing load worse than SCL but better than SCC. This puts SCI in between SCC and SCL and makes it appropriate for cases when one cannot afford to optimise one measure regardless of what will happen to the other one and an average performance on both of them is more desirable.

In general, the results suggest that the proposed approach would exhibit better performance when the relations among tags are more static. In such a case, new tagsets would be encountered less often, minimising the need for repartitions. A comparison between the plots in Figure 5.11 and Figure 5.12 suggests a correlation between the accuracy and the number of repartitions, with less repartitions leading to lower error. An example dataset with these characteristics could be Twitter data grouped over bigger time periods. For example, one could consider tagsets that co-occur in a number of consecutive small, e.g. 5 minutes, windows. These tagsets reflect more permanent relations, for example, {`#Bieber, #Gomez`}. Other tag co-occurrences could be considered to represent sub-events with shorter life span, e.g. {`#Bieber, #Gomez, #tattoo`}. In order to compute the Jaccard coefficients for tagsets corresponding to sub-events, Disseminators could ignore non-assigned tags and forward the tagsets to all Calculators having any subset of the tagsets' tags. In the current approach, a tagset is forwarded to a Calculator only if the Calculator has been assigned all the tags in the tagset.

For datasets with rapidly changing tag relations, e.g. the Twitter dataset when focusing on small subsets of previous data, a different topology, not assuming any kind of relations preserved in time, might be more appropriate. Such a topology is shown in Figure 5.16. In this topology each operator performs a simple processing on the data it receives and quickly forwards the message to the next operator, trying to avoid having messages queued in its input.

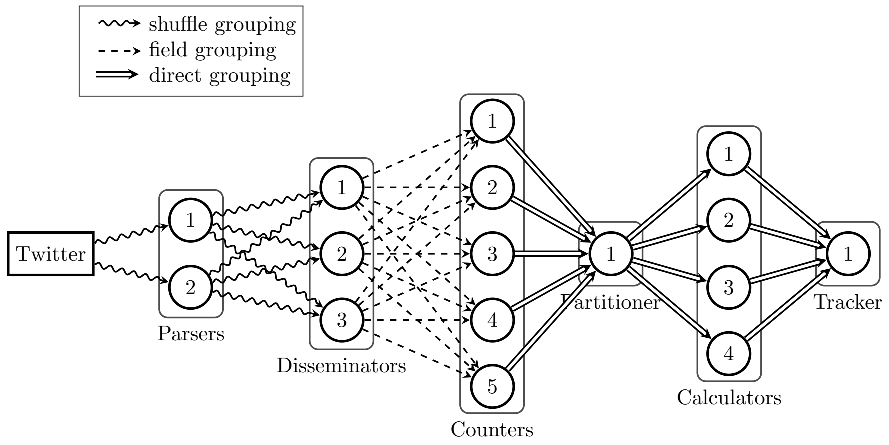


Figure 5.16: Topology

More specifically, the Parsers receive tweets using shuffle grouping. They extract from each tweet the set of tags reflecting its topic and forward it to the Disseminators using shuffle grouping. The Disseminators create all possible subsets for each received tagset and forward each of them to one Counter using field grouping. This way, the same tagset is always received by the same Counter. The Counters, as their name implies, are responsible to maintain one counter for each received tagset. From time to time, the Counters sent the tagsets along with the computed counters to the Partitioner. The Partitioner decides how to partition the tagsets to the Calculators and forwards the counters according to this decision. For the partitioning, any of the algorithms presented in Section 5.3 can be used. The Calculators, upon receiving the counters, combine them to compute the Jaccard coefficients for as many tagsets as possible. The Jaccard coefficients are forwarded to the Tracker which can use them for further processing.

We implemented this topology and, since it does not pre-assign the tags to the Calculators, it managed to compute a Jaccard coefficient for any seen tagset. The accuracy of the computed coefficients reached 100%. As can be seen in Figure 5.16, this topology has a single Partitioner which is a drawback, since it can be a bottleneck. However, the Partitioner receives messages in batches and should process them before the next batch is received in order to avoid having messages queued. Considering small time windows minimises the load of the Partitioner, since less tagsets are seen in the input.

5.8 Distributed Emerging Topics Identification

Computing distributed Jaccard was inspired by our enBlogue approach on finding emerging topics. In enBlogue (see also Chapter 4) the idendification of emerging topics can be broken down into three main steps:

- **Step 1:** The correlations of the tagsets in the current window are computed.
- **Step 2:** For each correlated tagset, the expected correlation is computed.
- **Step 3:** The emergence of each topic is estimated comparing the real and the expected correlations of the tagset representing it.

Our approach on computing in parallel the Jaccard coefficients of all possible sets of co-occurring tags can be used in Step 1 to compute the correlation values of all tagsets in the current window. In this case, the computed correlations are forwarded to the Tracker. The Tracker can then compute for each received coefficient the expected correlation value and use them subsequently to estimate the emergence of each topic (Steps 2 and 3). However, having a single machine computing all expected correlations and estimating the emergence of all topics could potentially shadow the benefits achieved from parallelising the computation of the Jaccard coefficients.

A straightforward method to parallelise Steps 2 and 3 is to use the Tracker as a Distributor that collects all computed coefficients and splits them to a number of nodes. Each node would then be responsible to compute the expected correlations and estimate the emergence for the topics it has received coefficients for. However, since the tagsets are already split into various nodes, the Calculators, it seems natural to have the same nodes computing also the expected correlation and estimate the emergence of the topics. This is not trivial, thus, since it is not known in advance which Calculator will compute which coefficients.

In order for a node to be able to compute the expected correlation of a tagset, a number of previous correlation values are required. Since a tagsets is not always assigned to the same Calculator, these previous correlations should be requested from other Calculators. This implies the existence of some form of communication between the Calculators which is not allowed in our approach and could potentially result in the exchange of numerous messages between the nodes until all of them have the information they need. Additionally, the Calculators would have to store a number of previous coefficients in order to be able to forward them to other Calculators. Currently, the Calculators are stateless and only information about the current coefficients is stored on them.

However, any attempt on increasing the accuracy of the computed coefficients in the current approach might require a form of communication between Calculators, so that they can, for example, exchange counters after a repartition. In such a case, the same communication channels could be used to exchange previous coefficients that will allow the Calculators to compute the expected coefficients and estimate the emergence of the topics. The necessary changes to achieve this, are not straightforward and require further research to avoid flooding the system with messages, while sending all necessary information to each Calculator.

5.9 Summary

In this chapter, we presented in detail an approach that can be used to compute in parallel the Jaccard coefficients for multiple tagsets. According to the proposed approach, all tagsets used in the tweets sent during a short past period are considered together, and a partitioning of the tags in them is decided assuming that the relations between the tags will remain the same in the future. Each partition is assigned to a different node which receives documents based on it and computes the Jaccard coefficients for all sets of co-occurring tags in it. Four algorithms that can be used for the partitioning of the tags, focusing either on communication or in processing load, have been proposed. New tagsets, not covered in the partitions are incrementally added to them to avoid loosing the corresponding Jaccard coefficients completely. Repartitions of tags are created from time to time to avoid excessive communication or load imbalance.

We experimentally evaluated our approach considering all four partitioning algorithms. The results proved the feasibility of it. However, the results are greatly affected by the highly dynamic nature of tag relations in Twitter. This lead us to consider a possible alternative approach that does not assume the same relations for the future but forwards any received tagset based only on the tags it contains, i.e. there are no pre-assigned tags to nodes. This alternative approach demands, in general, the exchange of many more messages but, when bandwidth is no problem, the alternative approach can be used to get Jaccard coefficients with accuracy 100% for all tagsets seen in the input, regardless of the total number of times they have been encountered. However, the presence of a single Partitioner is a potential bottleneck that requires further research.

Chapter 6

Events Identification in Relational Databases

In the previous chapters, we focused on real time identification of events over unstructured or semi-structured dynamic data. In this chapter, we focus on structured data stored in relational databases. Relational databases can store massive amounts of data, allowing ad-hoc and predefined queries over them. A common approach to obtain further insights on this data is to transfer it in a data warehouse. The data in a data warehouse is organised over multiple attributes and a measure of interest is precomputed for each combination of attributes binding, as described in more detail in Section 2.7.

The usually big volume of data in these settings does not allow the immediate reflection of the updates performed in the database to the data warehouse, since even a small update in the data can cause a chain of re-computations in the data warehouse. This makes very difficult for analysts to identify and react immediately in changes caused in the data.

While keeping everything up-to-date is almost impossible, maintaining only the essence of it is feasible and of interest to the users. Arguably, the most natural way to condense large amounts of information into a conceivable form is the computation of rankings. Rankings allow users to focus on the portion of data that shows an outstanding performance. Changes on the top positions of the rankings are usually most important compared to changes in the middle positions of the rankings. These are the changes that need to be identified in real time.

For instance, consider a manufacturer that uses a number of raw materials to create her final product. Monitoring in real time, for example, the top-20 materials with the least quantity in stock is of great importance to her. Tracking this information in parallel with the quantities of each material that are used for the creation of each final product allows her to decide as early as possible the necessary supply orders. Failing to spot as early as possible the top materials with the least available quantities could result in stopping the production due to deficiencies in material resources. Stopping the production might, not only, result in temporary revenue loss for the manufacturer but also in the complete loss of the consumers confidence to the company.

In this chapter, we present two algorithms that allow identifying changes in the top-k positions of rankings created over the various dimensions defined in a data warehouse. This work has been presented at the 7th International Workshop on Ranking in Databases (DBRank) held in conjunction with the International Conference on Very Large Data Bases (VLBD 2013) [AM13b].

6.1 Framework

We consider a database storing a set of relations. A user defines the rankings of interest providing an *SQL-like* query using the following template:

```
SELECT primary attribute ,
   function(numeric attributes)
GROUP BY secondary attributes
TOP k
ORDER direction
```

Through the above SQL-like query template, the user provides information about:

primary attribute : The rankings are created over the instances of the primary attribute. The performance of these instances should be monitored.

secondary attributes : The secondary attributes are used to define groups. Each possible combination of instances of these attributes defines a different group. The performance of the instances of the primary attribute should be monitored in each of these groups. The secondary attributes define the dimensions in a data warehouse.

numeric attributes : The numeric attributes are used to compute the measure of interest for each instance of the primary attribute.

function : The function that is used to compute the measure of interest.

k : The number of the primary attribute instances comprising the rankings. This top portion of instances, in each ranking, should be maintained and be always up to date.

direction : Whether the best (*descending*) or the worst (*ascending*) performing instances are of interest.

Using the information provided by the user, we create one top-k query for each possible combination of the secondary attributes instances according to the following template:

```
SELECT primary attribute ,
   function(numeric attributes)
FROM relations
WHERE (secondary attribute)_1 =
          (secondary attribute)_1.instance_i
  AND (secondary attribute)_2 =
          (secondary attribute)_2.instance_j
   AND ...
   AND (secondary attribute)n =
           (secondary attribute)n.instancel
GROUP BY primary attribute
ORDER BY function(numeric attributes) direction
LIMIT k
```

For example, if the user is interested in monitoring the top-20 product types with the highest revenue of each brand in each country stored in the relations products and sales, she should provide the following SQL-like query:

```
SELECT products.type,
   SUM(products.price*sales.quantity)
GROUP BY products.brand, sales.country
TOP 20
ORDER ascending
```

The queries derived from the above, called ranking queries in the rest of the chapter, are of the following form:

```
SELECT P.type, SUM(P.price*S.quantity)
FROM products P, sales S
WHERE P.id = S.pid AND P.brand='X' AND
   S.country='Y'
GROUP BY P.type
ORDER BY SUM(P.price*S.quantity) ASC
LIMIT K
```

where `'X'` and `'Y'` are instances respectively of the attributes `products.brand` and `sales.country`. One such query is created for each possible combination of the attributes `products.brand` and `sales.country`.

For consistency, we introduce the keyword *ANY* which is used to indicate that a secondary attribute is not bound to a specific instance. For example, the following query is used to monitor the product types of *ANY* brand in a specific country `'Y'` that have the top-k highest revenues.

```
SELECT P.type, SUM(P.price*S.quantity)
FROM products P, sales S
WHERE P.id = S.pid AND P.brand=ANY AND
   S.country='Y'
GROUP BY P.type
ORDER BY SUM(P.price*S.quantity) ASC
LIMIT K
```

Allowing any of the secondary attributes not to be bound to a specific instance creates the generalisations used in a data warehouse.

6.1.1 Updates

A continuous stream of updates, in the form of insertion queries, changes the data stored in the database. The updates we consider are of the following form

(`updated instance`, `properties`, `added value`)

where the `updated instance` is the instance of the primary attribute that is affected by the update, `properties` are the instances of the secondary attributes that define the properties of the updated instance and `added value` is the change in the score of the updated instance imposed by the update. An example update is the following:

```
(products.type='Z', {products.brand='X',
   sales.country=ANY, products.price=452},
   sales.quantity=+3)
```

For simplicity, we assume that each update contains information about all attributes involved in the rankings (primary, secondary and numeric). This assumption can be easily withdrawn by creating a pre-processing step which will take the update and will obtain the missing information from the database. Querying the database might seem expensive but with the use of appropriate indices the missing information can be retrieved in O(1). For example, in the above update, the price of the product might be missing. Using an index on the product type allows to quickly obtain this information. The instances over which these indices should be created are not expected to change frequently, e.g. new product types are not added every day, not even every month. Thus, updating the indices will be an infrequent procedure, adding only a minimal overhead in the whole maintenance framework.

After each insertion, all the above rankings should reflect the most recent information. Re-evaluating them after each update is a straightforward solution. Unfortunately, traditional RDBMs do not benefit from the limit condition ([IBS08]) present in all our ranking queries to early terminate their evaluation. They evaluate them as if the limit condition was not present and then they output the top-k results. But even in the case of early termination being utilised by the RDBMs, the aggregate nature of the above queries makes it very difficult to find an aborting condition that will allow accessing fewer data.

In our approach, we focus on the maintenance of the ranking queries rather than their re-evaluation and propose two algorithms to achieve this goal. Both algorithms focus on updating the top-k results minimising the interaction with the underlying database. Estimating the top-k results of a query requires knowledge about a big portion of the data and in a setting with multiple such queries having all necessary data in memory might not be possible. In such a case, updates can cause a number of disk accesses leading in a significant degradation in performance.

6.2 Algorithms

A simple and obvious optimisation is to ignore updates that cannot affect the ranking because of the structural differences between the update query and the ranking query. This applies when either of the following is true:

- The update query affects a relation that is not used in the ranking query.
- The update query affects a relation used by the ranking query but not the same attributes.

Consider for example the update:

```
(products.type='Z', {products.brand='X',
   sales.country=ANY, products.price=452},
   sales.quantity=+3)
```

This update is ignored by all rankings not using the relations `products` and/or `sales` and the rankings using either of the two relations but none of the attributes `products.type`, `products.brand`, `sales.country`, `products.price` and `sales.quantity`.

All other updates can potentially affect the top-k results and, thus, need to be handled by the algorithms we present.

6.2.1 Naive Approach

In a naive approach, each ranking stores in memory its top-k results and every time it receives an update, it checks whether the updated instance exists in this top-k portion. In case it does, it updates its score. Updating incrementally the result of an aggregate function is straightforward when the function used is distributive, e.g. sum, count or average. Incrementally updating non-distributive aggregate function is quite challenging and [PSCP02] studies this problem. In the worst case, the set of values used to compute the aggregated score could be stored and re-used every time the score needs to be updated.

In case the instance affected by an update is not found in the top-k portion, a query is issued to the database to obtain the measure of interest for the missing instance and decide whether it should enter the top-k results. We call such a query *Verification Query.*

Definition 3. *A Verification Query is the ranking query having the primary attribute bound to a specific instance.*

For our products–revenue example, the Verification Query executed to obtain the aggregated value for the product type `'Z'` of brand `'X'` in country `'Y'` is the following

```
SELECT P.type, SUM(P.price*S.quantity)
FROM products P, sales S
WHERE P.id = S.pid AND P.brand='X' AND
   S.country='Y' AND P.type='Z'
GROUP BY P.type
```

Since the top-k results are assumed to contain a rather small portion of all instances, most updates in the Naive approach will need the execution of a Verification Query, resulting in a significant degradation of the system's performance.

6.2.2 Estimates Algorithm (EA)

An optimisation can be achieved exploiting the top-k nature of the rankings. Each ranking needs to have exact scores only for its top-k instances. Hence, the rest of them can have an *estimated score*. This is the idea in the Estimates Algorithm (EA).

In EA, all rankings store in memory the exact scores of their top-k results. In addition, each ranking has a *Buffer* where instances with non-exact, or estimated, scores are stored. The estimated score is always better (or the same) to the real (exact) score of the instance. This assures that no instances that qualify for the top-k portion are ever missed.

Definition 4. *The basic score, for each ranking, is the worst score of any of the instances for which an exact score is maintained.*

Definition 5. *The estimated score, for each ranking, is the basic score assigned to an updated instance when its exact score is not known.*

To allow bigger flexibility on the range of the estimated score, the exact scores of N extra instances are maintained, resulting in a top-(k+N) ranking. In this case, the basic score is selected to be the worst score of the (k+N) instances.

Every time an update affects an instance which score is not known, the algorithm assigns to it an estimated score equal to the basic score and stores it in the Buffer. If an update affects an instance already in the Buffer, then its estimated score is updated as if it was its exact score. This is done so that the rankings can consider the cumulative changes in an instance's estimated score when it is updated multiple times. When the estimated score of an instance qualifies for the top-k results, its real score does not necessarily qualify too. To verify this, a Verification Query regarding this instance is executed.

If the Buffer becomes full, no more instances can be added to it and the algorithm falls back to the Naive Approach. To avoid that, when the Buffer is found to be full, a query is issued that verifies all instances currently in the Buffer. We call this query *Buffer Reset Query.* After such a query, the Buffer is reset to empty allowing again the addition of new instances.

Definition 6. *A Buffer Reset Query is the ranking query having the primary attribute bound to a set of instances.*

For our products - revenue example, if the Buffer stores three instances of the `products.type`, `‘A’,‘B’` and `‘C’` the Buffer Reset Query that will be executed is the following

```
SELECT P.type , SUM(P.price*S.quantity)
FROM products P, sales S
```

```
WHERE P.id = S.pid AND P.brand='X' AND
   S.country='Y' AND P.type IN
   ('A','B','C')
GROUP BY P.type
```

The in-memory structures necessary for the EA are shown in Figure 6.1. The idea of EA is described in the pseudocode of Algorithm 12. Every time we transfer an instance to the top-k or to the N extra instances, another instance is removed to keep the sizes of these sets unchanged. The instances in the top-k portion are always kept sorted according to the criteria defined by the user. This does not apply for the N extra instances.

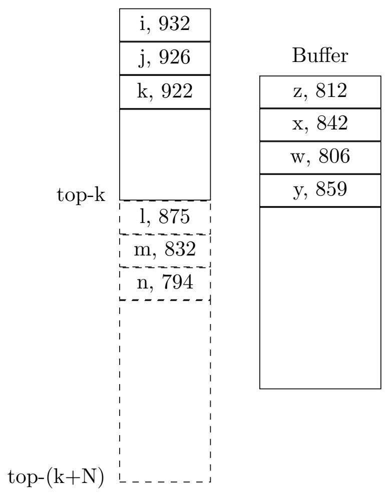


Figure 6.1: In-Memory structures: the actual top-(k+N) ranking (left) and the estimates for the previously unseen entities (right).

Two types of queries are executed in EA:

- the Verification Queries and
- the Buffer Reset Queries

A reduction in the number of Verification Queries is achieved through the selection of the basic score. Increasing the difference (gap) of the basic score to the worst score of any instance in the top-k results, increases the number of updates an estimated score can absorb before qualifying for the top-k portion. Of course, increasing the gap increases also the number of extra instances (N) that should be stored in memory. A reduction in the number of Buffer Reset Queries is achieved through the size of the Buffer. The bigger the Buffer, the more instances it can store and the less time it needs to be reset.

6.2.3 Groups Algorithm

The queries we examine, apart from focusing on the top-k portion of the results, have a special relation with each other: groups of them use the same primary, secondary and numeric attributes, and the same aggregation function. These

Algorithm 12: Estimates Algorithm (EA)

```
Input: Affected instance $inst$
       Integer $chg_{inst}$ // the change in the affected instance's score
       Integer $topK_{thr}$ // worst score in the top-k instances
       Integer $extra_{thr}$ // worst score in the N-Extra instances
if $inst \in top - k$ then // check the top-k instances
    update $score_{inst}$
else if $inst \in N - Extra$ then // check the N-Extra instances
    update $score_{inst}$
    if $score_{inst} > topK_{thr}$ then
        transfer $inst$ to $top\text{-}k$
    end if
else if $inst \in Buffer$ then // check the Buffer
    update $score_{inst}$
    if $score_{inst} > topK_{thr}$ then
        Execute query
        $score_{inst} = score_{query}$
        if $score_{inst} > topK_{thr}$ then
            transfer $inst$ to $top\text{-}k$
        else if $score_{inst} > extra_{thr}$ then
            transfer $inst$ to $N\text{-}Extra$
        end if
    end if
else // new
    Compute estimation
    Add $inst$ to $Buffer$
end if
```

queries can be organised in a subgroups lattice according to the tuples qualifying to their filtering condition. For the products-revenue example query, introduced in Section 6.1, assuming that there are only two instances of countries, `‘Y’` and `‘W’`, and a single instance of brand `‘X’` the lattice will be the one in Figure 6.2.

The basic characteristic of the queries organised in a lattice is that they share the same tuples. Each query lying in a join in the lattice is satisfied by all the tuples in the union of the sets of tuples satisfying the queries lying below it. Each query lying in a meet point is satisfied by all the tuples in the intersection of the sets of tuples satisfying the queries lying above it. This partial order relation between the aggregate queries is an immediate consequence of the filtering conditions of the queries and can help decreasing the interaction with the underlying database.

Coming again to the products-revenue example, consider the example update

```
(products.type=‘Z’, {products.brand=‘X’,
   sales.country=ANY, products.price=452},
   sales.quantity=+3)
```

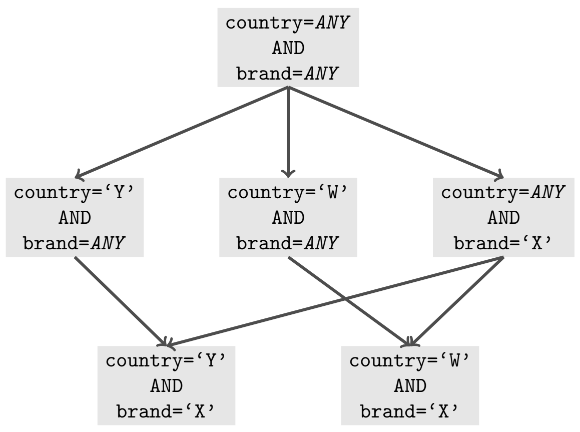

country=ANY AND brand=ANY
country='Y' AND brand=ANY
country='W' AND brand=ANY
country=ANY AND brand='X'
country='Y' AND brand='X'
country='W' AND brand='X'

Figure 6.2: Subgroups lattice organising the top-k aggregate queries using the same primary attribute, and the attributes country and brand in the filtering condition

which causes an increase of 3 units in the quantity sold for product type `'Z'`. The supremum ranking of the lattice (i.e. the ranking having the filtering condition `products.brand=`*`ANY`* `AND sales.country=`*`ANY`*) is the first ranking to know about this update. Assume that the product type `'Z'` does not exist in the top-(k+N) results of the supremum ranking and its estimated score qualifies for the top-k results. In this case, the following Verification Query will be issued by the supremum ranking in order to get the exact score of the product type `'Z'`.

```
SELECT P.type, P.brand, S.country,
   P.price*S.quantity
FROM products P, sales S
WHERE P.id = S.pid AND P.type=Z'
```

The Verification Query executed in this case is quite different from the Verification Query executed in the EA algorithm. The result of the Verification Query is not the aggregated score for product type `'Z'` but the set of tuples that should be used to compute the aggregated score for product type `'Z'`. The ranking uses the retrieved tuples to compute the aggregated score itself, outside of the database. At the same time, the retrieved tuples are forwarded to the rankings lying lower in the lattice.

In case the updated instance, `'Z'` in our example, already exists in the top-(k+N) results or its estimated score does not qualify for the top-(k+N) results, a Verification Query will not be executed, no tuples will be available and the update itself, instead of tuples, will be forwarded to the lower level rankings.

Receiving an update, a ranking in the lower level will check to find the updated instance in its Buffer or top(k+N) results. In case the instance cannot be found, a Verification Query will be executed. The ranking has already a filtering condition so, it will use it to further limit the retrieved tuples, since tuples that do not satisfy the filtering condition of the ranking will be of no use to this and to all rankings below it in the lattice. Assuming for example, that the ranking has the filtering condition `products.brand=`*`ANY`* `AND sales.country=`*`'W'`* the Verification Query will be the following:

```
SELECT P.type, P.brand, S.country,
   P.price*S.quantity
FROM products P, sales S
WHERE P.id = S.pid AND P.type='Z' AND
   p.country='W'
```

The difference of this Verification Query to the Verification Query executed by the supremum ranking is that this query has in addition the condition `p.country='W'` in the WHERE clause.

When receiving tuples, a ranking filters out the ones that do not qualify to its filtering condition and tries to use the remaining ones in order to avoid execute new queries. In case the updated instance does not exist in the top-(k+N) results, the ranking uses the received tuples to compute the exact score of the instance and decide whether it should be added in the top-(k+N) results. If the updated instance already exists in the Buffer, it is removed. Removing instances from the Buffer, re-using tuples obtained through the execution of Verification Queries by rankings lying higher in the lattice, allows emptying spaces for new instances in the Buffer that finally results in reducing the Buffer Reset Queries that should be executed.

If the aggregation function used by the rankings is distributive, e.g. `SUM` or `COUNT`, the ranking can query the score of each individual group instead of the single tuples, saving the cost of computing the score in each ranking separately.

We call this algorithm Groups Algorithm (GA). The idea of GA is described by the pseudocode of Algorithm 13. In GA, rankings retrieve tuples, instead of aggregated scores, also when they execute Buffer Reset Queries. These tuples are, similarly, forwarded to the rankings lying in the lower levels of the lattice which re-use them to compute exact scores for instances that possibly lie in their Buffer. This results in further reduction of the Buffer Reset Queries that need to be executed.

6.3 Experimental Evaluation

We implemented our methods in Java 1.6 using a multi-threaded approach and conducted a series of experiments. All measurements were performed on a server with two quad-core 2.4 GHz Intel Xeon processors and 47 GB of RAM. For the experiments we used the TPC-H dataset[1], a typical dataset for decision support systems.

As primary attribute, we used the attribute `p_partkey` from table `part`. We used `customer.c_mktsegment`, `orders.o_orderpriority` and `region.r_name` as the secondary attributes. The selected aggregate function was `sum` and the numeric attribute over which we computed the scores was `lineitem.l_quantity`. Using all possible combinations of the secondary attributes bindings resulted in 216 queries in total. Each of them involved five relations.

[1]http://www.tpc.org/tpch/

Algorithm 13: Groups Algorithm (GA)

```
Input: Affected instance $inst$
       Integer $chg_{inst}$ // the change in the affected instance's
       score
       Set of tuples $TUP$ /* the tuples comprising the various
    group for the affected or removed from estimates instances
    as sent from an upper level node                          */
       Integer $topK_{thr}$ // worst score in the top-k instances
       Integer $extra_{thr}$ // worst score in the N-Extra instances
1  Use $TUP$ to remove from estimates as many instances as possible
2  if $inst \in top - k$ then // check the top-k instances
3      update $score_{inst}$
4  else if $inst \in N - Extra$ then // check the N-Extra instances
5      update $score_{inst}$
6      if $score_{inst} > topK_{thr}$ then
7          transfer $inst$ to $top\text{-}k$
8      end if
9  else if $TUP \neq \emptyset$ then // compute score using received tuples
10     Compute $score_inst$ using $TUP$
11     if $score_{inst} > topK_{thr}$ then
12         transfer $inst$ to $top\text{-}k$
13     else if $score_{inst} > extra_{thr}$ then
14         transfer $inst$ to $N\text{-}Extra$
15     end if
16 else if $inst \in Buffer$ then // check the Buffer
17     update $score_{inst}$
18     if $score_{inst} > topK_{thr}$ then
19         Execute query
20         $score_{inst} = score_{query}$
21         if $score_{inst} > topK_{thr}$ then
22             transfer $inst$ to $top\text{-}k$
23         else if $score_{inst} > extra_{thr}$ then
24             transfer $inst$ to $N\text{-}Extra$
25         end if
26     end if
27 else // new
28     Compute estimation
29     Add $inst$ to $Buffer$
30 end if
```

For both Estimates Algorithm (EA) and Groups Algorithm (GA), we measure the average time needed to process each update. For completeness, we have also implemented the Naive approach and compare the results against it. Since the implementations are multi-threaded, multiple rankings are updated in parallel in each update. To isolate the gains achieved by the usage of our approaches from the gains due to the multi-threaded implementation, the runtime in each update is the one that would have been needed if the rankings were updated sequentially. We also measure the number of queries executed in each approach. We measure and report the number of Verification Queries and the number Buffer Reset Queries separately.

In all experiments, we performed 30,000 updates. The value that is added in each update varies between 1 and 50.

EA and GA were tested under different configurations varying the gap between the worst score of any instance in the top-k instances and the basic score, and the number of elements stored in the Buffer, i.e. the size of the Buffer. For the gap we tested the values 100% and 200%. A value of 100% means that the gap is at least equal to the maximum value that can be added in any update, i.e. 50. Similarly a value of 200% means that the gap is at least two times the maximum value that can be added in any update, i.e. 100. The Buffer size in each ranking was set to 100, 500, 1000, 5000 and 10000 elements.

We created two groups of updates. In the first group, each update affects (i.e. increases the score) a random entity. In the second group, the updates follow the 80-20 rule. According to this rule, 80% of the updates affect 20% of the instances. The goal of using two different sets of updates is to get an initial understanding of the kind of workloads our methods are more effective.

6.3.1 Updates following the 80-20 rule

In the plot of Figure 6.3, it is shown the change in the number of executed queries as the size of the Buffer and the gap change. As expected, the number of the Buffer Reset Queries (left plot) decreases with the increase of the Buffer size. GA executes fewer Buffer Reset Queries compared to EA, proving that forwarding the results to the rankings lying in lower levels in the lattice decreases indeed the number of queries. The difference is more prominent for the smaller Buffer size. For gap 200%, the number of Buffer Reset Queries increases for both EA and GA. Having a bigger gap results in fewer estimated scores qualifying for the top-k instances, thus more instances with estimated scores are added to the Buffer and the Buffer becomes full more often. In the right plot of Figure 6.3, the number of executed Verification Queries is shown. This plot verifies that the estimated scores of the instances qualify for the top-k results more often when the gap is smaller (one can hardly see the bars for gap 200%). As it is the case for the Buffer Reset Queries, GA executes also fewer Verification Queries compared to EA.

In the plot of Figure 6.4, it is shown the average time needed to process each update. For gap 200%, the fact that our rankings have this special relation does not seem to be important, since both EA and GA have the (almost) same performance. For gap set to 100%, GA is faster for small Buffer size but gets slower as the Buffer size increases. This happens because an increase on the Buffer size results in a decrease in the number of Buffer Reset Queries which means that GA cannot benefit much from re-using the results of these queries to remove instances from the Buffers of other rankings. Additionally, the Verification Queries take so little time to execute that the effort made to exchange the results between the rankings and compute the scores processing the received tuples overtakes the benefit of avoiding the execution of Verification Queries. However, in a setting where the execution of Verification Queries is very slow, the GA algorithm is expected to achieve better performance.

For the Naive Approach, 239985 Verification Queries are executed and the

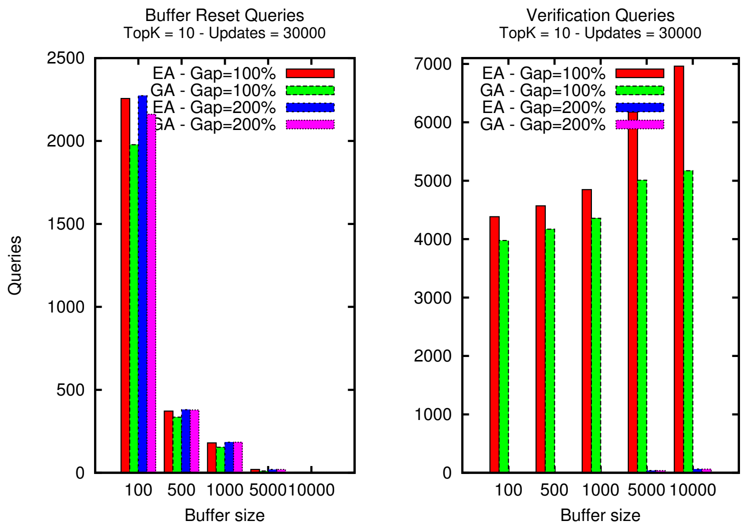


Figure 6.3: Buffer Reset and Verification Queries Queries (80-20 Updates)

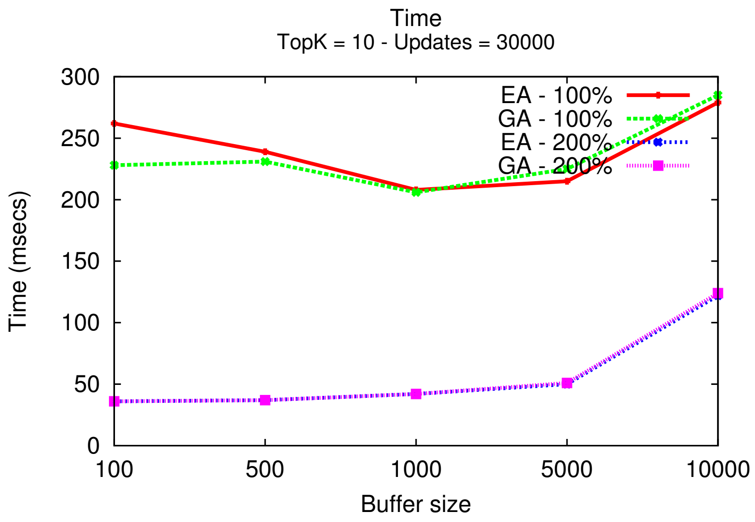


Figure 6.4: Runtime per update (80-20 Updates)

average time needed to process each update is more than 4 secs. These results are not included in the plots because they shadow the differences between the Estimates and Groups algorithms.

6.3.2 Random Updates

In the plots of Figure 6.5, it is shown the number of queries executed when the updates are random. Compared to the number of Verification Queries observed in the 80-20 updates, in this case there are less Verification Queries executed. Random updates are less likely to affect the same instance multiple times before the Buffer is reset. Thus, it is less likely an estimated score to grow enough to need the execution of a Verification Query.

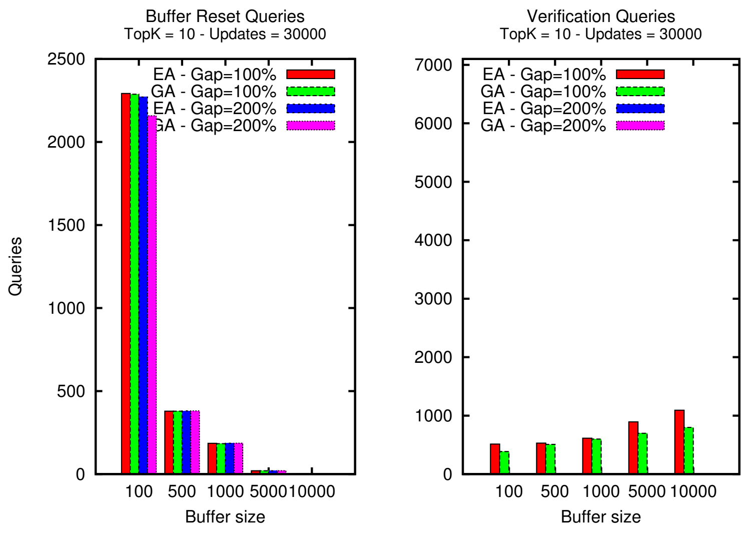


Figure 6.5: Buffer Reset and Verification Queries (Random Updates)

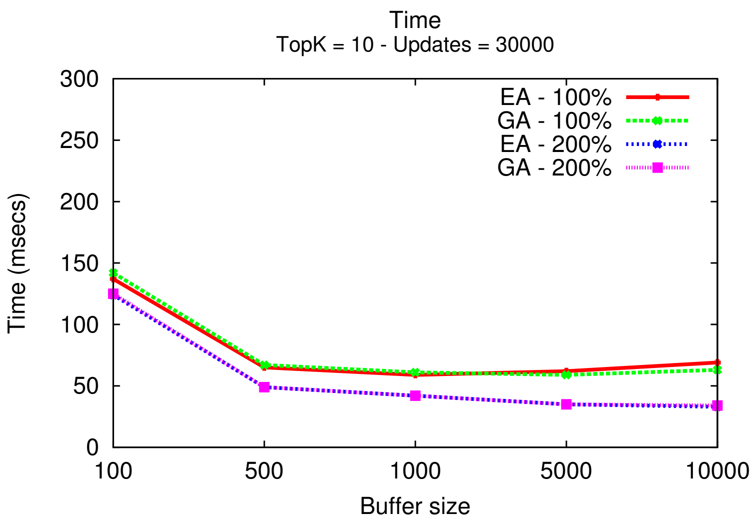


Figure 6.6: Runtime per update (Random Updates)

In the plot of Figure 6.6, it is shown the average time needed to process each update. What is interesting to observe is that the time needed to process each update when the updates are random either decreases constantly as the Buffer size increases or has a slight increase for very big Buffer sizes. On the contrary, in the 80-20 case, for bigger sizes of Buffer there is an acute increase in the runtime. As before, the reason is that random updates have lower probability to affect the same instance twice before the Buffer is reset. Hence, keeping an instance in memory longer (bigger sizes of the Buffer) does not cause its estimated score to become big enough to qualify for the top-k results and to cause a Verification Query. So, as long as the number of instances in the Buffer does not cause the Buffer Reset Query to become very slow, increasing its size can only be beneficial, with respect to time. On the other hand, in the 80-20 updates, keeping instances longer in memory results in many Verification Queries which deteriorate the runtime.

For the Naive Approach, 239977 Verification Queries are executed and the average time needed to process each update is more than 4 secs. Again, these results are not included in the plots so that the differences between Estimates and Groups algorithms can be shown clearer.

6.3.3 Additional Instances

The benefit achieved in runtime is not for free. Both EA and GA should store additional instances in order to maintain the top-k results. The number of these instances depends on the selection of the gap and the size of the Buffer. The plot in Figure 6.7 shows the number of additional instances for the various sizes of the Buffer when the gap is set to 100% and 200%. From the plot, it is obvious that setting the gap to 200% has much more space overhead. Especially in the very specific rankings (those binding all three secondary attributes to some instance), doubling the gap may result in having even five times more instances. This happens because as more instances are considered, the score differences between the instances are becoming smaller. The line labeled *ALL* shows the maximum number of instances (grouped using the filtering attributes) existing in the database. We use it as a baseline to give a notion of the extra storage cost.

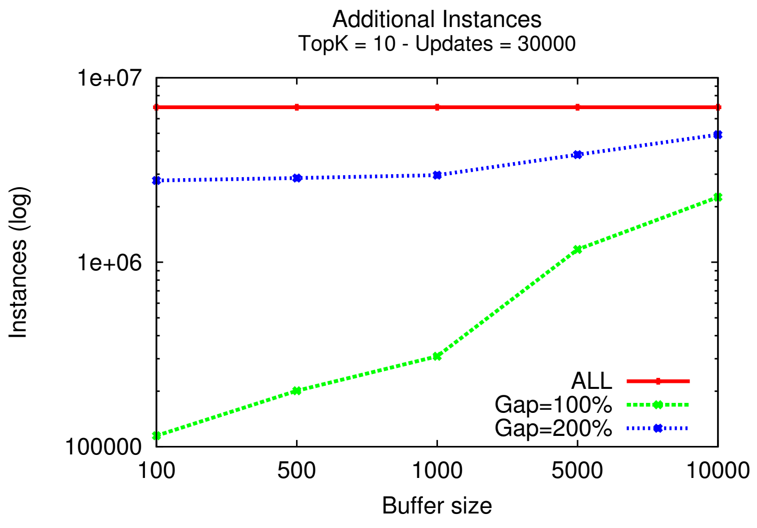


Figure 6.7: Extra instances stored (log scale)

6.4 Summary

In this chapter, we addressed the problem of maintaining top-k rankings in the presence of rapid updates arriving in an underlying database. Such a setup calls for methods that provide accurate (exact) top-k rankings while limiting the communication with the database itself. We presented two algorithms to solve that maintenance problem. The algorithms are centred around computing score estimates for previously unseen instances and leveraging containment relations for results recycling. Our experimental evaluation proves the usefulness of our approach showing a great reduction in the number of queries executed in the context of the top-k results maintenance. Apart from the benefits in the runtime, a reduction in the number of Verification Queries and Buffer Reset Queries allows the system to use its resources for other purposes, e.g. the execution of ad-hoc queries.

Chapter 7

Conclusions and Future Work

In this thesis, we examined the problem of event identification in big and dynamic data. Our main focus was on data over Web 2.0 sources like Twitter. An interesting characteristic of the data obtained in such scenarios is that each post is usually accompanied by a set of tags that indicate the topic of it. We defined a correlation measure over these tagsets that reflects how strongly correlated the tags are. We proposed an approach that uses correlation values measured during a small number of past time points to predict the correlation value of the current time point. We defined a topic to be emergent if its measured correlation value is greater than its predicted correlation value. To the emergent topics identified during the current time point we considered in addition emerging topics identified in the past. These topics were also presented to the users with an interestingness score dampened by some factor depending on the distance of the time point the topics were identified to the current time point. We conducted a user study that proved the ability of our approach to identify interesting and novel topics with quality three times higher to that of the topics provided by a state of the art approach. In some cases, the topics were identified before any traditional news portal had reported them. Additionally, we proposed an efficient implementation over a distributed stream processor. We proposed a number of algorithms, inspired by graph theory and the budgeted maximum coverage problem, that allow partitioning the tags to a number of machines based on their co-occurrences. The machines compute the correlations of tags in parallel based on the sets of tags they have been assigned. We theoretically examined several aspects of the data that can affect the performance of the partitioning algorithms.

A possible extension to our approach would provide personalised results to the users. For example, a user could create a profile and register to specific tags or broad categories. Only events related to the user preferences would be presented to him. Another possible extension could be to enrich the proposed topology with more functionality that would eliminate the problem caused by the repartitions when tagsets are assigned to different machines before and after the partitioning. Furthermore, a more close study of the alternative topology could be performed in order to obtain conclusive evidence on whether this topology is indeed suitable, and possibly better, for a dataset with the dynamic characteristics of Twitter. Finally, datasets with different dynamics to those found in Twitter could be used to test the usefulness of the proposed topology to a broader set of applications.

In addition to Web 2.0 data, we studied data stored in database systems. We considered data summarised using top-k rankings and defined an event to be any change in the elements within the ranking. To identify such events, we kept the top-k rankings up-to-date in the presence of continuous insertions that affected the underlying database. We presented two algorithms that maintain the top-k rankings combining techniques from the multi-query optimisation and view maintenance areas. The main objective of them is to minimise the interaction with the underlying database. The experimental results provided useful insights on the impact of the various parameters in the effectiveness of the methods. Trading the performance of the algorithms and the quality of the results could be an extension of the methods we proposed. More precisely, one could model the score distribution in the tail of rankings and use it to compute more realistic estimations for the scores of the unseen results, at the risk of introducing errors to the top-k rankings.

List of Figures

List of Algorithms

List of Tables

Bibliography

[ACD$^+$08] 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.

[AHWY03] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for clustering evolving data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 81–92, Berlin, Germany, 2003. VLDB Endowment.

[AM13a] 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.

[AM13b] Foteini Alvanaki and Sebastian Michel. A thin monitoring layer for top-k aggregation queries over a database. In *Proceedings of the 7th International Workshop on Ranking in Databases*, DBRank '13, pages 31–36, Riva del Garda, Italy, 2013. ACM.

[AM14] 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.

[AMRW12] Foteini Alvanaki, Sebastian Michel, Krithi Ramamritham, and Gerhard Weikum. See what's enblogue: Real-time emergent topic identification in social media. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 336–347, Berlin, Germany, 2012. ACM.

[APL98] 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.

[ASRW11] Foteini Alvanaki, Michel Sebastian, Krithi Ramamritham, and Gerhard Weikum. Enblogue: Emergent topic detection in web 2.0 streams. In *Proceedings of the 2011 ACM International Conference on Management of Data*, SIGMOD '11, pages 1271–1274, Athens, Greece, 2011. ACM.

[BBD$^+$02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, Madison, Wisconsin, 2002. ACM.

[BHKL06] 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.

[BJS09] K. Burton, A. Java, and I. Soboroff. The ICWSM 2009 Spinn3r dataset. In *Proceedings of the Third Annual Conference on Weblogs and Social Media*, ICWSM '09, San Jose, California, USA, 2009.

[BK07] Nilesh Bansal and Nick Koudas. Blogscope: A system for online analysis of high volume text streams. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1410–1413, Vienna, Austria, 2007. VLDB Endowment.

[BL08] G. Betti and A. Lemmi. *Advances on Income Inequality and Concentration Measures.* Routledge Frontiers of Political Economy. Taylor & Francis, 2008.

[BLT86] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently updating materialized views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 61–71, Washington, D.C., USA, 1986. ACM.

[BM91] Robert S. Boyer and J. Strother Moore. Mjrty: A fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, volume 1 of *Automated Reasoning Series*, pages 105–117. Springer Netherlands, 1991.

[BN09] A. Boggess and F.J. Narcowich. *A First Course in Wavelets with Fourier Analysis.* Wiley, 2009.

[BNG09] Hila Becker, Mor Naaman, and Luis Gravano. Event identification in social media. In *12th ACM SIGMOD Workshop on the Web and Databases*, WebDB 09, Providence, Rhode Island, USA, 2009.

[Bri88] E. Oran Brigham. *The Fast Fourier Transform and Its Applications.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[Bro63] Robert Goodell Brown. *Smoothing, forecasting and prediction of discrete time series.* Prentice-Hall quantitative methods series. Prentice-Hall, Englewood Cliffs, NJ, USA, 1963.

[CG99] Surajit Chaudhuri and Luis Gravano. Evaluating top-k selection queries. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 397–410, Edinburgh, Scotland, UK, 1999. Morgan Kaufmann Publishers Inc.

[CGM04] Surajit Chaudhuri, Luis Gravano, and Amelie Marian. Optimizing top-k selection queries over multimedia repositories. *IEEE Transactions on Knowledge and Data Engineering*, 16(8):992–1009, August 2004.

[CH08] Graham Cormode and Marios Hadjieleftheriou. Finding frequent items in data streams. *Proceedings of the Very Large Data Bases Endowment*, 1(2):1530–1541, August 2008.

[Che52] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sums of observations. *Annals of Mathematical Statistics*, 23:409–507, 1952.

[CJSS03] 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.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, 3rd edition, 2009.

[CMN99] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. On random sampling over joins. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 263–274, Philadelphia, Pennsylvania, USA, 1999. ACM.

[CSA05] Nauman A. Chaudhry, Kevin Shaw, and Mahdi Abdelguerfi, editors. *Stream Data Management*, volume 30 of *Advances in Database Systems.* Springer, 2005.

[Dek86] J. C. E. Dekker. The inclusion-exclusion principle for finitely many isolated sets. *The Journal of Symbolic Logic*, 51:435–447, June 1986.

[DH00] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, pages 71–80, Boston, Massachusetts, USA, 2000. ACM.

[DSJY11] Anish Das Sarma, Alpa Jain, and Cong Yu. Dynamic relationship and event discovery. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, WSDM '11, pages 207–216, Hong Kong, China, 2011. ACM.

[EK13] Milad Eftekhar and Nick Koudas. Partitioning and ranking tagged data sources. *Proceedings of the Very Large Data Bases Endowment*, 6(4):229–240, February 2013.

[ER60] Paul Erdős and Alfréd Rényi. On the evolution of random graphs. In *Publication of the Mathematical Institute of the Hungarian Academy of Sciences*, pages 17–61, 1960.

[FLN01] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '01, pages 102–113, Santa Barbara, California, USA, 2001. ACM.

[FM83] Philippe Flajolet and G. Nigel Martin. Probabilistic counting. In *24th Annual Symposium on Foundations of Computer Science*, FOCS '83, pages 76–82, Tucson, Arizona, USA, 1983. IEEE Computer Society.

[GBK00] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Optimizing multi-feature queries for image databases. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 419–428, Cairo, Egypt, 2000. Morgan Kaufmann Publishers Inc.

[GDH04] 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.

[GJM96] Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. Data integration using self-maintainable views. In *Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '96, pages 140–144, Avignon, France, 1996. Springer-Verlag.

[GKL08] Vicenç Gómez, Andreas Kaltenbrunner, and Vicente López. 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.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 9th ACM International Conference on Management of Data*, SIGMOD '84, pages 47–57, Boston, Massachusetts, USA, 1984. ACM Press.

[GZK10] Mohamed Medhat Gaber, Arkady B. Zaslavsky, and Shonali Krishnaswamy. Data stream mining. In Oded Maimon and Lior Rokach, editors, *Data Mining and Knowledge Discovery Handbook*, pages 759–787. Springer, 2010.

[HKG$^+$06] Maria Halkidi, Vana Kalogeraki, Dimitrios Gunopulos, Dimitris Papadopoulos, Demetrios Zeinalipour-Yazti, and Michail Vlachos. Efficient online state tracking using sensor networks. *2013 IEEE 14th International Conference on Mobile Data Management (MDM)*, 0:24, 2006.

[HM03] Sven Helmer and Guido Moerkotte. A performance study of four index structures for set-valued attributes of low cardinality. *The International Journal on Very Large Data Bases (VLDB)*, 12(3):244–261, October 2003.

[HMA10] Parisa Haghani, Sebastian Michel, and Karl Aberer. The gist of everything new: Personalized top-k processing over web 2.0 streams. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, CIKM '10, pages 489–498, Toronto, ON, Canada, 2010. ACM.

[IAE04] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. *The International Journal on Very Large Data Bases (VLDB)*, 13(3):207–221, September 2004.

[IBS08] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, 40(4):11:1–11:58, October 2008.

[IP99] Yannis E. Ioannidis and Viswanath Poosala. Histogram-based approximation of set-valued query-answers. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 174–185, Edinburgh, Scotland, 1999. Morgan Kaufmann Publishers Inc.

[IPF$^+$07] Marios Iliofotou, Prashanth Pappu, Michalis Faloutsos, Michael Mitzenmacher, Sumeet Singh, and George Varghese. Network monitoring using traffic dispersion graphs (tdgs). In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, pages 315–320, San Diego, California, USA, 2007. ACM.

[Jac12] Paul Jaccard. The distribution of the flora in the alpine zone. *New Phytologist*, 11(2):37–50, February 1912.

[JK02] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, October 2002.

[KMBS11] Shiva Prasad Kasiviswanathan, Prem Melville, Arindam Banerjee, and Vikas Sindhwani. Emerging topic detection using dictionary learning. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 745–754, Glasgow, Scotland, UK, 2011. ACM.

[KMN99] Samir Khuller, Anna Moss, and Joseph (Seffi) Naor. The budgeted maximum coverage problem. *Information Processing Letters*, 70(1):39–45, April 1999.

[KS09] Jürgen Krämer and Bernhard Seeger. Semantics and implementation of continuous sliding window queries over data streams. *ACM Transactions on Database Systems (TODS)*, 34(1):4:1–4:49, April 2009.

[KSP03] Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems (TODS)*, 28(1):51–55, March 2003.

[LCCCI06] Chengkai Li, Kevin Chen-Chuan Chang, and Ihab F. Ilyas. Supporting ad-hoc ranking aggregates. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 61–72, Chicago, IL, USA, 2006. ACM.

[LRB09] 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.

[MAEA05] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the 10th International Conference on Database Theory*, ICDT'05, pages 398–412, Edinburgh, UK, 2005. Springer-Verlag.

[MBP06] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. Continuous monitoring of top-k queries over sliding windows. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 635–646, Chicago, IL, USA, 2006. ACM.

[MFHH02] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: A tiny aggregation service for ad-hoc sensor networks. *ACM SIGOPS Operating Systems Review*, 36(SI):131–146, December 2002.

[MK10] Michael Mathioudakis and Nick Koudas. TwitterMonitor: Trend detection over the twitter stream. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1155–1158, Indianapolis, Indiana, USA, 2010. ACM.

[MM02] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 346–357, Hong Kong, China, 2002. VLDB Endowment.

[MP03] Junshui Ma and Simon Perkins. Online novelty detection on temporal sequences. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 613–618, Washington, D.C., 2003. ACM.

[MRS09] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze. *Introduction to Information Retrieval.* Cambridge University Press, 2009.

[MRSR01] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 307–318, Santa Barbara, California, USA, 2001. ACM.

[OMM$^+$02] Liadan O'Callaghan, Adam Meyerson, Rajeev Motwani, Nina Mishra, and Sudipto Guha. Streaming-data algorithms for high-quality clustering. In *Proceedings of the 18th International Conference on Data Engineering*, ICDE '02, pages 685–694, San Jose, California, USA, 2002. IEEE Computer Society.

[PP10] 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.

[PPKG03] Themistoklis Palpanas, Dimitris Papadopoulos, Vana Kalogeraki, and Dimitrios Gunopulos. Distributed deviation detection in sensor networks. *SIGMOD Record*, 32(4):77–82, December 2003.

[PPP11] Ana-Maria Popescu, Marco Pennacchiotti, and Deepa Paranjpe. Extracting events and event descriptions from twitter. In *Proceedings of the 20th International Conference Companion on World Wide Web*, WWW '11, pages 105–106, Hyderabad, India, 2011. ACM.

[PSCP02] Themistoklis Palpanas, Richard Sidle, Roberta Cochrane, and Hamid Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 802–813, Hong Kong, China, 2002. VLDB Endowment.

[Raf99] Davood Rafiei. On similarity-based queries for time series data. In *Proceedings of the 18th International Conference on Data Engineering*, ICDE '99, pages 410–417. IEEE Computer Society, 1999.

[RJN12] João B. Rocha-Junior and Kjetil Nørvåg. Top-k spatial keyword queries on road networks. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 168–179, Berlin, Germany, 2012. ACM.

[RJVDN10] João B. Rocha-Junior, Akrivi Vlachou, Christos Doulkeridis, and Kjetil Nørvåg. Efficient processing of top-k spatial preference queries. *Proceedings of the Very Large Data Bases Endowment*, 4(2):93–104, November 2010.

[RMEC12] Alan Ritter, Mausam, Oren Etzioni, and Sam Clark. Open domain event extraction from twitter. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 1104–1112, Beijing, China, 2012. ACM.

[SFD$^+$10] Bharath Sriram, Dave Fuhry, Engin Demir, Hakan Ferhatosmanoglu, and Murat Demirbas. Short text classification in twitter to improve information filtering. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '10, pages 841–842, Geneva, Switzerland, 2010. ACM.

[SHM09] 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.

[SKM99] Sunil Samtani, Vijay Kumar, and Mukesh K. Mohania. Self maintenance of multiple views in data warehousing. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, CIKM '99, pages 292–299, Kansas City, Missouri, USA, 1999. ACM.

[SS12] Ankan Saha and Vikas Sindhwani. Learning evolving and emerging topics in social media: A dynamic nmf approach with temporal regularization. In *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining*, WSDM '12, pages 693–702, Seattle, Washington, USA, 2012. ACM.

[TCY03] Wei-Guang Teng, Ming-Syan Chen, and Philip S. Yu. A regression-based temporal pattern mining scheme for data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases*, volume 29 of *VLDB '03*, pages 93–104, Berlin, Germany, 2003. VLDB Endowment.

[TcZ$^+$03] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases*, volume 29 of *VLDB '03*, pages 309–320, Berlin, Germany, 2003. VLDB Endowment.

[TTY11] Toshimitsu Takahashi, Ryota Tomioka, and Kenji Yamanishi. Discovering emerging topics in social streams via link anomaly detection. In *Proceedings of the 2011 IEEE 11th International Conference on Data Mining*, ICDM '11, pages 1230–1235, Vancouver, British Columbia, Canada, 2011. IEEE Computer Society.

[Tur01] Peter D. Turney. Mining the web for synonyms: PMI-IR versus LSA on TOEFL. In *Proceedings of the 12th European Conference on Machine Learning*, EMCL '01, pages 491–502, Freiburg, Germany, 2001. Springer-Verlag.

[UYTI11] Yasuhiro Urabe, Kenji Yamanishi, Ryota Tomioka, and Hiroki Iwai. Real-time change-point detection using sequentially discounting normalized maximum likelihood coding. In *Proceedings of the 15th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining - Volume Part II*, PAKDD'11, pages 185–197, Shenzhen, China, 2011. Springer-Verlag.

[Was10] Larry Wasserman. *All of Statistics: A Concise Course in Statistical Inference.* Springer Publishing Company, Incorporated, 2010.

[WFYH03] Haixun Wang, Wei Fan, Philip S. Yu, and Jiawei Han. Mining concept-drifting data streams using ensemble classifiers. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 226–235, Washington, D.C., USA, 2003. ACM.

[WH07] F. Wu and B. A. Huberman. Novelty and collective attention. *Proceedings of the National Academy of Sciences USA*, 104(45):17599–17601, 2007.

[WMS12] Andreas Weiler, Svetlana Mansmann, and Marc H. Scholl. Towards an advanced system for real-time event detection in high-volume data streams. In *Proceedings of the 5th Ph.D. Workshop on Information and Knowledge*, PIKM '12, pages 87–90, Maui, Hawaii, USA, 2012. ACM.

[WOOO11] Kazufumi Watanabe, Masanao Ochi, Makoto Okabe, and Rikio Onai. Jasmine: A real-time local-event detection system based on geolocation information propagated to microblogs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 2541–2544, Glasgow, Scotland, UK, 2011. ACM.

[YCLZ04] Jeffery Xu Yu, Zhihong Chong, Hongjun Lu, and Aoying Zhou. False positive or false negative: Mining frequent itemsets from high speed transactional data streams. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 204–215, Toronto, ON, Canada, 2004. VLDB Endowment.

[YMH08] Man Lung Yiu, Nikos Mamoulis, and Vagelis Hristidis. Extracting k most important groups from data efficiently. *Data and Knowledge Engineering*, 66(2):289–310, August 2008.

[YSJ$^+$00] Byoung-Kee Yi, Nikolaos Sidiropoulos, Theodore Johnson, H. V. Jagadish, Christos Faloutsos, and Alexandros Biliris. Online data mining for co-evolving time sequences. In *Proceedings of the 16th*

International Conference on Data Engineering, ICDE '00, pages 13–22, San Diego, California, USA, 2000. IEEE Computer Society.

[YYY$^+$03] Ke Yi, Hai Yu, Jun Yang, Gangqiang Xia, and Yuguo Chen. Efficient maintenance of materialized top-k views. In *Proceedings of the 19th International Conference on Data Engineering*, ICDE '00, pages 189–200, Bangalore, India, 2003. IEEE Computer Society.

[ZGMHW95] Yue Zhuge, Héctor García-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *Proceedings of the 1995 ACM International Conference on Management of Data*, SIGMOD '95, pages 316–327, San Jose, California, USA, 1995. ACM.

[ZHC$^+$06] Zhen Zhang, Seung-won Hwang, Kevin Chen-Chuan Chang, Min Wang, Christian A. Lang, and Yuan-chi Chang. Boolean + ranking: Querying a database by k-constrained optimization. In *Proceedings of the 32nd ACM International Conference on Management of Data*, SIGMOD '06, pages 359–370, Chicago, IL, USA, 2006. ACM.

[ZKOS05] Rui Zhang, Nick Koudas, Beng Chin Ooi, and Divesh Srivastava. Multiple aggregations over data streams. In *Proceedings of the 2005 ACM International Conference on Management of Data*, SIGMOD '05, pages 299–310, Baltimore, Maryland, USA, 2005. ACM.

[ZM06] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), July 2006.

[ZMC07] 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.

[ZS02] Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 358–369, Hong Kong, China, 2002. VLDB Endowment.