IMT Institute for Advanced Studies

Lucca, Italy

Big Data and the Web: Algorithms for Data Intensive Scalable Computing

PhD Program in Computer Science and Engineering

XXIV Cycle

By

Gianmarco De Francisci Morales

2012

The dissertation of Gianmarco De Francisci Morales is approved.

Program Coordinator: Prof. Rocco De Nicola, IMT Lucca

Supervisor: Dott. Claudio Lucchese, ISTI-CNR Pisa

Co-Supervisor: Dott. Ranieri Baraglia, ISTI-CNR Pisa

Tutor: Dott. Leonardo Badia, University of Padova

The dissertation of Gianmarco De Francisci Morales has been reviewed by:

Aristides Gionis, Yahoo! Research Barcelona

Iadh Ounis, University of Glasgow

IMT Institute for Advanced Studies, Lucca

2012

Where is the wisdom we
have lost in knowledge?
Where is the knowledge we
have lost in information?

To my mother, for her unconditional love
and support throughout all these years.

Acknowledgements

I owe my deepest and earnest gratitude to my supervisor, Claudio Lucchese, who shepherded me towards this goal with great wisdom and everlasting patience.

I am grateful to all my co-authors, without whom this work would have been impossible. A separate acknowledgement goes to Aris Gionis for his precious advice, constant presence and exemplary guidance.

I thank all the friends and colleagues in Lucca with whom I shared the experience of being a Ph.D. student, my Lab in Pisa that accompanied me through this journey, and all the people in Barcelona that helped me feel like at home.

Thanks to my family and to everyone who believed in me.

Contents

List of Figures

List of Tables

Publications

1. G. De Francisci Morales, A. Gionis, C. Lucchese, “From Chatter to Headlines: Harnessing the Real-Time Web for Personalized News Recommendations", WSDM’12, 5th ACM International Conference on Web Search and Data Mining, Seattle, 2012, pp. 153-162.
2. G. De Francisci Morales, A. Gionis, M. Sozio, “Social Content Matching in MapReduce”, PVLDB, Proceedings of the VLDB Endowment, 4(7):460-469, 2011.
3. R. Baraglia, G. De Francisci Morales, C. Lucchese, “Document Similarity Self-Join with MapReduce”, ICDM’10, 10th IEEE International Conference on Data Mining, Sydney, 2010, pp. 731-736.
4. G. De Francisci Morales, C. Lucchese, R. Baraglia, “Scaling Out All Pairs Similarity Search with MapReduce”, LSDS-IR’10, 8th Workshop on Large-Scale Distributed Systems for Information Retrieval, @SIGIR’10, Geneva, 2010, pp. 25-30.
5. G. De Francisci Morales, C. Lucchese, R. Baraglia, “Large-scale Data Analysis on the Cloud”, XXIV Convegno Annuale del CMG-Italia, Roma, 2010.

Presentations

1. G. De Francisci Morales, “Harnessing the Real-Time Web for Personalized News Recommendation”, Yahoo! Labs, Sunnyvale, 16 Februrary 2012.
2. G. De Francisci Morales, “Big Data and the Web: Algorithms for Data Intensive Scalable Computing”, WSDM’12, Seattle, 8 Februrary 2012.
3. G. De Francisci Morales, “Social Content Matching in MapReduce”, Yahoo! Research, Barcelona, 10 March 2011.
4. G. De Francisci Morales, “Cloud Computing for Large Scale Data Analysis”, Yahoo! Research, Barcelona, 2 December 2010.
5. G. De Francisci Morales, “Scaling Out All Pairs Similarity Search with MapReduce”, Summer School on Social Networks, Lipari, 6 July 2010.
6. G. De Francisci Morales, “How to Survive the Data Deluge: Petabyte Scale Cloud Computing”, ISTI-CNR, Pisa, 18 January 2010.

Abstract

This thesis explores the problem of large scale Web mining by using Data Intensive Scalable Computing (DISC) systems. Web mining aims to extract useful information and models from data on the Web, the largest repository ever created. DISC systems are an emerging technology for processing huge datasets in parallel on large computer clusters.

Challenges arise from both themes of research. The Web is heterogeneous: data lives in various formats that are best modeled in different ways. Effectively extracting information requires careful design of algorithms for specific categories of data. The Web is huge, but DISC systems offer a platform for building scalable solutions. However, they provide restricted computing primitives for the sake of performance. Efficiently harnessing the power of parallelism offered by DISC systems involves rethinking traditional algorithms.

This thesis tackles three classical problems in Web mining. First we propose a novel solution to finding similar items in a bag of Web pages. Second we consider how to effectively distribute content from Web 2.0 to users via graph matching. Third we show how to harness the streams from the real-time Web to suggest news articles. Our main contribution lies in rethinking these problems in the context of massive scale Web mining, and in designing efficient MapReduce and streaming algorithms to solve these problems on DISC systems.

Chapter 1

Introduction

An incredible *"data deluge"* is currently drowning the world. Data sources are everywhere, from Web 2.0 and user-generated content to large scientific experiments, from social networks to wireless sensor networks. This massive amount of data is a valuable asset in our information society.

Data analysis is the process of inspecting data in order to extract useful information. Decision makers commonly use this information to drive their choices. The quality of the information extracted by this process greatly benefits from the availability of extensive datasets.

The Web is the biggest and fastest growing data repository in the world. Its size and diversity make it the ideal resource to mine for useful information. Data on the Web is very diverse in both content and format. Consequently, algorithms for Web mining need to take into account the specific characteristics of the data to be efficient.

As we enter the *"petabyte age"*, traditional approaches for data analysis begin to show their limits. Commonly available data analysis tools are unable to keep up with the increase in size, diversity and rate of change of the Web. Data Intensive Scalable Computing is an emerging alternative technology for large scale data analysis. DISC systems combine both storage and computing in a distributed and virtualized manner. These systems are built to scale to thousands of computers, and focus on fault tolerance, cost effectiveness and ease of use.

1.1 The Data Deluge

How would you sort 1GB of data? Today's computers have enough memory to keep this quantity of data, so any optimal in-memory algorithm will suffice. What if you had to sort 100 GB of data? Even if systems with more than 100 GB of memory exist, they are by no means common or cheap. So the best solution is to use a disk based sorting algorithm. However, what if you had 10 TB of data to sort? At a transfer rate of about 100 MB/s for a normal disk it would take more than one day to make a single pass over the dataset. In this case the bandwidth between memory and disk is the bottleneck. In any case, today's disks are usually 1 to 2 TB in size, which means that just to hold the data we need multiple disks. In order to obtain acceptable completion times, we also need to use multiple computers and a parallel algorithm.

This example illustrates a general point: the same problem at different scales needs radically different solutions. In many cases we even need change the model we use to reason about the problem because the simplifying assumptions made by the models do not hold at every scale. Citing Box and Draper (1986) "Essentially, all models are wrong, but some are useful", and arguably "most of them do not scale".

Currently, an incredible *"data deluge"* is drowning the world. The amount of data we need to sift through every day is enormous. For instance the results of a search engine query are so many that we are not able to examine all of them, and indeed the competition now focuses the top ten results. This is just an example of a more general trend.

The issues raised by large datasets in the context of analytical applications are becoming ever more important as we enter the so-called *"petabyte age"*. Figure 1.1 shows the sizes of the datasets for problems we currently face (Anderson, 2008). The datasets are orders of magnitude greater than what fits on a single hard drive, and their management poses a serious challenge. Web companies are currently facing this issue, and striving to find efficient solutions. The ability to manage and analyze more data is a distinct competitive advantage for them. This issue has been labeled in various ways: petabyte scale, Web scale or *"big data"*.

Figure 1.1: The petabyte age.

But how do we define *"big data"*? The definition is of course relative and evolves in time as technology progresses. Indeed, thirty years ago one terabyte would be considered enormous, while today we are commonly dealing with such quantity of data.

Gartner (2011) puts the focus not only on size but on three different dimensions of growth for data, the 3V: *Volume*, *Variety* and *Velocity*. The data is surely growing in size, but also in complexity as it shows up in different formats and from different sources that are hard to integrate, and in dynamicity as it arrives continuously, changes rapidly and needs to be processed as fast as possible.

Loukides (2010) offers a different point of view by saying that big data is "when the size of the data itself becomes part of the problem" and "traditional techniques for working with data run out of steam".

Along the same lines, Jacobs (2009) states that big data is "data whose size forces us to look beyond the tried-and-true methods that are prevalent at that time". This means that we can call *big* an amount of data that forces us to use or create innovative methodologies.

We can think that the intrinsic characteristics of the object to be analyzed demand modifications to traditional data managing procedures.

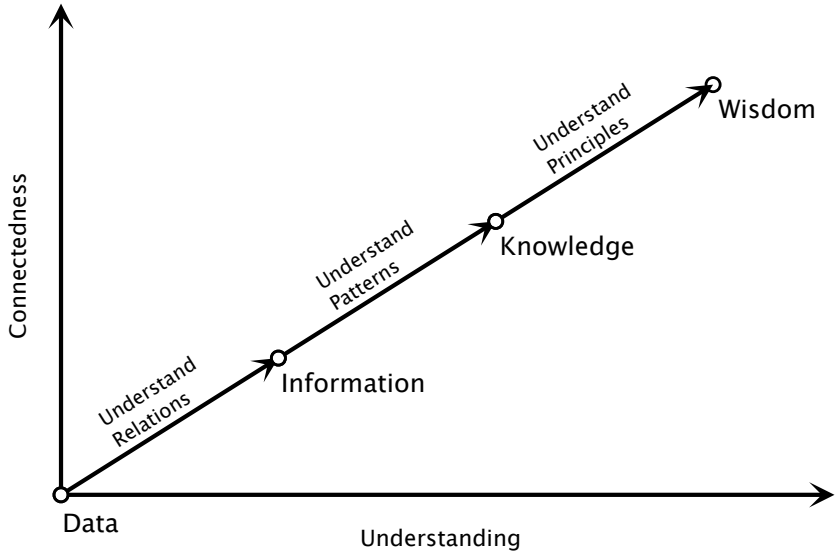

Figure 1.2: Data Information Knowledge Wisdom hierarchy.

Alternatively, we can take the point of view of the subject who needs to manage the data. The emphasis is thus on user requirements such as throughput and latency. In either case, all the previous definitions hint to the fact that big data is a driver for research.

But why are we interested in data? It is common belief that data without a model is just noise. Models are used to describe salient features in the data, which can be extracted via *data mining*. Figure 1.2 depicts the popular Data Information Knowledge Wisdom (DIKW) hierarchy (Rowley, 2007). In this hierarchy data stands at the lowest level and bears the smallest level of *understanding*. Data needs to be processed and condensed into more connected forms in order to be useful for event comprehension and decision making. Information, knowledge and wisdom are these forms of understanding. Relations and patterns that allow to gain deeper awareness of the process that generated the data, and principles that can guide future decisions.

For data mining, the scaling up of datasets is a *"double edged"* sword. On the one hand, it is an opportunity because *"no data is like more data"*. Deeper insights are possible when more data is available (Halevy et al., 2009). Oh the other hand, it is a challenge. Current methodologies are often not suitable to handle huge datasets, so new solutions are needed.

The large availability of data potentially enables more powerful analysis and unexpected outcomes. For example, Google Flu Trends can detect regional flu outbreaks up to ten days faster than the Center for Disease Control and Prevention by analyzing the volume of flu-related queries to the Web search engine (Ginsberg et al., 2008). Companies like IBM and Google are using large scale data to solve extremely challenging problems like avoiding traffic congestion, designing self-driving cars or understanding *Jeopardy* riddles (Loukides, 2011). Chapter 2 presents more examples of interesting large scale data analysis problems.

Data originates from a wide variety sources. Radio-Frequency Identification (RFID) tags and Global Positioning System (GPS) receivers are already spread all around us. Sensors like these produce petabytes of data just as a result of their sheer numbers, thus starting the so called *"industrial revolution of data"* (Hellerstein, 2008).

Scientific experiments are also a huge data source. The Large Hadron Collider at CERN is expected to generate around 50 TB of raw data per day. The Hubble telescope captured millions of astronomical images, each weighting hundreds of megabytes. Computational biology experiments like high-throughput genome sequencing produce large quantities of data that require extensive post-processing.

The focus of our work is directed to another massive source of data: the Web. The importance of the Web from the scientific, economical and political point of view has grown dramatically over the last ten years, so much that internet access has been declared a human right by the United Nations (La Rue, 2011). Web users produce vast amounts of text, audio and video contents in the Web 2.0. Relationships and tags in social networks create massive graphs spanning millions of vertexes and billions of edges. In the next section we highlight some of the opportunities and challenges found when mining the Web.

1.2 Mining the Web

The Web is easily the single largest publicly accessible data source in the world (Liu, 2007). The continuous usage of the Web has accelerated its growth. People and companies keep adding to the already enormous mass of pages already present.

In the last decade the Web has increased its importance to the point of becoming the center of our digital lives (Hammersley, 2011). People shop and read news on the Web, governments offer public services through it and enterprises develop Web marketing strategies. Investments in Web advertising have surpassed the ones in television and newspaper in most countries. This is a clear testament to the importance of the Web.

The estimated size of the indexable Web was at least 11.5 billion pages as of January 2005 (Gulli and Signorini, 2005). Today, the Web size is estimated between 50 and 100 billion pages and roughly doubling every eight months (Baeza-Yates and Ribeiro-Neto, 2011), faster than Moore's law. Furthermore, the Web has become infinite for practical purpose, as it is possible to generate an infinite number of dynamic pages. As a result, there is on the Web an abundance of data with growing value.

The value of this data lies in being representative of collective user behavior. It is our *digital footprint*. By analyzing a large amount of these traces it is possible to find common patterns, extract user models, make better predictions, build smarter products and gain a better understanding of the dynamics of human behavior. Enterprises have started to realize on which gold mine they are sitting on. Companies like Facebook and Twitter base their business model entirely on collecting user data.

Data on the Web is often produced as a byproduct of online activity of the users, and is sometimes referred to as *data exhaust*. This data is silently collected while the users are pursuing their own goal online, e.g. query logs from search engines, co-buying and co-visiting statistics from online shops, click through rates from news and advertisings, and so on.

This process of collecting data automatically can scale much further than traditional methods like polls and surveys. For example it is possible to monitor public interest and public opinion by analyzing collective

click behavior in news portals, references and sentiments in blogs and micro-blogs or query terms in search engines.

As another example, Yahoo! and Facebook (2011) are currently replicating the famous "small world" experiment ideated by Milgram. They are leveraging the social network created by Facebook users to test the "six degrees of separation" hypothesis on a planetary scale. The large number of users allows to address the critiques of selection and non-response bias made to the original experiment.

Let us now more precisely define Web mining. Web mining is the application of data mining techniques to discover patterns from the Web. According to the target of the analysis at hand, Web mining can be categoryzed into three different types: Web structure mining, Web content mining and Web usage mining (Liu, 2007).

Web structure mining mines the hyperlink structure of the Web using graph theory. For example, links are used by search engines to find important Web pages, or in social networks to discover communities of users who share common interests.

Web content mining analyzes Web page contents. Web content mining differs from traditional data and text mining mainly because of the semi-structured and multimedial nature of Web pages. For example, it is possible to automatically classify and cluster Web pages according to their topics but it is also possible to mine customer product reviews to discover consumer sentiments.

Web usage mining extracts information from user access patterns found in Web server logs, which record the pages visited by each user, and from search patterns found in query logs, which record the terms searched by each user. Web usage mining investigates what users are interested in on the Web.

Mining the Web is typically deemed highly promising and rewarding. However, it is by no means an easy task and there is a flip side of the coin: data found on the Web is extremely noisy.

The noise comes from two main sources. First, Web pages are complex and contain many pieces of information, e.g., the main content of the page, links, advertisements, images and scripts. For a particular application, only part of the information is useful and the rest is considered noise. Second, the Web is open to anyone and does not enforce any quality control of information. Consequently a large amount of information on the Web is of low quality, erroneous, or even misleading, e.g., automatically generated spam, content farms and dangling links. This applies also to Web server logs and Web search engine logs, where erratic behaviors, automatic crawling, spelling mistakes, spam queries and attacks introduce a large amount of noise.

The signal, i.e. the useful part of the information, is often buried under a pile of dirty, noisy and unrelated data. It is the duty of a data analyst to *"separate the wheat from the chaff"* by using sophisticated cleaning, pre-processing and mining techniques.

This challenge is further complicated by the sheer size of the data. Datasets coming from the Web are too large to handle using traditional systems. Storing, moving and managing them are complex tasks by themselves. For this reason a data analyst needs the help of powerful yet easy to use systems that abstract away the complex machinery needed to deliver the required performance. The goal of these systems is to reduce the time-to-insight by speeding up the design-prototype-test cycle in order to test a larger number of hypothesis, as detailed in Section 1.3.

1.2.1 Taxonomy of Web data

The Web is a very diverse place. It is an open platform where anybody can add his own contribution. Resultingly, information on the Web is heterogeneous. Almost any kind of information can be found on it, usually reproduced in a proliferation of different formats. As a consequence, the categories of data available on the Web are quite varied.

Data of all kinds exist on the Web: semi-structured Web pages, structured tables, unstructured texts, explicit and implicit links, and multimedia files (images, audios, and videos) just to name a few. A complete

classification of the categories of data on the Web is out of the scope of this thesis. However, we present next what we consider to be the most common and representative categories, the ones on which we focus our attention. Most of the Web fits one of these three categories:

Bags are unordered collections of items. The Web can be seen as a collections of documents when ignoring hyperlinks. Web sites that collect one specific kind of items (e.g. flickr or YouTube) can also be modeled as bags. The items in the bag are typically represented as sets, multisets or vectors. Most classical problems like similarity, clustering and frequent itemset mining are defined over bags.

Graphs are defined by a set of vertexes connected by a set of edges. The Web link structure and social networks fit in this category. Graph are an extremely flexible data model as almost anything can be seen as a graph. They can also be generated from predicates on a set of items (e.g. similarity graph, query flow graph). Graph algorithms like PageRank, community detection and matching are commonly employed to solve problems in Web and social network mining.

Streams are unbounded sequences of items ordered by time. Search queries and click streams are traditional examples, but streams are generated as well by news portals, micro-blogging services and real-time Web sites like twitter and "status updates" on social networks like Facebook, Google+ and LinkedIn. Differently from time series, Web streams are textual, multimedial or have rich metadata. Traditional stream mining problems are clustering, classification and estimation of frequency moments.

Each of these categories has its own characteristics and complexities. Bags of items include very large collections whose items can be analyzed independently in parallel. However this lack of structure can also complicate analysis as in the case of clustering and nearest neighbor search, where each item can be related to any other item in the bag.

In contrast, graphs have a well defined structure that limits the local relationships that need to be taken into account. For this reason local properties like degree distribution and clustering coefficient are very

easy to compute. However global properties such as diameter and girth generally require more complex iterative algorithms.

Finally, streams get continuously produced and a large part of their value is in their freshness. As such, they cannot be analyzed in batches and need to be processed as fast as possible in an online fashion.

For the reasons just described, the algorithms and the methodologies needed to analyze each category of data are quite different from each other. As detailed in Section 1.4, in this work we present three different algorithms for large scale data analysis, each one explicitly tailored for one of these categories of data.

1.3 Management of Data

Providing data for analysis is a problem that has been extensively studied. Many solutions exist but the traditional approach is to employ a Database Management System (DBMS) to store and manage the data.

Modern DBMS originate in the '70s, when Codd (1970) introduced the famous *relational* model that is still in use today. The model introduces the familiar concepts of tabular data, relation, normalization, primary key, relational algebra and so on.

The original purpose of DBMSs was to process transactions in business oriented processes, also known as Online Transaction Processing (OLTP). Queries were written in Structured Query Language (SQL) and run against data modeled in relational style. On the other hand, currently DBMSs are used in a wide range of different areas: besides OLTP, we have Online Analysis Processing (OLAP) applications like data warehousing and business intelligence, stream processing with continuous queries, text databases and much more (Stonebraker and Çetintemel, 2005). Furthermore, stored procedures are preferred over plain SQL for performance reasons. Given the shift and diversification of application fields, it is not a surprise that most existing DBMSs fail to meet today's high performance requirements (Stonebraker et al., 2007a,b).

High performance has always been a key issue in database research. There are usually two approaches to achieve it: *vertical* and *horizon-*

tal. The former is the simplest, and consists in adding resources (cores, memory, disks) to an existing system. If the resulting system is capable of taking advantage of the new resources it is said to scale up. The inherent limitation of this approach is that the single most powerful system available on earth could not suffice. The latter approach is more complex, and consists in adding new separate systems in parallel. The multiple systems are treated as a single logical unit. If the system achieves higher performance it is said to scale out. However, the result is a parallel system with all the hard problems of concurrency.

1.3.1 Parallelism

Typical parallel systems are divided into three categories according to their architecture: *shared memory*, *shared disk* or *shared nothing*. In the first category we find Symmetric Multi-Processors (SMPs) and large parallel machines. In the second one we find rack based solutions like Storage Area Network (SAN) or Network Attached Storage (NAS). The last category includes large commodity clusters interconnected by a local network and is deemed to be the most scalable (Stonebraker, 1986).

Parallel Database Management Systems (PDBMSs) (DeWitt and Gray, 1992) are the result of these considerations. They attempt to achieve high performance by leveraging parallelism. Almost all the designs of PDBMSs use the same basic dataflow pattern for query processing and horizontal partitioning of the tables on a cluster of shared nothing machines for data distribution (DeWitt et al., 1990).

Unfortunately, PDBMSs are very complex systems. They need fine tuning of many *"knobs"* and feature simplistic fault tolerance policies. In the end, they do not provide the user with adequate ease of installation and administration (the so called *"one button"* experience), and flexibility of use, e.g., poor support of User Defined Functions (UDFs).

To date, despite numerous claims about their scalability, PDBMSs have proven to be profitable only up to the tens or hundreds of nodes. It is legitimate to question whether this is the result of a fundamental theoretical problem in the parallel approach.

Parallelism has some well known limitations. Amdahl (1967) argued in favor of a single-processor approach to achieve high performance. Indeed, the famous *"Amdahl's law"* states that the parallel speedup of a program is inherently limited by the inverse of his *serial fraction*, the non parallelizable part of the program. His law also defines the concept of *strong scalability*, in which the *total* problem size is fixed. Equation 1.1 specifies Amdahl's law for N parallel processing units where r_s and r_p are the serial and parallel fraction of the program ($r_s + r_p = 1$)

$$SpeedUp(N) = \frac{1}{r_s + \frac{r_p}{N}} \tag{1.1}$$

Nevertheless, parallelism has a theoretical justification. Gustafson (1988) re-evaluated Amdahl's law using a different assumption, i.e. that the problem sizes increases with the computing units. In this case the problem size *per unit* is fixed. Under this assumption, the achievable speedup is almost linear, as expressed by Equation 1.2. In this case r'_s and r'_p are the serial and parallel fraction measured on the parallel system instead of the serial one. Equation 1.2 defines the concept of *scaled speedup* or *weak scalability*

$$SpeedUp(N) = r'_s + r'_p * N = N + (1 - N) * r'_s \tag{1.2}$$

Even though the two equations are mathematically equivalent (Shi, 1996), they make drastically different assumptions. In our case the size of the problem is large and ever growing. Hence it seems appropriate to adopt Gustafson's point of view, which justifies the parallel approach.

Parallel computing has a long history. It has traditionally focused on *"number crunching"*. Common applications were tightly coupled and CPU intensive (e.g. large simulations or finite element analysis). Control-parallel programming interfaces like Message Passing Interface (MPI) or Parallel Virtual Machine (PVM) are still the de-facto standard in this area. These systems are notoriously hard to program. Fault tolerance is difficult to achieve and scalability is an art. They require explicit control of parallelism and are called *"the assembly language of parallel computing"*.

In stark contrast with this legacy, a new class of parallel systems has emerged: *cloud computing*. Cloud systems focus on being scalable, fault tolerant, cost effective and easy to use.

Lately cloud computing has received a substantial amount of attention from industry, academia and press. As a result, the term "cloud computing" has become a buzzword, overloaded with meanings. There is lack of consensus on what is and what is not *cloud*. Even simple client-server applications are sometimes included in the category (Creeger, 2009). The boundaries between similar technologies are fuzzy, so there is no clear distinction among grid, utility, cloud, and other kinds of computing technologies. In spite of the many attempts to describe cloud computing (Mell and Grance, 2009), there is no widely accepted definition.

However, within cloud computing, there is a more cohesive subset of technologies which is geared towards data analysis. We refer to this subset as Data Intensive Scalable Computing (DISC) systems. These systems are aimed mainly at I/O intensive tasks, are optimized for dealing with large amounts of data and use a data-parallel approach. An interesting feature is they are *"dispersed"*: computing and storage facilities are distributed, abstracted and intermixed. These systems attempt to move computation as close to data as possible because moving large quantities of data is expensive. Finally, the burden of dealing with the issues caused by parallelism is removed from the programmer. This provides the programmer with a *scale-agnostic* programming model.

The data-parallel nature of DISC systems abstracts away many of the details of parallelism. This allows to design clean and elegant algorithms. DISC systems offer a limited interface that allows to make strong assumptions about user code. This abstraction is useful for performance optimization, but constrains the class of algorithms that can be run on these systems. In this sense, DISC systems are not general purpose computing systems, but are specialized in solving a specific class of problems.

DISC systems are a natural alternative to PDBMSs when dealing with large scale data. As such, a fierce debate is currently taking place, both in industry and academy, on which is the best tool (Dean and S. Ghemawat, 2010; DeWitt and Stonebraker, 2008; Stonebraker et al., 2010).

1.3.2 Data Intensive Scalable Computing

Let us highlight some of the requirements for a system used to perform data intensive computing on large datasets. Given the effort to find a novel solution and the fact that data sizes are ever growing, this solution should be applicable for a long period of time. Thus the most important requirement a solution has to satisfy is *scalability*.

Scalability is defined as "the ability of a system to accept increased input volume without impacting the profits". This means that the gains from the input increment should be proportional to the increment itself. This is a broad definition used also in other fields like economy. For a system to be fully scalable, the size of its input should not be a design parameter. Forcing the system designer to take into account all possible deployment sizes in order to cope with different input sizes leads to a scalable architecture without fundamental bottlenecks.

However, apart from scalability, there are other requirements for a large scale data intensive computing system. Real world systems cost money to build and operate. Companies attempt to find the most cost effective way of building a large system because it usually requires a significant money investment. Partial upgradability is an important money saving feature, and is more easily attained with a loosely coupled system. Operational costs like system administrators' salaries account for a large share of the budget of IT departments. To be profitable, large scale systems must require as little human intervention as possible. Therefore *autonomic* systems are preferable, systems that are self-configuring, self-tuning and self-healing. In this respect *fault tolerance* is a key property.

Fault tolerance is "the property of a system to operate properly in spite of the failure of some of its components". When dealing with a large number of systems, the probability that a disk breaks or a server crashes raises dramatically: it is the norm rather than the exception. A performance degradation is acceptable as long as the systems does not halt completely. A denial of service of a system has a negative economic impact, especially for Web-based companies. The goal of fault tolerance techniques is to create a *highly available system*.

To summarize, a large scale data analysis system should be scalable, cost effective and fault tolerant.

To make our discussion more concrete we give some examples of DISC systems. A more detailed overview can be found in Chapter 2 while here we just quickly introduce the systems we use in our research. While other similar systems exist, we have chosen these systems because of their availability as open source software and because of their widespread adoption both in academia and in industry. These factors increase the reusability of the results of our research and the chances of having practical impact on real-world problems.

The systems we make use of in this work implement two different paradigms for processing massive datasets: MapReduce (MR) and streaming. MapReduce offers the capability to analyze massive amounts of stored data while streaming solutions are designed to process a multitude of updates every second. We provide a detailed descriptions of these paradigms in Chapter 2.

Hadoop[1] is a distributed computing framework that implements the MapReduce paradigm(Dean and Ghemawat, 2004) together with a companion distributed file system called Hadoop Distributed File System (HDFS). Hadoop enables the distributed processing of huge datasets across clusters of commodity computers by means of a simple functional programming model.

A mention goes to Pig[2], a high level framework for data manipulation that runs on top of Hadoop (Olston et al., 2008). Pig is a very useful tool for data exploration, pre-processing and cleaning.

Finally, S4[3] is a distributed scalable stream processing engine (Neumeyer et al., 2010). While still a young project, its potential lies in complementing Hadoop for stream processing.

[1] http://hadoop.apache.org
[2] http://pig.apache.org
[3] http://incubator.apache.org/s4

1.4 Contributions

DISC systems are an emerging technology in the data analysis field that can be used to capitalize on massive datasets coming from the Web. *"There is no data like more data"* is a famous motto that epitomizes the opportunity to extract significant information by exploiting very large volumes of data. Information represents a competitive advantage for actors operating in the information society, an advantage that is all the greater the sooner it is achieved. Therefore, in the limit online analytics will become an invaluable support for decision making.

To date, DISC systems have been successfully employed for batch processing, while their use for online analytics has not received much attention and is still an open area of research. Many data analysis algorithms spanning different application areas have been already proposed for DISC systems. So far, speedup and scalability results are encouraging. We give an overview of these algorithms in Chapter 2.

However, it is not clear in the research community which problems are a good match for DISC systems. More importantly, the ingredients and recipes for building a successful algorithm are still hidden. Designing efficient algorithms for these systems requires thinking at scale, carefully taking into account the characteristics of input data, trading off communication and computing and addressing skew and load balancing problems. Meeting these requirements on a system with restricted primitives is a challenging task and an area for research.

This thesis explores the landscape of algorithms for Web mining on DISC systems and provides theoretical and practical insights on algorithm design and performance optimization.

Our work builds on previous research in Data Mining, Information Retrieval and Machine Learning. The methodologies developed in these fields are essential to make sense of data on the Web. We also leverage Distributed Systems and Database research. The systems and techniques studied in these fields are the key to get an acceptable performance on Web-scale datasets.

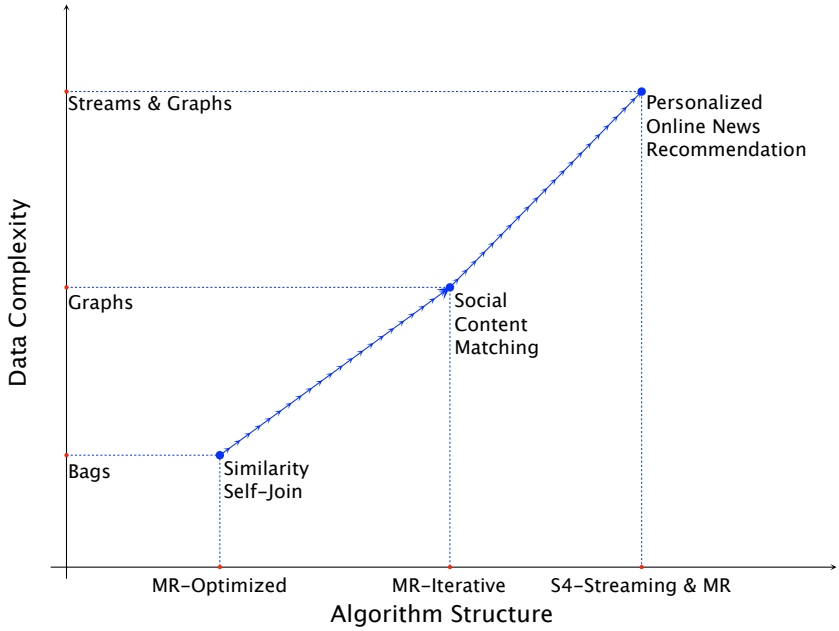

Figure 1.3: Complexity of contributed algorithms.

Concretely, our contributions can be mapped as shown in Figure 1.3. We tackle three different problems that involve Web mining tasks on different categories of data. For each problem, we provide algorithms for Data Intensive Scalable Computing systems.

First, we tackle the problem of similarity on bags of Web documents in Chapter 3. We present SSJ-2 and SSJ-2R, two algorithms specifically designed for the MapReduce programming paradigm. These algorithms are batch oriented and operate in a fixed number of steps.

Second, we explore graph matching in Chapter 4. We propose an application of matching to distribution of content from social media and Web 2.0. We describe STACKMR and GREEDYMR, two iterative MapReduce algorithms with different performance and quality properties. Both algorithms provide approximation guarantees and scale to huge datasets.

Third, we investigate news recommendation for social network users in Chapter 5. We propose a solution that takes advantage of the real-time Web to provide personalized and timely suggestions. We present T.REX, a methodology that combines stream and graph processing and is amenable to parallelization on stream processing engines like S4.

To summarize, the main contribution of this thesis lies in addressing classical problems like similarity, matching and recommendation in the context of Web mining and in providing efficient and scalable solutions that harness the power of DISC systems. While pursuing this general objective we use some more specific goals as concrete stepping stones.

In Chapter 3 we show that carefully designing algorithms specifically for MapReduce gives substantial performance advantages over trivial parallelization. By leveraging efficient communication patterns SSJ-2R outperforms state-of-the-art algorithms for similarity join in MapReduce by almost five times. Designing efficient MapReduce algorithms requires rethinking classical algorithms rather than using them as black boxes. By applying this principle we provide scalable algorithms for exact similarity computation without any need to tradeoff precision for performance.

In Chapter 4 we propose the first solution to the graph matching problem in MapReduce. STACKMR and GREEDYMR are two algorithms for graph matching with provable approximation guarantees and high practical value for large real-world systems. We further propose a general scalable computational pattern for iterative graph mining in MR. This pattern can support a variety of algorithms and we show how to apply it to the two aforementioned algorithms for graph matching.

Finally in Chapter 5 we describe a novel methodology that combines several signals from the real-time Web to predict user interest. T.REX is able to harnesses information extracted from user-generated content, social circles and topic popularity to provide personalized and timely news suggestions. The proposed system combines offline iterative computation on MapReduce and online processing of incoming data in a streaming fashion. This feature allows both to provide always fresh recommendations and to cope with the large amount of input data.

Chapter 2

Related Work

In this chapter we give an overview of related work in terms of systems, paradigms and algorithms for large scale Web mining.

We start by describing a general framework for DISC systems. Large scale data challenges have spurred the design of a multitude of new DISC systems. Here we review the most important ones and classify them according to our framework in a layered architecture. We further distinguish between batch and online systems to underline their different targets in the data and application spectrum. These tools compose the big data software stack used to tackle data intensive computing problems.

Then we offer a more detailed overview of the two most important paradigms for large scale Web mining: MapReduce and streaming. These two paradigms are able to cope with huge or even unbounded datasets. While MapReduce offers the capability to analyze massive amounts of stored data, streaming solutions offer the ability to process a multitude of updates per second with low latency.

Finally, we review some of the most influential algorithms for large scale Web mining on DISC systems. We focus mainly on MapReduce algorithms, which have received the largest share of attention. We show different kind of algorithms that have been proposed in the literature to process different types of data on the Web.

2.1 DISC systems

Even though existing DISC systems are very diverse, they share many common traits. For this reason we propose a general architecture of DISC systems, that captures the commonalities in the form of a multi-layered stack, as depicted in Figure 2.1. A complete DISC solution is usually devised by assembling multiple components. We classify the various components in three layers and two sublayers.

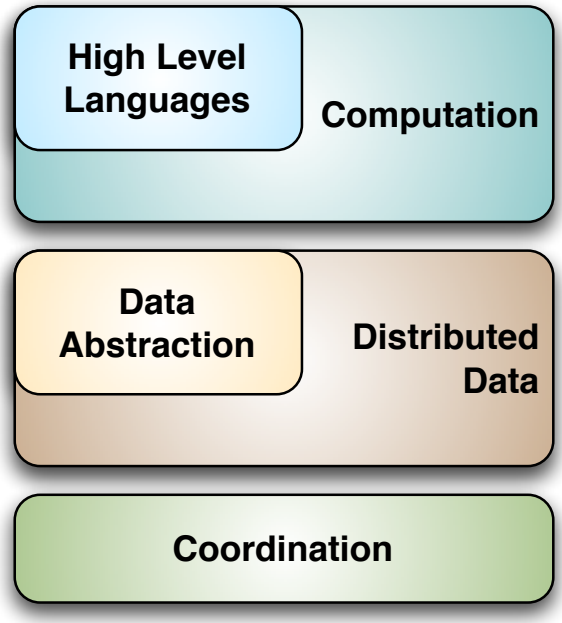

Figure 2.1: DISC architecture.

At the lowest level we find a *coordination* layer that serves as a basic building block for the distributed services higher in the stack. This layer deals with basic concurrency issues.

The *distributed data* layer builds on top of the coordination one. This layer deals with distributed data access, but unlike a traditional dis-

tributed file system it does not offer standard POSIX semantics for the sake of performance. The *data abstraction* layer is still part of the data layer and offers different, more sophisticated interfaces to data.

The *computation* layer is responsible for managing distributed processing. As with the data layer, generality is sacrificed for performance. Only embarrassingly data parallel problems are commonly solvable in this framework. The *high level languages* layer encompasses a number of languages, interfaces and systems that have been developed to simplify and enrich access to the computation layer.

Table 2.1: Major DISC systems.

	Batch	**Online**
High Level Languages	Sawzall, SCOPE, Pig Latin, Hive, DryadLINQ, FlumeJava, Cascading, Crunch	
Computation	MapReduce, Hadoop, Dryad, Pregel, Giraph, Hama	S4, Storm, Akka
Data Abstraction		BigTable, HBase, PNUTS, Cassandra, Voldemort
Distributed Data	GFS, HDFS, Cosmos	Dynamo
Coordination		Chubby, Zookeeper

Table 2.1 classifies some of the most popular DISC systems.

In the coordination layer we find two implementations of a consensus algorithm. Chubby (Burrows, 2006) is an implementation of Paxos (Lamport, 1998) while Zookeeper (Hunt et al., 2010) implements ZAB (Reed and Junqueira, 2008). They are distributed services for maintaining configuration information, naming, providing distributed synchronization and group services. The main characteristics of these services are very high availability and reliability, thus sacrificing high performance.

On the next level, the distributed data layer presents different kinds of data storages. A common feature in this layer is to avoid full POSIX semantic in favor of simpler ones. Furthermore, consistency is somewhat relaxed for the sake of performance.

HDFS [1], Google File System (GFS) (Ghemawat et al., 2003) and Cosmos (Chaiken et al., 2008) are distributed file systems geared towards large batch processing. They are not general purpose file systems. For example, in HDFS files can only be appended but not modified and in GFS a record might get appended more than once (*at least once* semantics). They use large blocks of 64 MB or more, which are replicated for fault tolerance. Dynamo (DeCandia et al., 2007) is a low latency key-values store used at Amazon. It has a Peer-to-Peer (P2P) architecture that uses consistent hashing for load balancing and a gossiping protocol to guarantee eventual consistency (Vogels, 2008).

The systems described above are either mainly append-only and batch oriented file systems or simple key-value stores. However, it is sometime convenient to access data in a different way, e.g. by using richer data models or by employing read/write operations. Data abstractions built on top of the aforementioned systems serve these purposes.

BigTable (Chang et al., 2006) and HBase[2] are non-relational data stores. They are actually multidimensional, sparse, sorted maps designed for semi-structured or non structured data. They provide random, realtime read/write access to large amounts of data. Access to data is provided via primary key only, but each key can have more than one column. PNUTS (Cooper et al., 2008) is a similar storage service developed by Yahoo! that leverages geographic distribution and caching, but offers limited consistency guarantees. Cassandra (Lakshman and Malik, 2010) is an open source Apache project initially developed by Facebook. It features a BigTable-like interface on a Dynamo-style infrastructure. Voldemort[3] is an open source non-relational database built by LinkedIn, basically a large persistent Distributed Hash Table (DHT).

[1] `http://hadoop.apache.org/hdfs`
[2] `http://hbase.apache.org`
[3] `http://project-voldemort.com`

In the computation layer we find paradigms for large scale data intensive computing. They are mainly dataflow paradigms with support for automated parallelization. We can recognize the same pattern found in previous layers also here: trade off generality for performance.

MapReduce (Dean and Ghemawat, 2004) is a distributed computing engine developed by Google, while Hadoop[4] is an open source clone. A more detailed description of this framework is presented in Section 2.2. Dryad (Isard et al., 2007) is Microsoft's alternative to MapReduce. Dryad is a distributed execution engine inspired by macro-dataflow techniques. Programs are specified by a Direct Acyclic Graph (DAG) whose vertexes are operations and whose edges are data channels. The system takes care of scheduling, distribution, communication and execution on a cluster. Pregel (Malewicz et al., 2010), Giraph[5] and Hama (Seo et al., 2010) are systems that implement the Bulk Synchronous Parallel (BSP) model (Valiant, 1990). Pregel is a large scale graph processing system developed by Google. Giraph implements Pregel's interface as a graph processing library that runs on top of Hadoop. Hama is a generic BSP framework for matrix processing.

S4 (Neumeyer et al., 2010) by Yahoo!, Storm[6] by Twitter and Akka[7] are distributed stream processing engines that implement the Actor model (Agha, 1986) They target a different part of the spectrum of big data, namely online processing of high-speed and high-volume event streams. Inspired by MapReduce, they provide a way to scale out stream processing on a cluster by using simple functional components.

At the last level we find high level interfaces to these computing systems. These interfaces are meant to simplify writing programs for DISC systems. Even though this task is easier than writing custom MPI code, DISC systems still offer fairly low level programming interfaces, which require the knowledge of a full programming language. The interfaces at this level allow even non programmers to perform large scale processing.

Sawzall (Pike et al., 2005), SCOPE (Chaiken et al., 2008) and Pig Latin

[4] http://hadoop.apache.org
[5] http://incubator.apache.org/giraph
[6] https://github.com/nathanmarz/storm
[7] http://akka.io

(Olston et al., 2008) are special purpose scripting languages for MapReduce, Dryad and Hadoop. They are able to perform filtering, aggregation, transformation and joining. They share many features with SQL but are easy to extend with UDFs. These tools are invaluable for data exploration and pre-processing. Hive (Thusoo et al., 2009) is a data warehousing system that runs on top of Hadoop and HDFS. It answers queries expressed in a SQL-like language called *HiveQL* on data organized in tabular format. Flumejava (Chambers et al., 2010), DryadLINQ (Yu et al., 2008), Cascading[8] and Crunch[9] are native language integration libraries for MapReduce, Dryad and Hadoop. They provide an interface to build pipelines of operators from traditional programming languages, run them on a DISC system and access results programmatically.

2.2 MapReduce

When dealing with large datasets like the ones coming from the Web, the costs of serial solutions are not acceptable. Furthermore, the size of the dataset and supporting structures (indexes, partial results, etc...) can easily outgrow the storage capabilities of a single node. The MapReduce paradigm (Dean and Ghemawat, 2004, 2008) is designed to deal with the huge amount of data that is readily available nowadays. MapReduce has gained increasing attention due to its adaptability to large clusters of computers and to the ease of developing highly parallel and fault-tolerant solutions. MR is expected to become the normal way to deal with massive datasets in the future (Rajaraman and Ullman, 2010).

MapReduce is a distributed computing paradigm inspired by concepts from functional languages. More specifically, it is based on two higher order functions: Map and Reduce. The Map function reads the input as a list of key-value pairs and applies a UDF to each pair. The result is a second list of intermediate key-value pairs. This list is sorted and grouped by key in the shuffle phase, and used as input to the Reduce function. The Reduce function applies a second UDF to each intermedi-

[8] http://www.cascading.org
[9] https://github.com/cloudera/crunch

ate key with all its associated values to produce the final result. The two phases are strictly non overlapping. The general signatures of the two phases of a MapReduce computation are as follows:

$$\begin{aligned} \textsf{Map}: &\quad \langle k_1, v_1 \rangle &\rightarrow &\quad [\langle k_2, v_2 \rangle] \\ \textsf{Reduce}: &\quad \langle k_2, [v_2] \rangle &\rightarrow &\quad [\langle k_3, v_3 \rangle] \end{aligned}$$

The Map and Reduce function are purely functional and thus without side effects. This property makes them easily parallelizable because each input key-value is independent from the other ones. Fault tolerance is also easily achieved by just re-executing the failed function instance.

MapReduce assumes a distributed file systems from which the Map instances retrieve their input data. The framework takes care of moving, grouping and sorting the intermediate data produced by the various mappers (tasks that execute the Map function) to the corresponding reducers (tasks that execute the Reduce function) .

The programming interface is easy to use and does not require any explicit control of parallelism. A MapReduce program is completely defined by the two UDFs run by mappers and reducers. Even though the paradigm is not general purpose, many interesting algorithms can be implemented on it. The most paradigmatic application is building an inverted index for a Web search engine. Simplistically, the algorithm reads the crawled and filtered web documents from the file system, and for every word it emits the pair $\langle word, doc\_id \rangle$ in the Map phase. The Reduce phase simply groups all the document identifiers associated with the same word $\langle word, [doc\_id_1, doc\_id_2, \ldots] \rangle$ to create an inverted list.

The MapReduce data flow is illustrated in Figure 2.2. The mappers read their data from the distributed file system. The file system is normally co-located with the computing system so that most reads are local. Each mapper reads a split of the input, applies the Map function to the key-value pair and potentially produces one or more output pairs. Mappers sort and write intermediate values on the local disk.

Each reducer in turn pulls the data from various remote locations. Intermediate key-value pairs are already partitioned and sorted by key by the mappers, so the reducer just merge-sorts the different partitions to

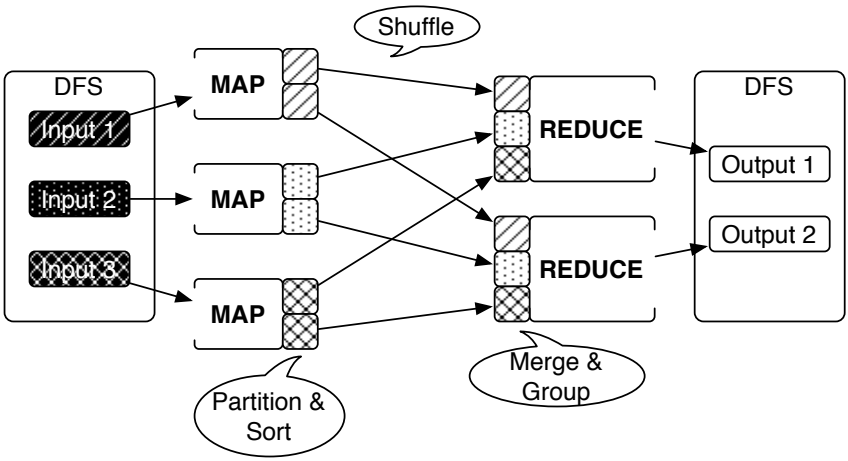

Figure 2.2: Data flow in the MapReduce programming paradigm.

group the same keys together. This phase is called *shuffle* and is the most expensive in terms of I/O operations. The shuffle phase can partially overlap with the Map phase. Indeed, intermediate results from mappers can start being transferred as soon as they are written to disk. In the last phase each reducer applies the Reduce function to the intermediate key-value pairs and write the final output to the file system.

MapReduce has become the de-fact standard for the development of large scale applications running on thousand of inexpensive machines, especially with the release of its open source implementation Hadoop.

Hadoop is an open source MapReduce implementation written in Java. Hadoop also provides a distributed file system called HDFS, used as a source and sink for MapReduce jobs. Data is split in chunks, distributed and replicated among the nodes and stored on local disks. MR and HDFS daemons run on the same nodes, so the framework knows which node contains the data. Great emphasis is placed on data locality. The scheduler tries to run mappers on the same nodes that hold the input data in order to reduce network traffic during the Map phase.

2.2.1 Computational Models and Extensions

A few computational models for MapReduce have been proposed. Afrati and Ullman (2009) propose an I/O cost model that captures the essential features of many DISC systems. The key assumptions of the model are:

- *Files* are replicated sets of records stored on a distributed file system with a very large block size b and can be read and written in parallel by processes;
- *Processes* are the conventional unit of computation but have limits on I/O: a lower limit of b (the block size) and an upper limit of s, a quantity that can represent the available main memory;
- *Processors* are the computing nodes, with a CPU, main memory and secondary storage, and are available in infinite supply.

The authors present various algorithms for multiway join and sorting, and analyze the communication and processing costs for these examples. Differently from standard MR, an algorithm in this model is a DAG of processes, in a way similar to Dryad. Additionally, the model assumes that keys are not delivered in sorted order to the Reduce. Because of these departures from the traditional MR paradigm, the model is not appropriate to compare real-world algorithms developed for Hadoop.

Karloff et al. (2010) propose a novel theoretical model of computation for MapReduce. The authors formally define the Map and Reduce functions and the steps of a MR algorithm. Then they proceed to define a new algorithmic class: $\mathcal{MRC}^i$. An algorithm in this class is composed by a finite sequence of Map and Reduce rounds with some limitations. Given an input of size n:

- each Map or Reduce is implemented by a random access machine that uses sub-linear space and polynomial time in n;
- the total size of the output of each Map is less than quadratic in n;
- the number of rounds is $O(\log^i n)$.

The model makes a number of assumptions on the underlying infrastructure to derive the definition. The number of available processors is assumed to be sub-linear. This restriction guarantees that algorithms in $\mathcal{MRC}$ are practical. Each processor has a sub-linear amount of memory. Given that the Reduce phase can not begin until all the Maps are done, the intermediate results must be stored temporarily in memory. This explains the space limit on the Map output which is given by the total memory available across all the machines. The authors give examples of algorithms for graph and string problems. The result of their analysis is an algorithmic design technique for $\mathcal{MRC}$.

A number of extensions to the base MR system have been developed. Many of these works focus on extending MR towards the database area.

Yang et al. (2007) propose an extension to MR in order to simplify the implementation of relational operators. More specifically they target the implementation of join, complex, multi-table select and set operations. The normal MR workflow is extended with a third final Merge phase. This function takes as input two different key-value pair lists and outputs a third key-value pair list. The model assumes that the output of the Reduce function is fed to the Merge. The signature are as follows.

$$
\begin{array}{rcl}
\textsf{Map}: & \langle k_1, v_1 \rangle_\alpha & \rightarrow \quad [\langle k_2, v_2 \rangle]_\alpha \\
\textsf{Reduce}: & \langle k_2, [v_2] \rangle_\alpha & \rightarrow \quad \langle k_2, [v_3] \rangle_\alpha \\
\textsf{Merge}: & \langle \langle k_2, [v_3] \rangle_\alpha , \langle k_3, [v_4] \rangle_\beta \rangle & \rightarrow \quad [\langle k_4, v_5 \rangle]_\gamma
\end{array}
$$

where α, β, γ represent *data lineages*. The lineage is used to distinguish the source of the data, a necessary feature for joins.

The signatures for the Reduce function in this extension is slightly different from the one in traditional MR. The Merge function requires its input to be organized in partitions. For this reason the Reduce function passes along the key k_2 received from the Map to the next phase without modifying it. The presence of k_2 guarantees that the two inputs of the Merge function can be matched by key.

The implementation of the Merge phase is quite complex so we refer the reader to the original paper for a detailed description. The proposed framework is efficient even though complex for the user. To implement a single join algorithm the programmer needs to write up to five different functions for the Merge phase only. Moreover, the system exposes many internal details that pollute the clean functional interface of MR.

The other main area of extension is adapting MR for online analytics. This modification would give substantial benefits in adapting to changes, and the ability to process stream data. Hadoop Online Prototype (HOP) is a proposal to address this issue (Condie et al., 2009). The authors modify Hadoop in order to pipeline data between operators, and to support online aggregation and continuous queries. In HOP a downstream dataflow element can begin consuming data before a producer element has completed its execution. Hence HOP can generate and refine an approximate answer by using online aggregation (Hellerstein et al., 1997). Pipelining also enables to push data as it comes inside a running job, which in turn enables stream processing and continuous queries.

To implement pipelining, mappers push data to reducers as soon as it is ready. The pushed data is treated as tentative to retain fault tolerance, and discarded in case of failure. Online aggregation is performed by applying the Reduce function to all the pipelined data received so far. The result is a snapshot of the computation and is saved on HDFS. Continuous MR jobs are implemented by using both pipelining and aggregation on data streams, by reducing a sliding window of mappers output.

HOP presents several shortcomings because of hybrid model. Pipelining is only possible between mappers and reducers in one job. Online aggregation recomputes the snapshot from scratch each time, consuming computing and storage resources. Stream processing can only be performed on a window because of fault tolerance. Anyway reducers do not process the data continuously but are invoked periodically. HOP tries to transform MapReduce from a batch system to an online system by reducing the batch size and addressing the ensuing inefficiencies.

HStreaming[10] also provides stream processing on top of Hadoop.

[10] `http://www.hstreaming.com`

2.3 Streaming

Streaming is an fundamental model for computations on massive datasets (Alon et al., 1999; Henzinger et al., 1998). In the traditional streaming model we are allowed only a constant number of passes on the data and poly-logarithmic space in the size of the input n. In the semi-streaming model, we are allowed a logarithmic number of passes and $O(n \cdot \text{polylog}\, n)$ space (Feigenbaum et al., 2005). Excluding the distribution over multiple machines, these models are indeed very similar to the model of computation allowed in MapReduce. Feldman et al. (2007) explore the relationship between streaming and MapReduce algorithms.

Streaming is applicable to a wide class of data-intensive applications in which the data is modeled best as transient data streams rather than persistent relations. Examples of such applications include financial applications, network monitoring, security and sensor networks.

A data stream is a continuous, ordered sequence of items. However, its continuous arrival in multiple, rapid, possibly unpredictable and unbounded streams raises new interesting challenges (Babcock et al., 2002).

Data streams differ from the traditional batch model in several ways:

- data elements in the stream arrive online;
- the algorithm has no control over the order of the items;
- streams are potentially unbounded in size;
- once an item has been processed it is discarded or archived.

This last fact implies that items cannot be retrieved easily unless they are explicitly stored in memory, which is usually small compared to the size of the input data streams.

Processing on streams in performed via *continuous queries*, which are evaluated continuously as streams arrive. The answer to a continuous query is produced over time, always reflecting the stream items seen so far. Continuous query answers may be stored and updated as new data arrives, or they may be produced as data streams themselves.

Since data streams are potentially unbounded, the amount of space required to compute an exact answer to a continuous query may also grow without bound. While batch DISC systems are designed to handle massive datasets, such systems are not well suited to data stream applications since they lack support for continuous queries and are typically too slow for real-time response.

The data stream model is most applicable to problems where short response times are important and there are large volumes of data that are being continually produced at a high speed. New data is constantly arriving even as the old data is being processed. The amount of computation time per data element must be low, or else the latency of the computation will be too high and the algorithm will not be able to keep pace with the data stream.

It is not always possible to produce exact answers for data stream queries by using a limited amount of memory. However, high-quality approximations are often acceptable in place of exact answers (Manku and Motwani, 2002). Sketches, random sampling and histograms are common tools employed for data summarization. Sliding windows are the standard way to perform joins and other relational operators.

2.3.1 S4

S4 (Simple Scalable Streaming System) is a distributed stream-processing engine inspired by MapReduce (Neumeyer et al., 2010). It is a general-purpose, scalable, open platform built from the ground up to process data streams as they arrive, one event at a time. It allows to easily develop applications for processing unbounded streams of data when a single machine cannot possibly cope with the volume of the streams.

Yahoo! developed S4 to solve problems in the context of search advertising with specific use cases in data mining and machine learning. The initial purpose of S4 was to process user feedback in real-time to optimize ranking algorithms. To accomplish this purpose, search engines build machine learning models that accurately predict how users will click on ads. To improve the accuracy of click prediction the models in-

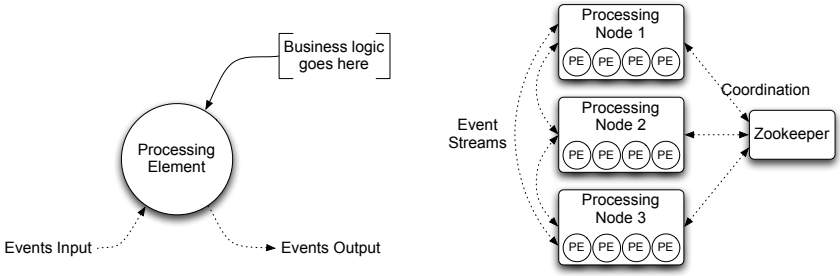

Figure 2.3: Overview of S4.

corporate recent search queries, clicks, and timing information as a form of personalization. A production system needs to process thousands of search queries per second from potentially millions of users per day.

The authors initially considered extending Hadoop, but:

> *"Rather than trying to fit a square peg into a round hole we decided to explore a programming paradigm that is simple and can operate on data streams in real-time."*

S4 implements the well known Actor model (Agha, 1986). In an Actor based system, actors are the element of computation and communication is handled exclusively by message passing. Actors share no state, therefore a system based on the Actor model lends itself to easy parallelization on a cluster of shared nothing machines. The Actor model provides encapsulation and location transparency, thus allowing applications to be massively concurrent while exposing a simple API to developers.

In the context of S4, actors are called Processing Elements (PEs) and they communicate via event streams as shown in Figure 2.3. Much like in MR, events consist of key-value pairs. Keyed data events are routed with affinity to PEs. PEs consume the events and may publish results or emit one or more events which may be consumed by other PEs.

The platform instantiates a PE for each value of the key attribute. When a new key is seen in an event, S4 creates a new instance of the PE associated to that key. Thus, for each PE there will be as many instances

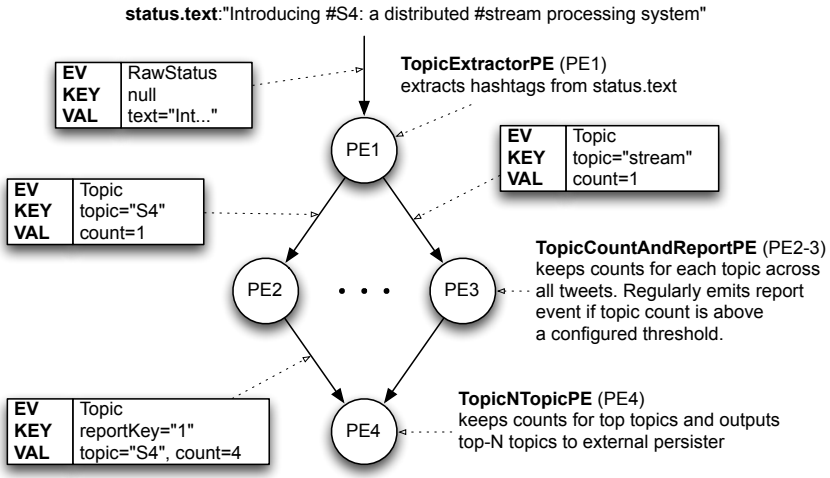

Figure 2.4: Twitter hashtag counting in S4.

as the number of different keys in the data stream. The instances are spread among the cluster to provide load balancing. Each PE consumes exactly those events whose key matches its own instantiation key.

A PE is implemented by defining a ProcessEvent function. This UDF is handed the current event, can perform any kind of computation and emit new keyed events on a stream. A communication layer will take care of routing all the events to the appropriate PE. The user can define an application by specifying a DAG of PEs connected by event streams.

Unlike other systems S4 has no centralized master node. Instead, it leverages ZooKeeper to achieve a clean and symmetric cluster architecture where each node provides the same functionalities. This choice greatly simplifies deployments and cluster configuration changes.

Figure 2.4 presents a simple example of an S4 application that keeps track of trending hashtag on Twitter. PE1 extracts the hashtags from the tweet and sends them down to PE2 and PE3 that increment their counters. At regular intervals the counters are sent to an aggregator that publishes the top-N most popular ones.

2.4 Algorithms

Data analysis researchers have seen in DISC systems the perfect tool to scale their computations to huge datasets. In the literature, there are many examples of successful applications of DISC systems in different areas of data analysis. In this section, we review some of the most significative example, with anl emphasis on Web mining and MapReduce.

Information retrieval has been the classical area of application for MR Indexing bags of documents is a representative application of this field, as the original purpose of MR was to build Google's index for Web search. Starting from the original algorithm, many improvements have been proposed, from single-pass indexing to tuning the scalability and efficiency of the process. (McCreadie et al., 2009a,b, 2011).

Several information retrieval systems have been built on top of MR. Terrier (Ounis et al., 2006), Ivory (Lin et al., 2009) and MIREX (Hiemstra and Hauff, 2010) enable testing new IR techniques in a distributed setting with massive datasets. These systems also provide state-of-the-art implementation of common indexing and retrieval functionalities.

As we explain in Chapter 3, pairwise document similarity is a common tool for a variety of problems such as clustering and cross-document coreference resolution. MR algorithms for closely related problems like fuzzy joins (Afrati et al., 2011a) and pairwise semantic similarity (Pantel et al., 2009) have been proposed. Traditional two-way and multiway joins have been explored by Afrati and Ullman (2010, 2011)

Logs are a gold mine of information about users' preferences, naturally modeled as bags of lines. For instance frequent itemset mining of query logs is commonly used for query recommendation. Li et al. (2008) propose a massively parallel algorithm for this task. Zhou et al. (2009) design a DISC-based OLAP system for query logs that supports a variety of data mining applications. Blanas et al. (2010) study several MR join algorithms for log processing, where the size of the bags is very skewed.

MapReduce has proven itself as an extremely flexible tool for a number of tasks in text mining such as expectation maximization (EM) and latent dirichlet allocation (LDA) (Lin and Dyer, 2010; Liu et al., 2011).

Mining massive graphs is another area of research where MapReduce has been extensively employed. Cohen (2009) presents basic patterns of computation for graph algorithms in MR, while Lin and Schatz (2010) presents more advanced performance optimizations. Kang et al. (2011) propose an algorithm to estimate the graph diameter for graphs with billions of nodes like the Web, which is by itself a challenging task.

Frequently computed metrics such as the clustering coefficient and the transitivity ratio involve counting triangles in the graph. Tsourakakis et al. (2009) and Pagh and Tsourakakis (2011) propose two approaches for approximated triangle counting on MR using randomization, while Yoon and Kim (2011) propose an improved sampling scheme. Suri and Vassilvitskii (2011) propose exact algorithms for triangle counting that scale to massive graphs on MapReduce.

MR has been used also to address NP-complete problems. Brocheler et al. (2010) and Afrati et al. (2011b) propose algorithms for subgraph matching. Chierichetti et al. (2010) tackles the problem of max covering, which can be applied to vertexes and edges in a graph. Lattanzi et al. (2011) propose a filtering technique for graph matching tailored for MapReduce. In Chapter 4 we will present two MapReduce algorithms for graph b-matching with applications to social media.

A number of different graph mining tasks can be unified via generalization of matrix operations. Starting from this observation, Kang et al. (2009) build a MR library for operation like PageRank, random walks with restart and finding connected components. Co-clustering is the simultaneous clustering of the rows and columns of a matrix in order to find correlated sub-matrixes. Papadimitriou and Sun (2008) describe a MR implementation called Dis-Co. Liu et al. (2010) explore the problem of non-negative matrix factorization of Web data in MR.

Streaming on DISC system is still an unexplored area of research. Large scale general purpose stream processing platforms are a recent development. However, many streaming algorithms in the literature can be adapted easily. For an overview of distributed stream mining algorithms see the book by Aggarwal (2007). For large scale stream mining algorithms see Chapter 4 of the book by Rajaraman and Ullman (2010).

DISC technologies have also been applied to other fields, such as scientific simulations of earthquakes (Chen and Schlosser, 2008), collaborative Web filtering (Noll and Meinel, 2008) and bioinformatics (Schatz, 2009). Machine learning has been a fertile ground for MR applications such as prediction of user rating of movies (Chen and Schlosser, 2008). In general, many widely used machine learning algorithms can be implemented in MR (Chu et al., 2006).

Chapter 3

Similarity Self-Join in MapReduce

Similarity self-join computes all pairs of objects in a collection such that their similarity value satisfies a given criterion. For example it can be used to find similar users in a community or similar images in a set. In this thesis we focus on bags of Web pages. The inherent sparseness of the lexicon of pages on the Web allows for effective pruning strategies that greatly reduce the number of pairs to evaluate.

In this chapter we introduce SSJ-2 and SSJ-2R, two novel parallel algorithms for the MapReduce framework. Our work builds on state of the art pruning techniques employed in serial algorithms for similarity join. SSJ-2 and SSJ-2R leverage the presence of a distributed file system to support communication patterns that do not fit naturally the MapReduce framework. SSJ-2R shows how to effectively utilize the sorting facilities offered by MapReduce in order to build a streaming algorithm. Finally, we introduce a partitioning strategy that allows to avoid memory bottlenecks by arbitrarily reducing the memory footprint of SSJ-2R.

We present a theoretical and empirical analysis of the performance of our two proposed algorithms across the three phases of MapReduce. Experimental evidence on real world data from the Web demonstrates that SSJ-2R outperforms the state of the art by a factor of 4.5.

3.1 Introduction

In this chapter, we address the *similarity self-join* problem: given a collection of objects, we aim at discovering every pair of objects with high similarity, according to some similarity function. The task of discovering similar objects within a given collection is common to many real world data mining and machine learning problems.

Item-based and user-based *recommendation algorithms* require to compute, respectively, pair-wise similarity among users or items. Due to the large number of users and objects commonly handled by popular recommender systems similarity scores are usually computed off-line.

Near duplicate detection is commonly performed as a pre-processing step before building a document index. It may be used to detect redundant documents, which can therefore be removed, or it may be a hint to discover content farms and spam websites exploiting content repurposing strategies. Near duplicate detection finds application also in the area of copyright protection as a tool for discovering plagiarism.

Density-based *clustering* algorithms like DBSCAN (Ester et al., 1996) or OPTICS (Ankerst et al., 1999) inherently join the input data based on similarity relationships. Correlation clustering (Bansal et al., 2004) uses similarity relationship between the objects instead of the actual representation of the objects. All these algorithms will draw high benefit from efficient and scalable MapReduce implementations of similarity joins.

Finding similar objects is also a basic step for *data cleaning* tasks. When dealing with real world data, we wish to normalize different but similar representations of the same entity, e.g. different names or addresses, caused by mistakes or formatting conventions. Similarity joins also play an important role in distance-based outlier detection.

We focus on document collections and model them as bags of vectors (see Section 1.2.1). Each document is a vector in a highly-dimensional space. Documents are a particular kind of data that exhibits a significant sparseness: only a small subset of the whole lexicon occurs in any given document. This sparsity allows to exploit pruning strategies that reduce the potentially quadratic number of candidate pairs to evaluate.

Finally, the size of the collection at hand poses new interesting challenges. This is particularly relevant for Web-related collections, where the number of documents involved is measured in billions. For this setting, distributed solutions become mandatory. Indeed, there are a few related proposals that exploit the MapReduce framework. Elsayed et al. (2008) propose a MapReduce algorithm to compute the similarity between every pair of documents in a collection, without taking advantage of any pruning strategy. Vernica et al. (2010) present a MapReduce algorithm to compute the similarity join of collections of *sets*.

Our main contributions are the following.

- We present SSJ-2 and SSJ-2R the efficient distributed algorithms based on the MapReduce framework to compute the similarity self-join on a collection of documents.
- We borrow from existing pruning techniques designed for serial algorithms, and we extend them to the distributed setting.
- We exploit selective communication patterns that do not naturally fit the MapReduce paradigm.
- We provide detailed analytical and empirical analysis of the algorithms across the various phases of the MapReduce framework.
- We show that SSJ-2R performs nicely on a sample of the WT10g TREC Web Corpus.[1]

The rest of the chapter is organized as follows. In Section 3.2 we introduce the similarity self-join problem and some solutions adopted by non-parallel algorithms. In Section 3.3 we discuss the algorithms for similarity self-join that are based on the MapReduce framework. Section 3.4 presents our proposed algorithms, which are theoretically analyzed in Section 3.5 and experimentally evaluated in Section 3.6.

[1] `http://ir.dcs.gla.ac.uk/test_collections/wt10g.html`

3.2 Problem definition and preliminaries

We address the similarity self-join problem applied to a collection $\mathcal{D}$ of documents. Let $\mathcal{L}$ be the lexicon of the collection. Each document d is represented as a $|\mathcal{L}|$-dimensional vector. Without loss of generality, we assume that documents are normalized, so $d[i]$ denotes the number of occurrences of the i-th term in the document d divided by the norm $\|d\|$. We adopt the cosine distance to measure the similarity between two documents. For normalized documents the cosine similarity can be simply computed as the inner product between document vectors.

Problem 1 (Similarity Self-Join Problem)**.** *Given a collection of normalized document-vectors $\mathcal{D} = \{d_1, \ldots, d_N\}$, a similarity function $\sigma \colon \mathcal{D} \times \mathcal{D} \to \mathbb{R}$ and a minimum similarity threshold $\overline{\sigma}$, the* Similarity Self-Join *problem requires to discover all those pairs $d_i, d_j \in \mathcal{D}$ with similarity above the threshold:*

$$\sigma(d_i, d_j) = \sum_{0 \leq t < |\mathcal{L}|} d_i[t] \cdot d_j[t] \geq \overline{\sigma}$$

Note that the results discussed in this chapter can be easily applied to other similarity measures, such as Jaccard, Dice or overlap.

Differently from normal join problems, self-join involves joining a large table with itself. Thus it is not possible to apply traditional algorithms for data warehousing, that join a large fact table with smaller dimension tables. Recent algorithms for log processing in MR also assume that one of the tables is much smaller that the other (Blanas et al., 2010).

Following Arasu et al. (2006); Xiao et al. (2009), we classify algorithms for similarity self-join by their solution to the following sub-problems:

1. **Signature scheme:** a compact representation of each document;
2. **Candidate generation:** identifying potentially similar document pairs given their signature;
3. **Verification:** computing document similarity;
4. **Indexing:** data structures for speeding up candidate generation and verification.

The naïve approach produces the document itself as the signature. It generates all the $O(N^2)$ possible document-signature pairs, and computes their actual similarity without any supporting indexing structure. This approach is overly expensive in the presence of sparse data such as bag of documents. It is very common to have documents thad to not share any term, leading to a similarity of zero. In this case, smart pruning strategies can be exploited to discard early such document pairs.

Broder et al. (1997) adopt a simple signature scheme, that we refer to as *Term-Filtering*. The signature of a document is given by the terms occurring in that document. These signatures are stored in an inverted index, which is then used to compare only pairs of documents sharing at least one item in their signature, i.e. one term.

Prefix-Filtering (Chaudhuri et al., 2006) is an extension of the *Term-Filtering* idea. Let $\hat{d}$ be an artificial document such that $\hat{d}[t] = \max_{d \in \mathcal{D}} d[t]$. Given a document d, let its *boundary* $b(d)$ be the largest integer such that $\sum_{0 \leq t < b(d)} d[t] \cdot \hat{d}[t] < \overline{\sigma}$. The signature of the document d is the set of terms occurring in the document vector beginning from position $b(d)$, $S(d) = \{b(d) \leq t < |\mathcal{L}| \mid d[t] \neq 0\}$. It is easy to show that if the signatures of two documents d_i, d_j have empty intersection, then the two documents have similarity below the threshold. Without loss of generality let us assume that $b(d_i) \prec b(d_j)$.

$$(S(d_i) \cap S(d_j) = \emptyset) \quad \Rightarrow \quad \sum_{b(d_i) \leq t < |\mathcal{L}|} d_i[t] \cdot d_j[t] = 0 \quad \Rightarrow$$

$$\sigma(d_i, d_j) = \sum_{0 \leq t < b(d_i)} d_i[t] \cdot d_j[t] \leq \sum_{0 \leq t < b(d_i)} d_i[t] \cdot \hat{d}[t] \quad < \overline{\sigma}$$

Therefore, only document signatures are stored in the inverted index and later used to generate candidate document pairs. Eventually, the full documents need to be retrieved in order to compute the actual similarity. This technique was initially proposed for set-similarity joins and later extended to documents with real-valued vectors.

In particular, Bayardo et al. (2007) adopt an online indexing and matching approach. Index generation, candidate pair generation and similarity score computation are all performed simultaneously and incrementally.

Each document is matched against the current index to find potential candidates. The similarity scores are computed by retrieving the candidate documents directly from the collection. Finally, a signature is extracted from the current document and added to the index.

This approach results in a single full scan of the input data, a small number of random accesses to single documents, and a smaller (on average) index to be queried. *Prefix-filtering* outperforms alternative techniques such as LSH (Gionis et al., 1999), PartEnum (Arasu et al., 2006) and ProbeCount-Sort (Sarawagi and Kirpal, 2004), which we therefore do not considered in our work. Unfortunately, an incremental approach that leverages a global shared index is not practical on a large parallel system because of contention on the shared resource.

Finally, Xiao et al. (2008) present some additional pruning techniques based on positional and suffix information. However these techniques are specifically tailored for set-similarity joins and cannot be directly applied to document similarity.

3.3 Related work

In this section we describe two MR algorithm for the similarity self-join problem. Each exploits one of the pruning techniques discussed above.

3.3.1 MapReduce Term-Filtering (ELSA)

Elsayed et al. (2008) present a MapReduce implementation of the *Term-Filtering* method. Hereafter we refer to it as ELSA for convenience. The authors propose an algorithm for computing the similarity of every document pair, which can be easily adapted to our setting by adding a simple post-filtering. The algorithm runs two consecutive MR jobs, the first builds an inverted index and the second computes the similarities.

Indexing: Given a document d_i, for each term, the mapper emits the term as the key, and a tuple $\langle i, d_i[t] \rangle$ consisting of document ID and weight as the value. The shuffle phase of MR groups these tuples by term and delivers these inverted lists to the reducers, that write them to disk.

$$\begin{aligned}
\textsf{Map}: &\quad \langle i, d_i \rangle &\rightarrow\ & [\langle t, \langle i, d_i[t] \rangle \rangle \mid d_i[t] > 0] \\
\textsf{Reduce}: &\quad \langle t, [\langle i, d_i[t] \rangle, \langle j, d_j[t] \rangle, \ldots] \rangle &\rightarrow\ & [\langle t, [\langle i, d_i[t] \rangle, \langle j, d_j[t] \rangle, \ldots] \rangle]
\end{aligned}$$

Similarity: Given the inverted list of term t, the mapper produces the contribution $w_{ij}[t] = d_i[t] \cdot d_j[t]$ for every pair of documents where the term t co-occurs. This value is associated with a key consisting of the pair of document IDs $\langle i, j \rangle$. For any document pair the shuffle phase will pass to the reducer the contribution list $\mathcal{W}_{ij} = \{w_{ij}[t] \mid w_{ij}[t] > 0,\ \forall t \in \mathcal{L}\}$ from the various terms, which simply need to be summed up.

$$\begin{aligned}
\textsf{Map}: &\quad \langle t, [\langle i, d_i[t] \rangle, \langle j, d_j[t] \rangle, \ldots] \rangle &\rightarrow\ & [\langle \langle i, j \rangle, w_{ij}[t] \rangle] \\
\textsf{Reduce}: &\quad \langle \langle i, j \rangle, \mathcal{W}_{ij} \rangle &\rightarrow\ & \left[\langle \langle i, j \rangle, \sigma(d_i, d_j) = \sum_{w \in \mathcal{W}_{ij}} w \rangle \right]
\end{aligned}$$

The *Term-Filtering* is exploited implicitly: the similarity of two documents that do not share any term is never evaluated because none of their terms ever occurs in the same inverted list.

The main drawback of this approach is caused by the large number of candidate document pairs that it generates. In fact, there are many document pairs that share only a few terms but have anyway a similarity largely below the threshold. The number of candidates directly affects completion time. More importantly it determines the shuffle size, the volume of intermediate data to be moved from mappers to reducers. Minimizing shuffle size is critical because network bandwidth is the only resource that cannot be easily added to a cluster.

Finally, recall that reducers may start only after every mapper has completed. Therefore, the presence of a few long inverted lists for common terms induces a significant load imbalance. Most of the resources will remain idle until the processing of the longest inverted list is completed. To alleviate this issue Elsayed et al. remove the top 1% most frequent terms, thus trading efficiency for precision. Our goal is to provide an efficient algorithm that computes the exact similarity.

Figure 3.1 illustrates an example of how ELSA works. For ease of explanation, the document vectors are not normalized. *Term-Filtering* avoids computing similarity scores of documents that do not share any term. For the special case in which an inverted list contains only one document, the similarity Map function does not produce any output. The two main problems of ELSA result evident from looking at this image. First, long inverted lists may produce a load imbalance and slow down the algorithm considerably, as for term "B" in the figure. Second, the algorithm computes low similarity scores which are not useful for the typical applications, as for document pair $\langle d_1, d_2 \rangle$ in the figure.

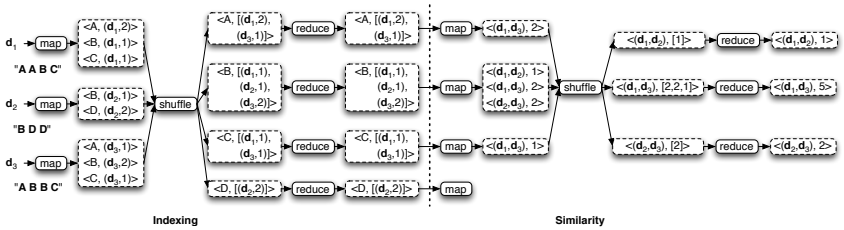

Figure 3.1: ELSA example.

3.3.2 MapReduce Prefix-Filtering (VERN)

Vernica et al. (2010) present a MapReduce algorithm based on *Prefix-Filtering* that uses only one MR step. Indeed, the authors discuss several algorithm for set-similarity join operations. Here we describe the best performing variant for set-similarity self-join. Hereafter we refer to it as VERN for convenience. For each term in the signature of a document $t \in S(d_i)$ as defined by *Prefix-Filtering*, the map function outputs a tuple with key the term t itself and value the whole document d_i. The shuffle phase delivers to each reducer a small sub-collection of documents that share at least one term in their signatures. This process can be thought as the creation of an inverted index of the signatures, where each posting is the document itself rather than a simple ID. Finally, each reducer

finds similar pairs among candidates by using state-of-the-art serial algorithms (Xiao et al., 2008). The Map and Reduce functions are as follows.

$$\begin{aligned} \textsf{Map}: &\ \langle i, d_i \rangle &&\rightarrow\ [\langle t, d_i \rangle \mid t \in S(d_i)] \\ \textsf{Reduce}: &\ \langle t, \mathcal{D}_v = [d_i, d_j, \ldots] \rangle &&\rightarrow\ [\langle \langle i, j \rangle, \sigma(d_i, d_j) \rangle \mid d_i, d_j \in \mathcal{D}_v] \end{aligned}$$

Nonetheless, this approach does not bring a significant improvement over the previous strategy. First, a document with a signature of length n is replicated n times, even if there are no other documents with an overlapping signature. Second, pairs of similar documents that have m common terms in their signatures are produced m times at different reducers. Computing these duplicates is a overhead of parallelization, and their presence requires a second scan of the output to get rid of them.

Figure 3.2 illustrates an example run of VERN. Light gray terms are pruned from the document by using *Prefix-Filtering*. Each document is replicated once for each non-pruned term it contains. Finally, the reducer computes the similarity of the bag of documents it receives by employing a serial SSJ algorithm. As shown in Figure 3.2, VERN computes the similarity of the pair $\langle d_1, d_3 \rangle$ multiple times at different reducers.

We propose here a simple strategy that solves the issue of duplicate similarity computation. We use this modified version of the original VERN algorithm in our evaluation. This modified version runs slightly but consistently faster than the original one, and it does not require post-processing. The duplicate similarities come from pair of documents that share more than one term in their signatures. Let $S_{ij} = S(d_i) \cap S(d_j)$ be the set of such shared terms in document pair $\langle d_i, d_j \rangle$. Let $\hat{t}$ be the last of such terms in the order imposed to the lexicon by *Prefix-Filtering* $\hat{t} = t_x \mid t_x \succeq t_y \ \forall\ t_x, t_y \in S_{ij}$. The reducer that receives each pair can simply check if its key corresponds to $\hat{t}$ and compute the similarity only in this case. The term $\hat{t}$ for each pair of documents can be efficiently computed by a simple backwards scan of the two arrays that represent the documents. This strategy allows to avoid computing duplicate similarities and does not impose any additional overhead the original algorithm.

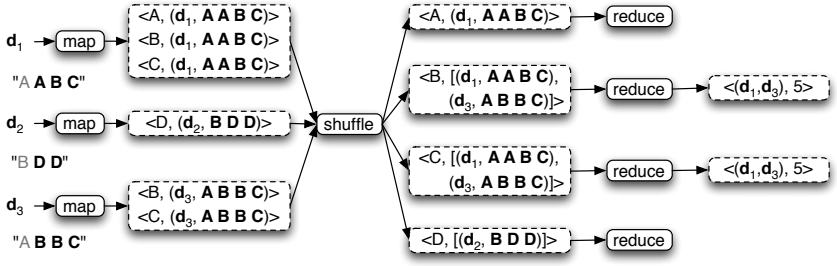

Figure 3.2: VERN example.

3.4 SSJ Algorithms

In this section we describe two algorithms based on *Prefix-Filtering* that overcome the weak points of the algorithms presented above. The first algorithm, named SSJ-2, performs a *Prefix-Filtering* in two MR steps. Note that in the first step we use a naïve indexing algorithm but if needed more advanced techniques can be used (McCreadie et al., 2009b, 2011). The second algorithm, named SSJ-2R, additionally uses a *remainder file* to broadcast data that is likely to be used by every reducer. It also effectively partitions the search space to reduce the memory footprint. Code for both algorithm is open source and available online.[2]

3.4.1 Double-Pass MapReduce Prefix-Filtering (SSJ-2)

This algorithm is and extention of ELSA, and also consists of an indexing phase followed by a similarity computation phase.

SSJ-2 shortens the inverted lists by employing *Prefix-Filtering*. As shown in Figure 3.3, the effect of *Prefix-Filtering* is to reduce the portion of document indexed. The terms occurring in d_i up to position $b(d_i)$, or b_i for short, need not be indexed. By sorting terms in decreasing order of frequency, the most frequent terms are discarded. This pruning shortens the longest inverted lists and brings a significant performance gain.

[2] https://github.com/azaroth/Similarity-Self-Join

To improve load balancing we employ a simple bucketing technique During the indexing phase we randomly hash the inverted lists to different buckets. This spreads the longest lists uniformly among the buckets. Each bucket is then consumed by a different mapper in the next phase.

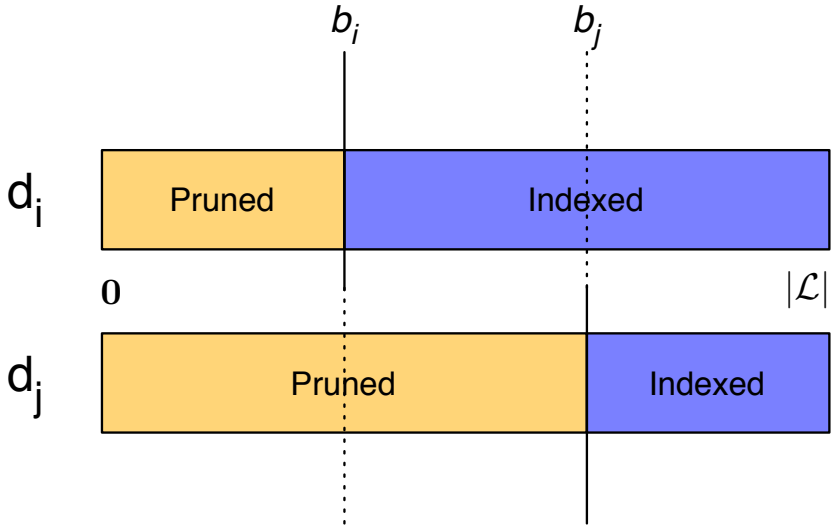

Figure 3.3: Pruned document pair: the left part (orange/light) has been pruned, the right part (blue/dark) has been indexed.

Unfortunately with this schema the reducers do not have enough information to compute the similarity between d_i and d_j. They receive only the contribution from terms $t \succeq b_j$, assuming $b_i \prec b_j$. For this reason, the reducers need two additional remote I/O operations to retrieve the two documents d_i and d_j from the underlying distributed file system.

SSJ-2 improves over both ELSA and VERN in terms of the amount of intermediate data produced. Thanks to *Prefix-Filtering*, the inverted lists are shorter than in ELSA and therefore the number of candidate pairs generated decreases quadratically. In VERN each document is *always* sent to a number of reducers equal to the size of its signature, even if no other document signature shares any term with it. In SSJ-2 a pair

of documents is sent to the reducer only if they share at least one term in their signature. Furthermore, while SSJ-2 computes the similarity between any two given documents just once, VERN computes the similarity of two documents in as many reducers as the size of the intersection of their signatures. Therefore, SSJ-2 has no computational overhead due to parallelization, and significantly reduces communication costs.

Figure 3.4 presents an example for SSJ-2. As before, light gray terms have been pruned and documents are not normalized to ease the explanation. Pairs of documents that are below the similarity threshold are discarded early. However the reducer of the similarity phase needs to access the remote file system to retrieve the documents in order to compute the correct final similarity score.

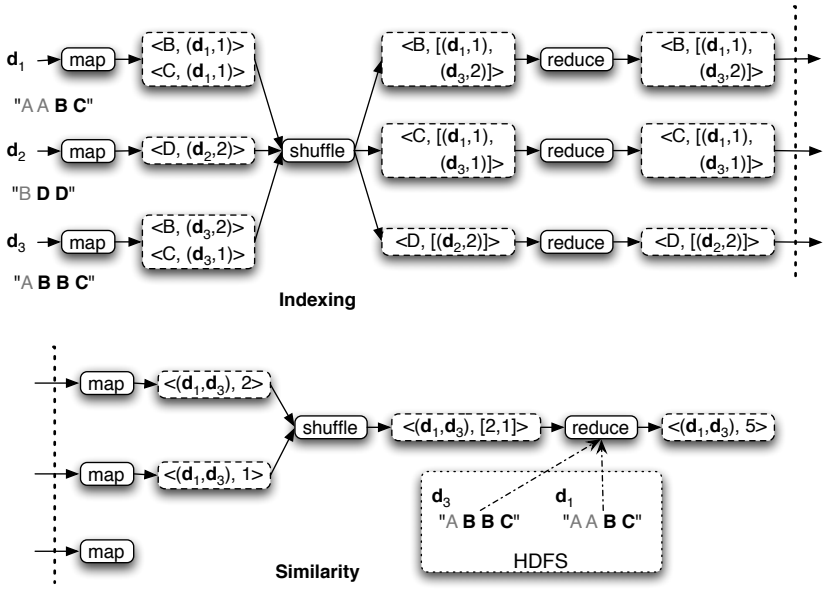

Figure 3.4: SSJ-2 example.

3.4.2 Double-Pass MapReduce Prefix-Filtering with Remainder File (SSJ-2R)

For any given pair of documents $\langle d_i, d_j \rangle$, the reducers of SSJ-2 receive only partial information. Therefore, they need to retrieve the full documents in order to correctly compute their similarity. Since a node runs multiple reducers over time, each of them remotely accessing two documents, in the worst case this is equivalent to broadcasting the full collection to every node, with obvious limitations to the scalability of the algorithm. We thus propose an improved algorithm named SSJ-2R that does not perform any remote random access, and that leverages the non-indexed portion of the documents.

Let us make a few interesting observations. First, some of the terms are very common, and therefore used to compute the similarity of most document pairs. For those terms, it would be more efficient to broadcast their contributions rather than pushing them through the MR framework. Indeed, such piece of information is the one pruned via Prefix-Filtering during the indexing phase. We thus propose to store the pruned portion of each document in a *remainder file* $\mathcal{D}_R$, which can be later retrieved by each reducer from the underlying distributed file system.

Second, the *remainder file* $\mathcal{D}_R$ does not contain all the information needed to compute the final similarity. Consider Figure 3.3, each reducer receives the contributions $w_{ij}[t] = d_i[t] \cdot d_j[t] \mid t \succeq b_j$. $\mathcal{D}_R$ contains information about the terms $d_i[t] \mid t \prec b_i\}$ and $\{d_j[t] \mid t \prec b_j$. But, subtly, no information is available for those terms $d_i[t] \mid b_i \preceq t \prec b_j$. On the one hand, those term weights are not in the remainder file because they have been indexed. On the other hand, the corresponding inverted lists contain the frequency of the terms in d_i but not of those occurring in d_j, and therefore, the weights $w_{ij}[t] = d_i[t] \cdot d_j[t]$ cannot be produced.

We thus propose to let one of the two documents be delivered to the reducer through the MR framework by shuffling it together with the weights of the document pairs. Given two documents d_i and d_j as shown in Figure 3.3, if $b_i \prec b_j$ we call d_i the *Least Pruned Document* and d_j the *Most Pruned Document*. Our goal is to group at the reducer each docu-

ment d with the contributions w of every document pair $\langle d_i, d_j \rangle$ for which d is the least pruned document between d_i and d_j. This document contains the pieces of information that we are missing in order to compute the final similarity score. This can be achieved by properly defining the keys produced by the mappers and their sorting and grouping operators.

More formally, we define two functions $\mathrm{LPD}(d_i, d_j)$ and $\mathrm{MPD}(d_i, d_j)$.

$$\mathrm{LPD}(d_i, d_j) = \begin{cases} i, & \text{if } b_i \prec b_j \\ j, & \text{otherwise} \end{cases} \qquad \mathrm{MPD}(d_i, d_j) = \begin{cases} j, & \text{if } b_i \prec b_j \\ i, & \text{otherwise} \end{cases}$$

First, we slightly modify the similarity Map function such that for every document pair of the given inverted list, it produces as keys a couple of document IDs where the first is always the LPD, and as values the MPD and the usual weight w.

$$\begin{aligned} \textsf{Map}\colon\ & [\langle t, [\langle i, d_i[t] \rangle, \langle j, d_j[t] \rangle, \ldots] \rangle] \rightarrow \\ & [\langle \langle \mathrm{LPD}(d_i, d_j), \mathrm{MPD}(d_i, d_j) \rangle, \langle \mathrm{MPD}(d_i, d_j), w_{ij}[t] \rangle \rangle] \end{aligned}$$

Second, we take advantage of the possibility of running independent Map functions whose outputs are then shuffled together. We define a Map function that takes the input collection and outputs its documents.

$$\textsf{Map}\colon \langle i, d_i \rangle \ \rightarrow\ [\langle \langle i, \phi \rangle, d_i \rangle]$$

where ϕ is a special value such that $\phi < i \quad \forall d_i \in \mathcal{D}$.

Third, we make use of the possibility offered by Hadoop to redefine parts of its communication pattern. The process we describe here is commonly known as *secondary sort* in MR parlance. We redefine the key partitioning function, which selects which reducer will process the key. Our function takes into account only the first document ID in the key. This partitioning scheme guarantees that all the key-value pairs that share the same LPD will end up in the same partition, together with one copy of the LPD document itself coming from the second Map function.

We instruct the shuffle phase of MR to sort the keys in each partition in ascending order. This order takes into account both IDs, the first as primary key and the second as secondary key. Consequently, for each

document d_i MR builds a list of key-value pairs such that the first key is $\langle i, \phi \rangle$, followed by every pair of document IDs for which d_i is the LPD.

Finally, we override the grouping function, which determines whether two keys belong to the same equivalence class. If two keys are equivalent, their associated values will be processed by the same single call to the Reduce function. Concretely, the grouping function builds the argument list for a call to the Reduce function by selecting values from the partition such that the corresponding keys are equivalent. Therefore our grouping function must consistent with our partitioning scheme: two keys $\langle i, j \rangle$ and $\langle i', j' \rangle$ are equivalent *iff* $i = i'$.

The input of the Reduce function is thus as follows.

$$\langle \langle i, \phi \rangle, [d_i, \langle j, w_{ij}[t'] \rangle, \langle j, w_{ij}[t''] \rangle, \ldots, \langle k, w_{ik}[t'] \rangle, \langle k, w_{ik}[t''] \rangle, \ldots] \rangle$$

The key is just the first key in the group, as they are all equivalent to MR. The first value is the LPD document d_i. Thanks to the sorting of the keys, it is followed by a set of contiguous stripes of weights. Each stripe is associated to the same document $d_j \mid j = \mathrm{MPD}(d_i, d_j)$.

SSJ-2R can now compute the similarity $\sigma(d_i, d_j)$ in one pass. Before the algorithm starts, each reducer loads the remainder file $\mathcal{D}_\mathcal{R}$ in memory. For each Reduce call, SSJ-2R starts by caching d_i, which is at the beginning of the list of values. Then, for each d_j in the list, it sums up all the corresponding weights w_{ij} to compute a partial similarity. Finally, it retrieves the pruned portion of d_j from $\mathcal{D}_\mathcal{R}$, computes the similarity of d_i to the pruned portion of d_j and adds this to the partial similarity computed before. This two step process correctly computes the final similarity $\sigma(d_i, d_j)$ because all the terms $b_i \preceq t \prec b_j$ are available both in d_i and in the pruned portion of d_j. This process is repeated for each stripe of weights belonging to the same document pair in the list of values.

Differently from SSJ-2, SSJ-2R performs no remote access to the document collection. Given a document pair, one is delivered fully through the MapReduce framework, and the other is partially retrieved from the remainder file $\mathcal{D}_R$. The advantage in terms of communication costs is given by the remainder file. Indeed, $\mathcal{D}_R$ is much smaller than the input collection. We show experimentally that its size is one order of magni-

tude smaller than the input. Therefore, $\mathcal{D}_R$ can be efficiently broadcast to every node through the distributed file system. Each reducer can load it in main memory so that the similarity computation can be fully accomplished without any additional disk access.

Figure 3.5 illustrates the full data flow of SSJ-2R with an example. As before, light gray terms have been pruned and documents are not normalized. SSJ-2R stores the pruned part of each document in the remainder file, which is put in the distributed cache in order to be broadcast. The reducer of the similarity phase reads the remainder file in memory before starting to process its value list. In the example, d_1 is delivered through the MR framework, while the pruned part of d_3 is recovered from the remainder file. This procedure allows to correctly compute the contributions of the term "A" which was pruned from the documents. The contributions from terms "B" and "C" are computed by the mapper of the similarity phase and delivered normally through MR.

3.4.3 Partitioning

Even though the remainder file $\mathcal{D}_R$ is much smaller than the original collection, its size will grow when increasing the collection size, and therefore it may hinder the scalability of the algorithm.

To overcome this issue, we introduce a partitioning scheme for the remainder file. Given a user defined parameter K, we split the range of document identifiers into K equally sized chunks, so that the unindexed portion of document d_i falls into the $\lfloor i/K \rfloor$-th chunk. Consequently, we modify the partitioning function which maps a key emitted by the mapper to a reducer. We map each key $\langle i, j \rangle$ to the $\lfloor j/K \rfloor$-th reducer instance, i.e. the mapping is done on the basis of the MPD. This means that each reducer will receive only weights associated to a portion of the document space. Therefore, the reducer needs to retrieve and load in memory only $1/K$-th of the remainder file $\mathcal{D}_R$.

This new partitioning scheme spreads the weights associated to the same LPD document over K different reducers. Therefore, to correctly compute the final similarity SSJ-2R needs a copy of the LPD document

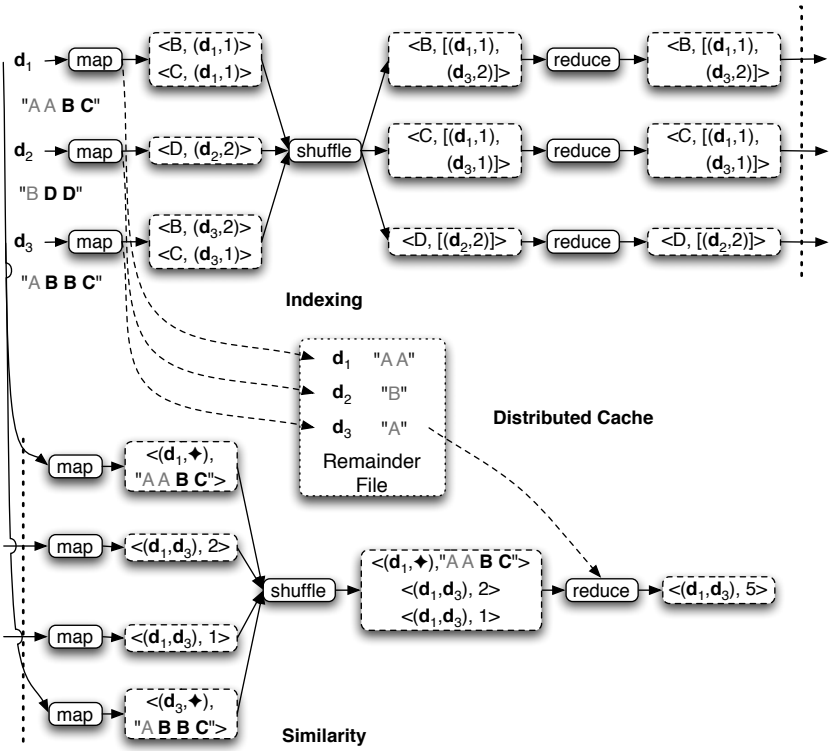

Figure 3.5: SSJ-2R example.

d_i at each of these reducers. For this reason, we replicate K times the special key $\langle i, \phi \rangle$ and its associated value d_i, once for each partition.

The parameter K allows to tune the memory usage of SSJ-2R and controls the tradeoff with the communication cost due to replication.

3.5 Complexity analysis

In Section 2.2.1 we presented a few proposals for modeling MapReduce algorithms (Afrati and Ullman, 2009; Karloff et al., 2010). Indeed, estimating the cost of a MapReduce algorithm is quite difficult, because of

Table 3.1: Symbols and quantities.

Symbol	Description
$\overline{d}$ $\widehat{d}$	Avg. and Max. document length
$\overline{s}$ $\widehat{s}$	Avg. and Max. signature length
$\overline{p}$ $\widehat{p}$	Avg. and Max. inverted list length with Prefix-Filtering
$\overline{l}$	Avg. inverted list length without pruning
r	Cost of a remote access to a document
R	Cost of retrieving the remainder file $\mathcal{D}_R$

Table 3.2: Complexity analysis.

ALGORITHM	Map	Shuffle	Reduce
ELSA	$\widehat{l}^2$	$\lvert\mathcal{L}\rvert \cdot \overline{l}^2$	$\widehat{d}$
VERN	$\widehat{s} \cdot \widehat{d}$	$\lvert\mathcal{D}\rvert \cdot \overline{s} \cdot \overline{d}$	$\widehat{p}^2 \cdot \overline{d}$
SSJ-2	$\widehat{p}^2$	$\lvert\mathcal{L}\rvert \cdot \overline{p}^2$	$\widehat{d} + r$
SSJ-2R	$\widehat{p}^2$	$\lvert\mathcal{L}\rvert \cdot \overline{p}^2 + \lvert D\rvert \cdot \overline{d} + R$	$\widehat{d}$

the inherent parallelism, the hidden cost of the shuffling phase, the overlap among computation and communication managed implicitly by the framework, the non-determinism introduced by combiners and so on.

We prefer to model separately the three main steps of a MapReduce jobs: the Map and Reduce functions, and the volume of the data to be shuffled. In particular, we take into consideration the cost associated to the function instance with the largest input. This way, we roughly estimate the maximum possible degree of parallelism and we are able to compare the different algorithms in deeper detail.

We refer to Table 3.1 for the symbols used in this section. The quantity $\overline{l}$ can also be seen as the average frequency of the terms in the lexicon $\mathcal{L}$. Similarly, $\overline{p}$ can also be seen as the number of signatures containing any given term. Clearly, it holds that $\overline{p} \leq \overline{l}$.

The running time of ELSA is dominated by the similarity phase. Its

Map function generates every possible document pair for each inverted list, with a cost of $O(\widehat{l}^2)$. Most of this large number of candidate pairs has a similarity below the threshold. Furthermore, the presence of long inverted lists introduces *stragglers*. Since reducers may start only after every mapper has completed, mappers processing the longest lists keep resources waiting idly. Since the shuffling step handles a single datum for every generated pair, we can estimate the cost by summing over the various inverted lists as $O(|\mathcal{L}| \cdot \bar{l}^2)$. The Reduce function just sums up the contributions of the various terms, with a cost $O(\widehat{d})$.

VERN runs only one MR job. The Map replicates each document d $|S(d)|$ times, with a cost of $O(\widehat{s} \cdot \widehat{d})$. Similarly, we can estimate the shuffle size as $O(|\mathcal{D}| \cdot \overline{d} \cdot \overline{s})$. The Reduce evaluates the similarity of every pair of documents in its input, with cost $O(\widehat{p}^2 \cdot \overline{d})$. Due to Prefix-Filtering, the maximum reducer input size is $\widehat{p}$, which is smaller than $\widehat{l}$.

By comparing the two algorithms, we notice the superiority of VERN in the first two phases of the computation. The Map function generates a number of replicas of each document, rather then producing a quadratic number of partial contributions. Considering that $|\mathcal{L}| \cdot \bar{l}$ is equal to $|\mathcal{D}| \cdot \overline{d}$, and that $\bar{l}$ is likely to be much larger than $\overline{s}$, it is clear that ELSA has the largest shuffle size. Conversely, the Reduce cost for VERN is quadratic in the size of the pruned lists $\widehat{p}$.

Our proposed algorithms SSJ-2 and SSJ-2R have a structure similar to ELSA. The Map cost $O(\widehat{p}^2)$ depends on the pruned inverted lists. For SSJ-2R, we are disregarding the cost of creating replicas of the input in the similarity Map function, since it is much cheaper than processing the longest inverted list. They have similar shuffles sizes, respectively $O(|\mathcal{L}| \cdot \overline{p}^2)$ and $O(|\mathcal{L}| \cdot \overline{p}^2 + |\mathcal{D}| \cdot \overline{d} + R)$. In this case we need to considered the cost of shuffling the input documents and the remainder file generated by SSJ-2R. Note that we are considering the remainder file as a shuffle cost even though it is not handled by MapReduce. Finally, the Reduce cost is different. In addition to the contributions coming from the pruned inverted index, SSJ-2 needs to access documents remotely with a total cost of $O(\widehat{d} + r)$. Conversely SSJ-2R has a cost of $O(\widehat{d})$ since the unindexed portion of the document is already loaded in memory at

the local node, delivered via the distributed file system.

Thanks to Prefix-Filtering, the Map cost of our proposed algorithm smaller than ELSA, yet larger than VERN. For the same reason, the shuffle size of SSJ-2 is smaller than ELSA but still larger than VERN, if we assume that $\overline{p}$ is not significantly smaller than $\overline{l}$. For SSJ-2R, the shuffle size is probably its weakest point. Shuffling the whole collection within the MapReduce data flow increases the volume of intermediate data. The Reduce function of SSJ-2 needs to remotely recover the documents being processed to retrieve their unindexed portions. This dramatically increases its cost beyond ELSA but can hardly be compared with VERN. SSJ-2R has about the same cost as ELSA when assuming that the remainder file is already loaded in memory.

It is difficult to combine the costs of the various phases of the four algorithms. Therefore, it is not possible to understand which is the best. We can conclude that Elsayed et al. is the worst of the four, due to the non exploitation of the Prefix-Filtering, and that the impact of the shuffling phase will determine the goodness of our proposed algorithms. In the experimental section, we evaluate empirically the efficiency of the four algorithms in the three steps of the MapReduce framework.

3.6 Experimental evaluation

In this section we describe the performance evaluation of the algorithms. We used several subsets of the TREC WT10G Web corpus. The original dataset has 1,692,096 english language documents. The size of the entire uncompressed collection is around 10GiB. In Table 3.3 we describe the samples of the collection we used, ranging from $\sim$ 17k to $\sim$ 63k documents. We preprocessed the data to prepare it for analysis. We parsed the dataset stripping HTML, removed stop-words, performed stemming and vectorization of the input and extracted the lexicon. We also sorted the features inside each document in decreasing order of term frequency in order to effectively utilize Prefix-Filtering.

We ran the experiments on a 5-node cluster. Each node was equipped with two Intel Xeon E5520 CPUs @2.27GHz, with 8 virtual cores (for a

Table 3.3: Samples from the TREC WT10G collection.

	D17K	D30K	D63K
# documents	17,024	30,683	63,126
# terms	183,467	297,227	580,915
# all pairs	289,816,576	941,446,489	3,984,891,876
# similar pairs	94,220	138,816	189,969

total of 80 virtual cores), a 2TiB disk, 8GiB of RAM, and Gigabit Ethernet.

We used one of the nodes to run Hadoop's master daemons (NameNode and JobTracker), the rest were configured as slaves running DataNode and TaskTracker daemons. Two of the virtual cores on each slave machine were reserved to run the daemons, the rest were equally split among map and reduce slots (7 each), for a total of 28 slots for each phase.

We tuned Hadoop's configuration as follows: we allocated 1GiB of memory to each daemon and 400MiB to each task, changed the block size of HDFS to 256MiB and the file buffer size to 128KiB, disabled speculative execution and enabled JVM reuse and map output compression.

For each algorithm, we wrote an appropriate combiner to reduce the shuffle size (a combiner is a reduce-like function that runs inside the mapper to aggregate partial results). For SSJ-2 and SSJ-2R the combiner perform the sums of partial scores in the values, according to the same logic used in the reducer. We also implemented raw comparators for every key type used in order to get better performance (raw comparators compare keys during sorting without deserializing them into objects).

3.6.1 Running time

We compared the four algorithms described in the previous sections. The two baselines ELSA and VERN and the two algorithms proposed by us: SSJ-2 and SSJ-2R. VERN would require an additional step for removing duplicate similar pairs. We did not need to implement this step thanks to the strategy described in Section 3.3.2 Therefore our analysis does not take into account the overhead induced by such duplicate removal.

For SSJ-2R we used a partitioning factor of the remainder file $K = 4$.

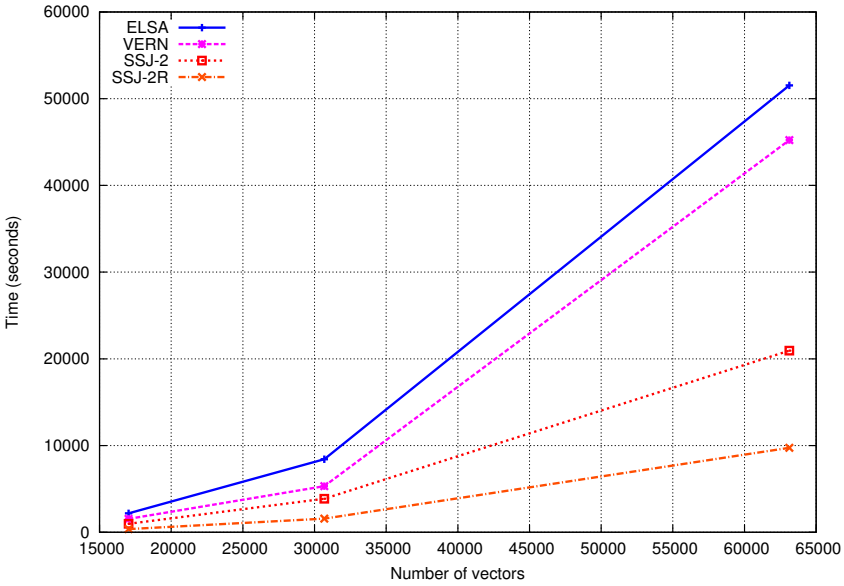

Figure 3.6: Running time.

We enabled partitioning just for the sake of completeness, given that the mappers were not even close to fill the available memory.

We use 56 mappers and 28 reducers, so that the mappers finish in two waves and all the reducers can start copying and sorting the partial results while the second wave of mappers is running. For all the experiments, we set the similarity threshold $\overline{\sigma} = 0.9$.

Figure 3.6 shows the running times of the four different algorithms. All of them have a quadratic step, so doubling the size of the input roughly multiplies by 4 the running time. Unexpectedly, VERN does not improve significantly over ELSA. This means that Prefix-Filtering is not fully exploited. Recall ELSA uses a simple Term-Filtering and compares any document pair sharing at least one term. Both our proposed algorithms outperform the two baselines. In particular, SSJ-2 is more than twice faster than VERN and SSJ-2R is about 4.5 times faster. Notice that

SSJ-2R is twice as fast compared to SSJ-2, which means that the improvement given by the use of the remainder file is significant.

We tried to fit a simple power law model $f(x) = ax^b$ to the running time of the algorithms. For all algorithms but VERN the parameter $b \approxeq 2.5$, while $a \approxeq 10^{-8}$ with ELSA having the largest one. VERN has a smaller constant $a \approxeq 10^{-10}$ with a larger $b \approxeq 2.9$. However, given the small number of data points available the analysis is not conclusive.

3.6.2 Map phase

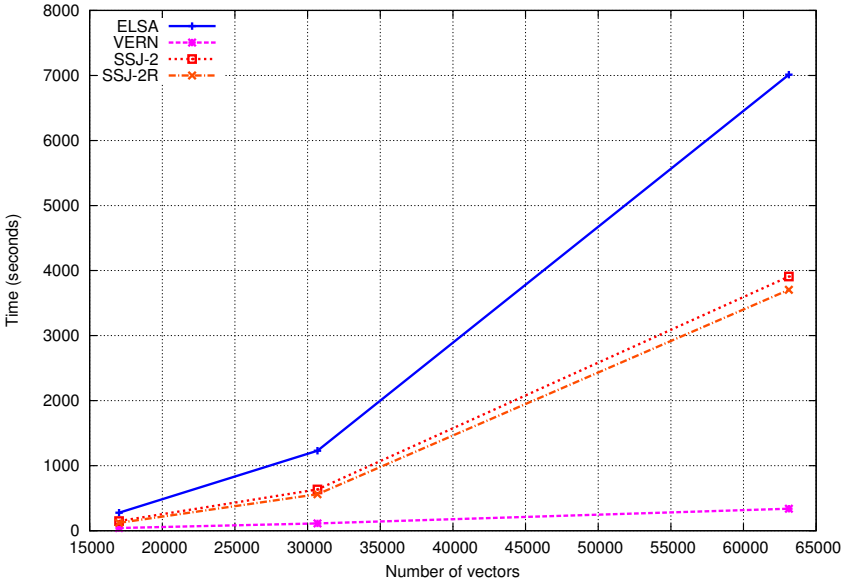

Figure 3.7: Average mapper completion time.

In Figure 3.7 we report the average map times for each algorithm. Clearly VERN is the fastest. The mapper only replicates input documents. The time required by the other algorithms grows quadratically as expected. The two algorithms we propose are two times faster than ELSA for two main reasons.

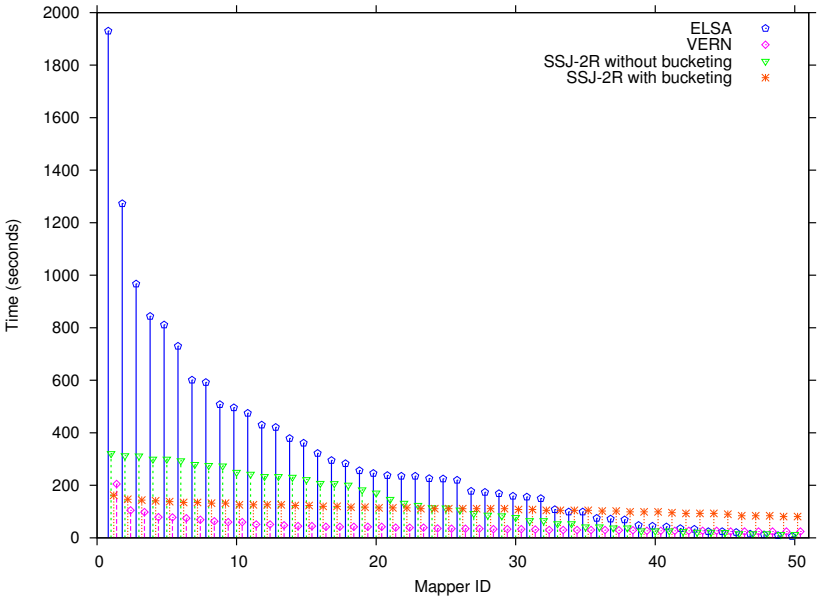

Figure 3.8: Mapper completion time distribution.

First, Prefix-Filtering reduces the maximum inverted list length. Even a small improvement here brings a quadratic gain. Since Prefix-Filtering removes the most frequent terms, the longest inverted lists are shortened or even removed. For D17K, the length of the longest inverted list is reduced from 6600 to 1729 thanks to Prefix-Filtering, as shown in Figure 3.9. The time to process such inverted list decreases dramatically, and also the amount of intermediate data generated is reduced. Interestingly, Prefix-Filtering does not affect significantly the index size. Only 10% of the index is pruned as reported in Table 3.4, meaning that its effects are limited to shortening the longest inverted lists.

Second, we employ a bucketing technique to improve load balancing. Since mappers process the inverted lists in chunks, the longest inverted lists might end up in the same chunk and be processed by a single mapper. SSJ-2 and SSJ-2R randomly hash the inverted lists into different

Table 3.4: Statistics for the four algorithms on the three datasets.

Dataset	Algorithm	# evaluated pairs (M)	Index size (MB)	Remainder size (MB)	Shuffle size (GB)	Running time (s)	Avg. map time (s)	Std. map time (%)	Avg. reduce time (s)	Std. reduce time (%)
D17K	ELSA	109	46		3.3	2,187	276	127.50	**49**	16.68
	VERN	401			3.7	1,543	**42**	68.10	892	22.20
	SSJ-2	**65**	41		**2.1**	971	148	37.92	575	**3.58**
	SSJ-2R	**65**	41	4.7	2.6	**364**	122	**26.50**	**49**	15.50
D30K	ELSA	346	92		11.3	8,430	1,230	132.08	**82**	15.91
	VERN	1,586			8.3	5,313	**112**	68.10	3,847	13.94
	SSJ-2	**224**	82		**8.1**	3,862	635	32.06	2,183	**5.81**
	SSJ-2R	**224**	82	8.2	10.5	**1,571**	560	**23.41**	155	15.50
D63K	ELSA	1,519	189		49.2	51,540	7,013	136.39	1,136	11.84
	VERN	6,871			**20.7**	28,534	**338**	59.61	16,849	9.36
	SSJ-2	**1,035**	170		35.8	20,944	3,908	24.32	11,328	**2.76**
	SSJ-2R	**1,035**	170	15.6	49.7	**9,745**	3,704	**20.26**	**846**	12.11

buckets, so that the longest lists are likely spread among all the mappers. Figure 3.8 shows the completion time of the mappers for the D17K dataset. VERN has the lowest average mapper completion time, as all the work is done in the reducer. However, due to skew in the input, the slowest mapper in VERN is slower than in SSJ-2R with bucketing. This can happen if there are a few very dense documents in the input, so that VERN needs to create a large number of copies of each of these document. ELSA is in any case slower than SSJ-2R without bucketing just because of the maximum inverted list length. Nevertheless, completion times are not evenly distributed in ELSA. This skew induces a strong load imbalance, forcing all the reducers to wait the slowest mapper before starting. Bucketing solves this issue by evenly spreading the load among mappers. As a result, the running time of the slowest mapper is almost halved. The standard deviation of map completion times for all algorithms is reported in Table 3.4.

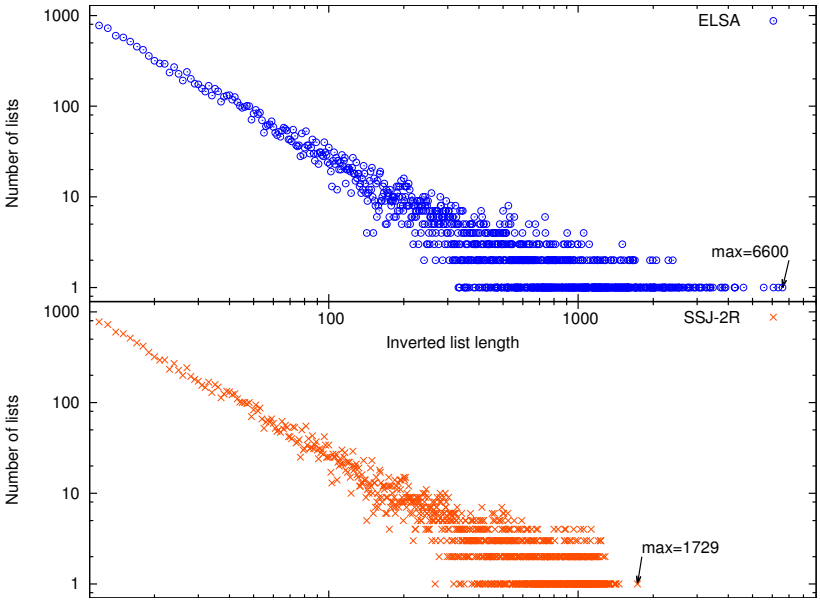

Figure 3.9: Effect of Prefix-filtering on inverted list length distribution.

3.6.3 Shuffle size

Figure 3.10 shows the shuffles sizes for the various algorithms when varying the input size. The shuffle size is affected by the combiners, that compact the key-value pairs emitted by the mappers into partial results by applying a reduce-like function. VERN has the smallest shuffle size. We intuitively expected that the replication performed during the map phase would generate a large amount of intermediate data. Conversely, the algorithms based on the inverted index generate the largest volume of data. This is consistent with the analysis in the previous section as VERN is the only algorithm without a quadratic term in the shuffle size.

More specifically, SSJ-2 produces less data thanks to Prefix-Filtering. However, SSJ-2R shuffles about the same data as ELSA due to the additional information needed the replication introduced by the partitioning of the remainder file. The shuffles size does not include the remainder

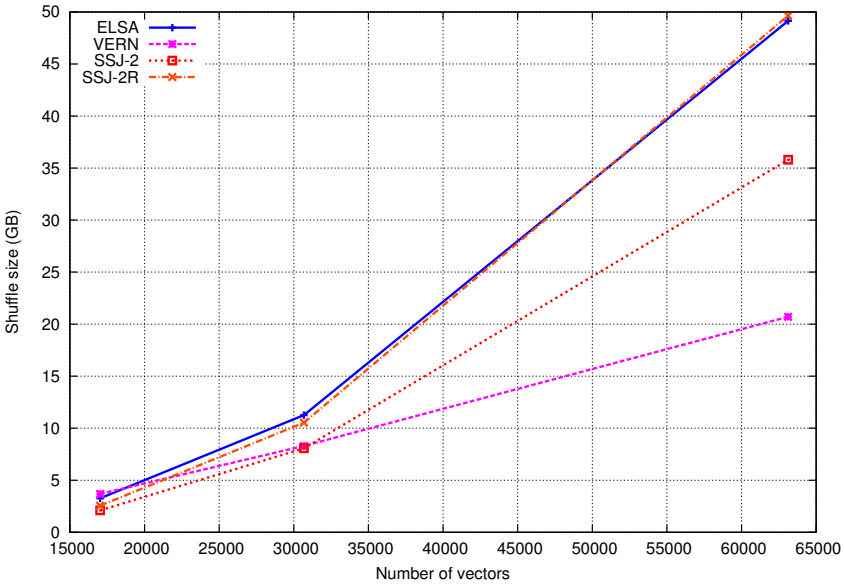

Figure 3.10: Shuffle size.

file, which is anyway small enough to be negligible as discussed below. Although SSJ-2R and ELSA have the same shuffle size, our algorithm is almost five times faster overall.

3.6.4 Reduce phase

Figure 3.11 shows the average running time of the reduce function. VERN is the most expensive as expected. While the other algorithms take advantage of the MR infrastructure to perform partial similarity computations, VERN just delivers to the reducer a collection of potentially similar documents. The similarity computation is entirely done at the reducer, with a cost quadratic in the number of documents received. In addition, the same document pair is evaluated several times at different reducers at least to check if the reducer is responsible for computing the similarity, thus increasing the total cost of the reduce phase.

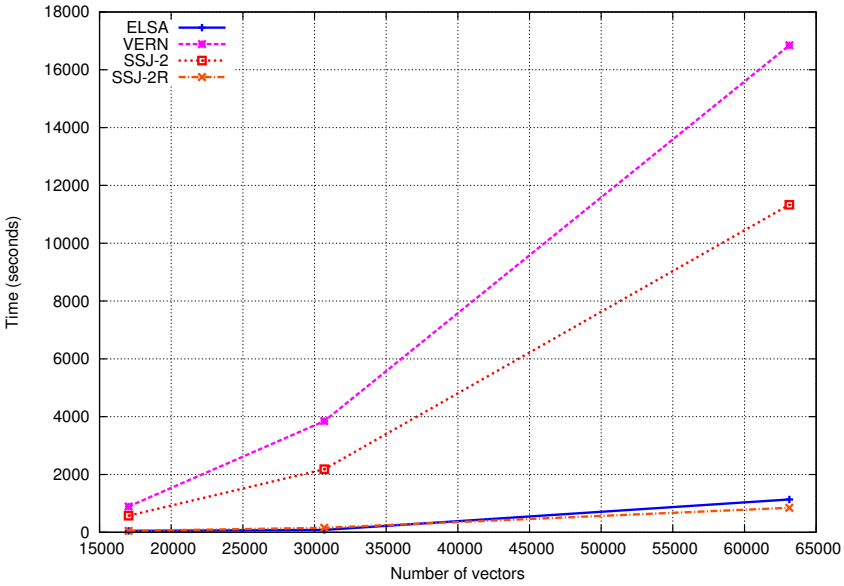

Figure 3.11: Average reducer completion time.

SSJ-2 is the second most expensive due to the remote retrieval of documents from the distributed file system.

Both SSJ-2R and ELSA are way faster compared to the other two algorithms. The reducer performs very little computation, the contributions from the various terms are simply summed up. The use of the remainder file significantly speeds up SSJ-2R compared to SSJ-2. The remainder file is quickly retrieved from the distributed file system and no random remote access is performed.

The performance of the reducer depends heavily on the number of candidate pairs evaluated. As shown in Table 3.4, both SSJ-2 and SSJ-2R evaluate about 33% less candidate than ELSA. VERN evaluates a number of pairs which is more than 5 times larger than SSJ-2 and SSJ-2R because of the replication of the input sent to the reducers. Even though all but one of these pairs are discarded, they still need to be checked.

As reported in Table 3.4, all reduce times have small standard deviation, which means that the load is well balanced across the reducers.

3.6.5 Partitioning the remainder file

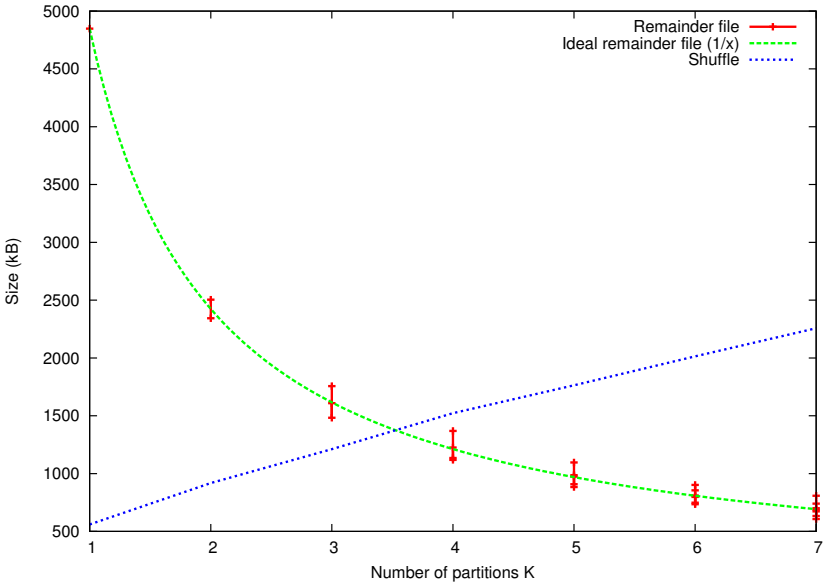

Figure 3.12: Remainder file and shuffle size varying K.

We have seen that using the remainder file in SSJ-2R almost halves the running time of SSJ-2. This result leverages the fact that the portion of information which is needed by every node can be easily broadcast via the distributed file system rather than inside the MR data flow. In Table 3.4, we report the size of the remainder file which is always about 10% of the inverted index when using a similarity threshold $\overline{\sigma} = 0.9$. Such limited size was not an issue in our experimental setting. However, this might become an issue with larger collections, or when many reducers run on the same node and share the same memory.

Figure 3.12 illustrates a small experiment on a single node with the D17K dataset, where we varied the number K of chunks of the remainder file. Since partitioning is done in the document identifier space, the size of each chunk is not exactly the same and depends on the distribution of document sizes. However, in our experiments the variance is quite low. This proves experimentally that a reducer needs to load in memory only a small chunk, thus allowing SSJ-2R to scale to very large collections. Figure 3.12 also shows the trade-off between shuffle size and remainder file size when increasing K. For each additional chunk we need to shuffle one extra copy of the input, so shuffle size increases linearly with K.

3.7 Conclusions

Finding similar items in a bag of documents is a challenging problem that arises in many applications in the areas of Web mining and information retrieval. The size of Web-related problems mandates the use of parallel approaches in order to achieve reasonable computing times.

In this chapter we presented two novel algorithms for the MapReduce framework. We showed that designing efficient algorithms for MR is not trivial and requires careful blending of several factors to effectively capitalize on the available parallelism. We analyzed the complexity of our algorithms, and compared it with the state-of-the-art by examining the map, shuffle and reduce phases. We validated this theoretical analysis with experimental evidence collected on a sample of the Web.

SSJ-2R borrows from previous approaches based on inverted index, and embeds pruning strategies that have been used only in sequential algorithms so far. We exploited the underlying distributed filesystem to support communication patterns that do not naturally fit the MR framework. We also described a partitioning strategy that overcomes memory limitations at the cost of an increased volume of intermediate data. SSJ-2R achieves scalability without sacrificing precision by exploiting each MR phase to perform useful work. Thanks to a careful design that takes into account the specific properties of MR, SSJ-2R outperforms the state-of-the-art by a factor of about 4.5.

Chapter 4

Social Content Matching in MapReduce

Matching is a classical problem in graph theory. It entails finding a subset of edges that satisfy a bound on the number of shared vertexes. This kind of problem falls in the second category of the taxonomy defined in Chapter 1 and requires iterative solutions. Matching problems are ubiquitous. They occur in economic markets and Web advertising, to name a few. In this chapter we focus on an application of matching for social media. Our goal is to distribute content from information suppliers to information consumers. We seek to maximize the overall relevance of the matched content from suppliers to consumers while regulating the overall activity, e.g., ensuring that no consumer is overwhelmed with data and that all suppliers have chances to deliver their content.

We propose two MapReduce matching algorithms: GREEDYMR and STACKMR. Both algorithms have provable approximation guarantees, and in practice they produce high-quality solutions. While both algorithms scale extremely well, we can show that STACKMR requires only a poly-logarithmic number of MapReduce steps, making it an attractive option for applications with very large datasets. We experimentally show the trade-offs between quality and efficiency of our solutions on two large datasets coming from real-world social media Web sites.

4.1 Introduction

The last decade has witnessed a radical paradigm shift on how information content is distributed among people. Traditionally, most of the information content has been produced by few specialized agents and consumed by the big masses. Nowadays, an increasing number of platforms allow everyone to participate both in information production and in information consumption. The phenomenon has been coined as *democratization of content*. The Internet, and its younger children, user-generated content and social media, have had a major role in this paradigm shift.

Blogs, micro-blogs, social-bookmarking sites, photo-sharing systems, and question-answering portals, are some of the social media that people participate in as both information suppliers and information consumers. In such social systems, not only consumers have many opportunities to find relevant content, but also suppliers have opportunities to find the right audience for their content and receive appropriate feedback. However, as the opportunities to find relevant information and relevant audience increase, so does the complexity of a system that would allow suppliers and consumers to meet in the most efficient way.

Our motivation is building a *"featured item"* component for social-media applications. Such a component would provide recommendations to consumers each time they log in the system. For example, flickr[1] displays photos to users when they enter their personal pages, while Yahoo! Answers[2] displays questions that are still open for answering. For consumers it is desirable that the recommendations are of high quality and relevant to their interests. For suppliers it is desirable that their content is delivered to consumers who are interested in it and may provide useful feedback. In this way, both consumers and suppliers are more satisfied by using the system and they get the best out of it.

Naturally, we model this problem as a matching problem. We associate a *relevance score* to each potential match of an item t to a user u. This score can be seen as the weight of the edge (t, u) of the bipartite

[1] `http://flickr.com`
[2] `http://answers.yahoo.com`

graph between items and users. For each item t and each user u we also consider constraints on the maximum number of edges that t and u can participate in the matching. These *capacity constraints* can be estimated by the activity of each user and the relative frequency with which items need to be delivered. The goal is to find a matching that satisfies all capacity constraints and maximizes the total weight of the edges in the matching. This problem is known as *b-matching* .

The b-matching problem can be solved exactly in polynomial time by max-flow techniques. However, the fastest exact algorithms today have complexity $\tilde{O}(nm)$ (Gabow, 1983; Goldberg and Rao, 1998), for graphs with n nodes and m edges, and thus do not scale to large datasets. Instead, in this work we focus on approximation algorithms that are scalable to very large datasets. We propose two algorithms for b-matching, STACKMR and GREEDYMR, which can be implemented efficiently in the MapReduce paradigm. While both our algorithms have provable approximation guarantees, they have different properties.

We design the STACKMR algorithm by drawing inspiration from existing distributed algorithms for matching problems (Garrido et al., 1996; Panconesi and Sozio, 2010). STACKMR is allowed to violate capacity constraints by a factor of $(1 + \epsilon)$, and yields an approximation guarantee of $\frac{1}{6+\epsilon}$, for any $\epsilon > 0$. We show that STACKMR requires a poly-logarithmic number of MapReduce steps. This makes STACKMR appropriate for realistic scenarios with large datasets. We also study a variant of STACKMR, called STACKGREEDYMR, in which we incorporate a greedy heuristic in order to obtain higher-quality results.

On the other hand, GREEDYMR is simpler to implement and has the desirable property that it can be stopped at any time and provide the current best solution. GREEDYMR is a $1/2$-approximation algorithm, so it has a better quality guarantee than STACKMR. GREEDYMR also yields better solutions in practice. However, it cannot guarantee a poly-logarithmic number of steps. A simple example shows that GREEDYMR may require a linear number of steps. Although GREEDYMR is theoretically less attractive than STACKMR, in practice it is a very efficient algorithm, and its performance is way far from the worst case.

Finally, we note that the b-matching algorithm takes as input the set of candidate edges weighted by their relevance scores. In some cases, this set of candidate edges is small, for instance when items are recommended only among friends in a social network. In other applications, any item can be delivered to any user, e.g., a user in flickr may view a photo of any other user. In the latter case, the graph is not explicit, and we need to operate on a bag of items and consumers. In this case, materializing all pairs of item-user edges is an unfeasible task. Thus, we equip our framework with a scheme that finds all edges with score greater than some threshold σ, and we restrict the matching to those edges. We use the SSJ-2R algorithm presented in Chapter 3 to solve the problem of finding all similar item-user pairs efficiently in MapReduce.

Our main contributions are the following.

- We investigate the problem of b-matching in the context of social content distribution, and devise a MapReduce framework to address it.
- We develop STACKMR, an efficient variant of the algorithm presented by Panconesi and Sozio (2010). We demonstrate how to adapt such an algorithm in MapReduce, while requiring only a poly-logarithmic number of steps. Our experiments show that STACKMR scales excellently to very large datasets.
- We introduce GREEDYMR, a MapReduce adaptation of a classical greedy algorithm. It has a $1/2$-approximation guarantee, and is very efficient in practice.
- We employ SSJ-2R to build the input graph for the b-matching.
- We perform a thorough experimental evaluation using large datasets extracted from real-world scenarios.

The rest of the chapter is organized as follows. In Section 4.3 we formally define the graph matching problem. In Section 4.4 we present the application scenario to social content distribution that we consider. In Section 4.5 we discuss the algorithms and their MR implementation. Finally, we present our experimental evaluation in Section 4.6.

4.2 Related work

The general problem of assigning entities to users so to satisfy some constraints on the overall assignment arises in many different research areas of computer science. Entities could be advertisements (Charles et al., 2010), items in an auction (Penn and Tennenholtz, 2000), scientific papers (Garg et al., 2010) or media content, like in our case. The b-matching problem finds applications also in machine learning (Jebara et al., 2009) and in particular in spectral clustering (Jebara and Shchogolev, 2006).

The weighted b-matching problem can be solved in polynomial time by employing maximum flow techniques (Gabow, 1983; Goldberg and Rao, 1998). In any case, the time complexity is still superlinear in the worst case. Christiano et al. (2010) have recently developed a faster approximation algorithm based on electrical flows.

In a distributed environment, there are some results for the unweighted version of the (simple) matching problem (Fischer et al., 1993; Garrido et al., 1996), while for the weighted case the approximation guarantee has progressively improved from $1/5$ (Wattenhofer and Wattenhofer, 2004) to $1/2$ (Lotker et al., 2008). For distributed weighted b-matching, a $1/2$-approximation algorithm was developed by Koufogiannakis and Young (2009). However, a MapReduce implementation is non-obvious. Lattanzi et al. (2011) recently proposed a MapReduce algorithm for b-matching.

4.3 Problem definition

In this section we introduce our notation and provide our problem formulation. We are given a set of content items $T = \{t_1, \ldots, t_n\}$, which are to be delivered to a set of consumers $C = \{c_1, \ldots, c_m\}$. For each t_i and c_j, we assume we are able to measure the interest of consumer c_j in item t_i with a positive weight $w(t_i, c_j)$. The distribution of the items T to the consumers C can be clearly seen as a matching problem on the bipartite graph with nodes T and C, and edge weights $w(t_i, c_j)$.

In order to avoid that each consumer c_j receive too many items, we enforce a capacity constraint $b(c_j)$ on the number of items that are matched

to c_j. Similarly, we would like to avoid the scenario when only a few items (e.g. the most popular ones) participate in the matching. To this end, we introduce a capacity constraint $b(t_i)$ on the number of consumers that each item t_i is matched to.

This variant of the matching problem is well known in the theoretical computer science community as the b-matching problem. This is defined as follows. We are given an undirected graph $G = (V, E)$, a function $b : V \to \mathbb{N}$ expressing node capacities (or budgets) and another function $w : E \to \mathbb{R}^+$ expressing edge weights. A b-matching in G is a subset of E such that for each node $v \in V$ at most $b(v)$ edges incident to v are in the matching. We wish to find a b-matching of maximum weight.

Although our algorithms work with any undirected graph, we focus on bipartite graphs which are relevant to our application scenarios. The problem we shall consider in the rest of the chapter is defined as follows.

Problem 2 (Social Content Matching Problem)**.** *We are given an undirected bipartite graph $G = (T, C, E)$, where T represents a set of items and C represents a set of consumers, a weight function $w : E \to \mathbb{R}^+$, as well as a capacity function $b : T \cup C \to \mathbb{N}$. A b-matching in G is a subset of E such that for each node $v \in T \cup C$ at most $b(v)$ edges incident to v are in the matching. We wish to find a b-matching of maximum weight.*

4.4 Application scenarios

To instantiate the problem we just defined, we need to (i) define the weights $w(t_i, c_j)$ between items t_i and consumers c_j, (ii) decide the set of potential edges that participates in the matching, and (iii) define the capacity constraints $b(t_i)$ and $b(c_j)$. In our work we focus only on the matching algorithm and we assume that addressing the details of the above questions depends on the application. However, for completeness we discuss our thoughts on these issues.

Scenario. We envision a scenario in which an application operates in consecutive phases. Depending on the dynamics of the application, the duration of each phase may range from hours to days. Before the beginning of the i-th phase the application makes a tentative allocation of

which items will be delivered to which consumers during the i-th phase. The items that participate in this allocation, i.e., the set T of Problem 2, are those that have been produced during the $(i-1)$-th phase, and perhaps other items that have not been distributed in previous phases.

Edge weights. A simple approach is to represent items and consumers in a vector space, i.e., items t_i and consumers c_j are represented by *feature vectors* $\mathbf{v}(t_i)$ and $\mathbf{v}(c_j)$. Then we can define the edge weight $w(t_i, c_j)$ using the cosine similarity $w(t_i, c_j) = \mathbf{v}(t_i) \cdot \mathbf{v}(c_j)/(\|\mathbf{v}(t_i)\| \cdot \|\mathbf{v}(c_j)\|)$. Potentially, more complex similarity functions can be used. Borrowing ideas from information retrieval, the features in the vector representation can be weighted by `tf·idf` scores. Alternatively, the weights $w(t_i, c_j)$ could be the output of a recommendation system that takes into account user preferences and user activities.

Candidate edges. With respect to deciding which edges to consider for matching, the simplest approach is to consider all possible pairs (t_i, c_j). This is particularly attractive, since we let the decision of selecting edges entirely to the matching algorithm. However, considering $O(|T||C|)$ edges makes the system highly inefficient. Thus, we opt for methods that prune the number of candidate edges. Our approach is to consider as candidates only edges whose weight $w(t_i, c_j)$ is above a threshold σ. The rationale is that since the matching algorithm will seek to maximize the total edge weight, we preferably discard low-weight edges.

We note that depending on the application, there may be other ways to define the set of candidate edges. For example, in social-networking sites it is common for consumers to subscribe to suppliers they are interested in. In such an application, we consider only candidate edges (t_i, c_j) for which c_j has subscribed to the supplier of t_i.

Capacity constraints. The consumer capacity constraints express the number of items that need to be displayed to each consumer. For example, if we display one different item to a consumer each time they access the application, $b(c_j)$ can be set to an estimate of the number of times that consumer c_j will access the application during the i-th phase. Such an estimate can be obtained from log data.

For the item capacity constraints, we observe that $B = \sum_{c \in C} b(c)$ is

an upper bound on the total number of distributed items, so we require $B = \sum_{t \in T} b(t)$ as well. Now we distinguish two cases, depending on whether there is a quality assessment on the items T or not. If there is no quality assessment, all items are considered equivalent, and the total distribution bandwidth B can be divided equally among all items, so $b(t) = \max\{1, \frac{B}{|T|}\}$, for all t in T.

If on the other hand there is a quality assessment on the items T, we assume a quality estimate $q(t)$ for each item t. Such an estimate can be computed using a machine-learning approach, as the one proposed by Agichtein et al. (2008), which involves various features like content, links, and reputation. Without loss of generality we assume normalized scores, i.e., $\sum_{t \in T} q(t) = 1$. We can then divide the total distribution bandwidth B among all items in proportion to their quality score, so $b(t) = \max\{1, q(t) \cdot B\}$. In a real-application scenario, the designers of the application may want to control the function $q(t)$ so that it satisfies certain properties, for instance, it follows a power-law distribution.

4.5 Algorithms

4.5.1 Computing the set of candidate edges

The first step of our algorithm is to compute the set of candidate edges, which in Section 4.4 were defined to be the edges with weight $w(t_i, c_j)$ above a threshold σ. This step is crucial in order to avoid considering $O(|T||C|)$ edges, which would make the algorithm impractical.

As we have already seen in the previous chapter, the problem of finding all the pairs of $t_i \in T$ and $c_j \in C$ so that $w(t_i, c_j) \geq \sigma$ is known as the similarity join problem. Since we aim at developing the complete system in the MapReduce framework, we obviously take advantage of SSJ-2R.

In particular, we adapt SSJ-2R to compute the similarity between item-consumer pairs. First, we build a document bag by interpreting the items t_i and the consumers c_j as documents via their vector representation. Second, SSJ-2R can be trivially modified to join the two sets T and C without considering pairs between two items or two consumers.

4.5.2 The STACKMR algorithm

Our first matching algorithm, STACKMR, is a variant of the algorithm developed by Panconesi and Sozio (2010). Panconesi and Sozio propose a complex mechanism to ensure that capacity constraints are satisfied, which unfortunately does not seem to have an efficient implementation in MapReduce. Here we devise a more practical variant that allows node capacities to be violated by a factor of at most $(1 + \epsilon)$, for any $\epsilon > 0$. This approximation is a small price to pay in our application scenarios, where small capacity violations can be tolerated.

For the sake of presentation, we describe our algorithm first in a centralized environment and then in a parallel environment. Pseudocode for STACKMR and for the algorithm by Panconesi and Sozio are shown in Algorithms 1 and 2, respectively.

The latter one has been slightly changed so to take into account implementation issues. However, we do not include an evaluation of the latter algorithm as it does not seem to be efficient. In the next section we describe in detail how to implement the former algorithm in MapReduce.

Our algorithm is based on the *primal-dual schema*, a successful and established technique to develop approximation algorithms. The primal-dual schema has proved to play an important role in the design of sequential and distributed approximation algorithms. We hold the belief that primal-dual algorithms bear the potential of playing an important role in the design of MapReduce algorithms as well.

The first step of any primal-dual algorithm is to formulate the problem at hand as an integer linear program (IP). Each element that might be included in a solution is associated with a 0-1 variable. The combinatorial structure of the problem is captured by an objective function and a set of constraints, both being a linear combination of the binary variables.

Then, integrality constraints are relaxed so that variables can take any value in the range $[0, 1]$. This linear program is called *primal*. From the primal program we derive the so-called *dual*. There is a direct correspondence between the variables of the primal and the constraints of the dual, as well as, the variables of the dual and the constraints of the primal.

Algorithm 1 STACKMR violating capacity constraints by a factor of at most (1+ϵ)

1: /* Pushing Stage */
2: **while** E is non empty **do**
3: Compute a maximal $\lceil \epsilon b \rceil$-matching M (each vertex v has capacity $\lceil \epsilon b(v) \rceil$), using the procedure in Garrido et al. (1996);
4: Push all edges of M on the distributed stack (M becomes a layer of the stack);
5: **for** all $e \in M$ in parallel **do**
6: Let $\delta(e) = (w(e) - y_u/b(u) - y_v/b(v))/2$;
7: increase y_u and y_v by $\delta(e)$;
8: **end for**
9: Update E by eliminating all edges that have become weakly covered
10: **end while**
11: /* Popping Stage */
12: **while** the distributed stack is nonempty **do**
13: Pop a layer M out of the distributed stack.
14: In parallel include all edges of M in the solution.
15: For each vertex v: let $\bar{b}(v)$ be the set of edges in M that are incident to v; update $b(v) \leftarrow b(v) - \bar{b}(v)$; if $b(v) \leq 0$ then remove all edges incident to v.
16: **end while**

A typical primal-dual algorithm proceeds as follows: at each step dual variables are raised and as soon as a dual constraint is satisfied with equality (or almost) the corresponding primal variable is set to one. The above procedure is iterated until no dual variable can be increased further without violating a dual constraint. The approximation guarantee follows from the fact that any feasible dual solution gives an upper bound on any optimum solution for the primal. Thus, the closer these two quantities the better the approximation guarantee.

We first present our algorithm in a centralized environment and later proceed to show how to adapt it in MapReduce. Let us now give the IP formulation for the b-matching problem.

Algorithm 2 STACKMR satisfying all capacity constraints

/* Pushing Stage */
while E is non empty **do**
  Compute a maximal $\lceil \epsilon b \rceil$-matching M (each vertex v has capacity $\lceil \epsilon b(v) \rceil$), using the procedure in Garrido et al. (1996);
  Push all edges of M on the distributed stack (M becomes a layer of the stack);
  for all $e \in M$ in parallel **do**
    Let $\delta(e) = (w(e) - y_u/b(u) - y_v/b(v))/2$;
    increase y_u and y_v by $\delta(e)$;
  end for
  Update E by eliminating all edges that have become weakly covered
end while
/* Popping Stage */
while the distributed stack is nonempty **do**
  Pop a layer M out of the distributed stack.
  In parallel tentatively include all edges of M in the solution.
  If there is a vertex v whose capacity is exceeded then mark all edges in M incident to v as *overflow*, remove them from the solution and remove all edges in $E \setminus M$ incident to v from the graph.
  For each vertex v: let $\bar{b}(v)$ be the number of edges in M that are incident to v; update $b(v) \leftarrow b(v) - \bar{b}(v)$; if $b(v) = 0$ then remove all edges incident to v.
end while
/* Computing a feasible solution */
while there are *overflow* edges **do**
  Let $\bar{\mathcal{L}}$ be the set of overflow edges such that for every $e \in \bar{\mathcal{L}}$ there is no overflow edge f, incompatible with e, for which $\delta(f) > (1+\epsilon)\delta(e)$.
  Compute a maximal b-matching $\bar{M}$ including only edges of $\bar{\mathcal{L}}$ ($\bar{M}$ shall be referred to as a sublayer of the stack);
  In parallel, for each edge $e \in \bar{M}$, include e in the solution if this maintains the solution feasible.
  For each vertex v: let $\bar{b}(v)$ be the set of edges in $\bar{M}$ that are incident to v (these edges are included in the solution); update $b(v) \leftarrow b(v) - \bar{b}(v)$; if $b(v) \leq 0$ then remove from the set of overflow edges all edges incident to v.
  Remove all edges in $\bar{M}$ from the set of overflow edges.
end while

$$\text{maximize} \quad \sum_{e \in E} w(e) x_e \tag{IP}$$

$$\text{such that} \quad \sum_{e \in E,\, v \in e} x_e \leq b(v) \qquad \forall v \in V, \tag{4.1}$$

where $x_e \in \{0, 1\}$ is associated to edge e, and a value of 1 means that e belongs to the solution. The dual program is as follows.

$$\text{minimize} \quad \sum_{v \in V} y_v \tag{DP}$$

$$\text{such that} \quad \frac{y_u}{b(u)} + \frac{y_v}{b(v)} \geq w(e) \qquad \forall e = (u, v) \in E, \tag{4.2}$$

$$y_v \geq 0 \qquad \forall v \in V. \tag{4.3}$$

Dual constraints (4.2) are associated with edges e. An edge is said to be *covered* if its corresponding constraint is satisfied with equality. The variables occurring in such a constraint are referred as e's dual variables and play an important role in the execution of the algorithm.

The centralized algorithm consists of two phases: a *push phase* where edges are pushed on a stack in arbitrary order, and a *pop phase* where edges are popped from the stack and a feasible solution is computed. When an edge $e(u, v)$ is pushed on the stack, each of its dual variables is increased by the same amount $\delta(e)$ so to satisfy Equation (4.2) with equality. The amount $\delta(e)$ is derived from Equation (4.2) as

$$\delta(e) = \frac{(w(e) - y_u/b(u) - y_v/b(v))}{2}. \tag{4.4}$$

Whenever edges become covered they are deleted from the input graph. The push phase terminates when no edge is left. In the pop phase, edges are successively popped out of the stack and included in the solution if feasibility is maintained.

In a parallel environment, we wish to parallelize as many operations as possible so to ensure poly-logarithmic running time. Thus, we need a mechanism to bound the number of push and pop steps, which in the

centralized algorithm may be linear in the number of edges. This is done by computing at each step a *maximal* $\lceil \epsilon b \rceil$-matching. Note the difference between maximum and maximal: a b-matching is maximal if and only if it is not properly contained in any other b-matching. All edges in a maximal set, called a *layer* of the stack, are pushed on the stack in parallel. In the popping phase, all edges within the same layer are popped out of the stack and included in the solution in parallel. Edges of nodes whose capacity constraints are satisfied or violated are deleted from the stack and ignored from further consideration. A maximal b-matching can be computed efficiently in MapReduce as we will discuss in Section 4.5.3.

Unfortunately, the total number of layers may still be linear in the maximum degree of a node. To circumvent this problem, we introduce the definition of *weakly covered edges*. Roughly speaking, a weakly covered edge is an edge whose constraint is only "partially satisfied" and thus gets covered after a small number of iterations.

Definition 1. (Weakly covered edges) *Given $\epsilon > 0$, at any time during the execution of our algorithm we say that an edge $e \in E$ is* weakly covered *if constraint (4.2) for $e = uv$ is such that*

$$\frac{\bar{y}_u}{b(u)} + \frac{\bar{y}_v}{b(v)} \geq \frac{1}{3 + 2\epsilon} w(e), \tag{4.5}$$

where $\bar{y}$ denotes the current value of y.

Observe that Equation (4.5) is derived from Equation (4.2).

To summarize, our parallel algorithm proceeds as follows. At each step of the push phase, we compute a maximal $\lceil \epsilon b \rceil$-matching using the procedure by Garrido et al.. All the edges in the maximal matching are then pushed on the stack in parallel forming a layer of the stack. For each of these edges we increase each of its dual variable by $\delta(e)$ in parallel. Some edges might then become weakly covered and are deleted from the input graph. The push phase is executed until no edge is left.

At the end of the push phase, layers are iteratively popped out of the stack and edges within the same layer are included in the solution in parallel. This can violate node capacities by a factor of at most $(1 + \epsilon)$, as every layer contains at most $\epsilon b(v)$ edges incident to any node v. Edges

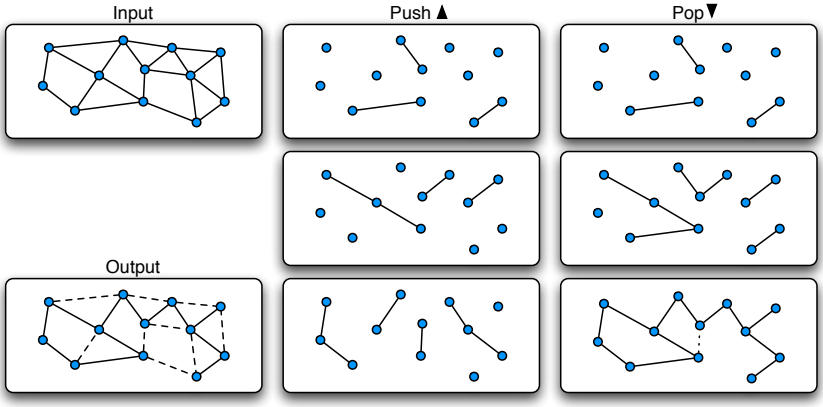

Figure 4.1: Example of a STACKMR run.

of nodes whose capacity constraints are satisfied or violated are deleted from the stack and ignored from further consideration. This phase is iterated until the stack becomes empty.

Figure 4.1 shows an example of a run of the STACKMR algorithm. On the top left we have the input: an undirected weighted graph. In the middle column the push phase builds the stack layer by layer, going upwards. Each layer is a maximal $\lceil \epsilon b \rceil$-matching. On the right column the pop phase consumes the stack in reverse order and merges each layer to create the final solution. Edges in lower layers might get discarded if capacities are saturated. The output of the algorithm is shown on the bottom left, where dashed edges have been excluded by STACKMR.

We can show that the approximation guarantee of our algorithm is $\frac{1}{6+\epsilon}$, for every $\epsilon > 0$. Moreover, we can show that the push phase is iterated at most $O(\log \frac{w_{\max}}{w_{\min}})$ steps, where $w_{\max}$ and $w_{\min}$ are the maximum and minimum weight of any edge in input, respectively. This fact, together with the fact that the procedure by Garrido et al. (1996) requires $O(\log^3 n)$ rounds imply the following theorem.

Theorem 1. *Algorithm 1 has an approximation guarantee of* $\frac{1}{6+\epsilon}$ *and violates capacity constraints by a factor of at most* $1+\epsilon$*. It requires* $O(\frac{\log^3 n}{\epsilon^2} \cdot \log \frac{w_{\max}}{w_{\min}})$

communication rounds, with high probability.

The non-determinism follows from the algorithm that computes maximal b-matchings. The proof of Theorem 1 is similar to the one given by Panconesi and Sozio. STACKMR is a factor $1/\epsilon$ faster than the original.

4.5.3 Adaptation in MapReduce

The distributed algorithm described in the previous section works in an iterative fashion. In each iteration we first compute a *maximal matching*, then we *push* it on a stack, we *update* edges, and we *pop* all levels from the stack. Below we describe how to implement these steps in MapReduce.

Maximal matching. To find maximal b-matchings we employ the algorithm of Garrido et al. which is an iterative probabilistic algorithm. Each iteration consists of four stages: (*i*) `marking`, (*ii*) `selection`, (*iii*) `matching`, and (*iv*) `cleanup`.

In the `marking` stage, each node v *marks* randomly $\lceil \frac{1}{2} b(v) \rceil$ of its incident edges. In the `selection` stage, each node v *selects* randomly $\max\{\lfloor \frac{1}{2} b(v) \rfloor, 1\}$ edges from those marked by its neighbors. Call F this set of selected edges. In the `matching` stage, if some node v has capacity $b(v) = 1$ and two incident edges in F, it randomly deletes one of them. At this point the set F is a valid b-matching. The set F is added to the solution and removed from the original graph. In the `cleanup` stage, each node updates its capacity in order to take into consideration the edges in F and saturated nodes are removed from the graph. These stages are iterated until there are no more edges left in the original graph. The process requires, on expectation, $O(\log^3 n)$ iterations to terminate.

To adapt this algorithm in MapReduce, we need one job for each stage of the algorithm. The input and output of each MapReduce job is always of the same format: a consistent view of the graph represented as adjacency lists. We maintain a node-based representation of the graph because we need to make decisions based on the local neighborhood of each node. Assuming the set of nodes adjacent to v_i is $\{v_j, \ldots, v_k\}$, the input and output of each job is a list of pairs $\langle v_i, [(v_j, T_{ij}), \ldots, (v_k, T_{ik})] \rangle$, where v_i is the key and $[(v_j, T_j), \ldots, (v_k, T_k)]$ the associated value. The

variables T represent the *state* of each edge. We consider five possible states of an edge: E in the main graph; K marked; F selected; D deleted; and M in the matching. The 4 stages of the algorithm are repeated until all the edges are in the matching (M) or are deleted (D).

The general signature of each job is as follows.

$$\begin{aligned}
\textsf{Map}: \quad & \langle v_i, [(v_j, T_{ij}), \ldots, (v_k, T_{ik})] \rangle \rightarrow \\
& \left[\langle v_i, T_{ij}^{(i)} \rangle, \langle v_j, T_{ij}^{(i)} \rangle, \ldots, \langle v_i, T_{ik}^{(i)} \rangle, \langle v_k, T_{ik}^{(i)} \rangle \right] \\
\textsf{Reduce}: \quad & \langle v_i, T_{ij}^{(i)} \rangle, \langle v_i, T_{ij}^{(j)} \rangle, \ldots, \langle v_i, T_{ik}^{(i)} \rangle, \langle v_k, T_{ik}^{(k)} \rangle \rightarrow \\
& \langle v_i, [(v_j, T'_{ij}), \ldots, (v_k, T'_{ik})] \rangle
\end{aligned}$$

where $T_{ij}^{(i)}$ is the state of edge (v_i, v_j) as locally viewed by v_i and T'_{ij} is the final state after unification of the views from v_i and v_j.

Each Map function alters the state of the graph locally to each node. Each Reduce function unifies the diverging views of the graph at each node. For each edge (v_i, v_j), each Map function will emit both v_i and v_j as keys, together with the current state of the edge as value. The Reduce function will receive the views of the state of the edge from both endpoints, and will unify them, yielding a consistent graph representation. Unification is performed by a simple precedence rule. If $a \succ b$ the algorithm unifies two diverging states (a, b) to state b. Only a subset of all the possible combinations of states may actually happen, so we need to specify only a few rules as follows.

$$E \succ K \succ F; \qquad F \succ M; \qquad E \succ D;$$

Each MapReduce job uses the same communication pattern and state unification rules. They only differ in the way they update the state of the edges. The communication cost of each job is thus $O(|E|)$, while the achievable degree of parallelism is $O(|V|)$. Figure 4.2 illustrates the communication pattern, which is the same for each vertex. Let us take into account the central node. For each incident edge, the mapper emits the edge state as viewed by the node twice: once for each end of the edge. In the picture the state view is represented by the arrows which are color coded to represent their origin and point to their destination. For each

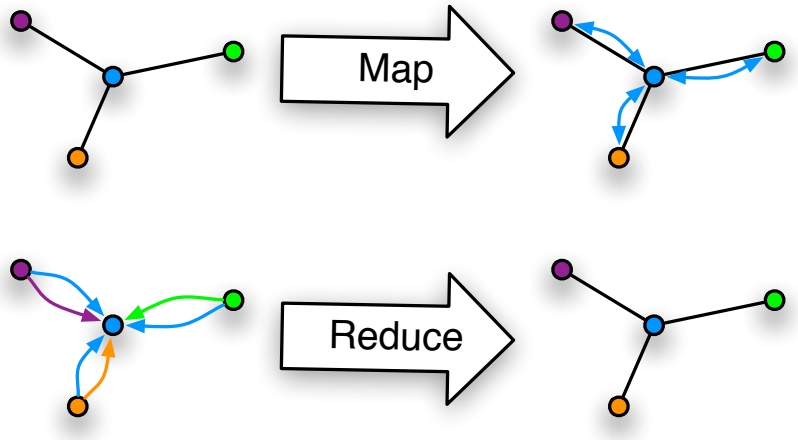

Figure 4.2: Communication pattern for iterative graph algorithms on MR.

incident edge, the reducer receives two different views of the state of the edge, one from each endpoint. The reducer reconciles the two views by applying the unification rules to obtain a consistent view of the graph.

Push, update, and pop. The basic scheme of communication for the *push*, *update*, and *pop* phases is the same as the one for computing maximal matching. We maintain the same invariant of representing the graph from Map to Reduce functions and in-between consecutive iterations. For these phases of the algorithm, we maintain a separate state for each edge. The possible states in which an edge can be are: E, edge in the graph; S, edge stacked; R, edge removed from the graph; and I, edge included in the solution. For each edge we also maintain an integer variable that represents the stack level in which the edge has been put.

During the push phase, for each edge included in the maximal matching, we set its state to S and the corresponding stack level. The update phase is needed to propagate the $\delta(e)$ contributions. Each edge sent to a node v carries the value of its sending node $y_u/b(u)$. Thus, each node can compute the new $\delta(e)$ and update its local y_v. This phase also removes

weakly covered edges by setting their state to R, and updates the capacities of the nodes for the next maximal-matching phase. Removed edges are not considered for the next maximal-matching phase.

The pop phase starts when all the edges in the graph are either stacked (S) or removed (R). During the pop phase, each stacked (S) edge in the current level (starting from the topmost) is included in the solution by setting its state to I. The capacities are locally updated, and nodes (and all incident edges) are removed when their capacity becomes non-positive. Overall, there is one MapReduce step for each stack level.

The decision to change the state of the edge during these phases depends only on the current state of the edge. The only local decision is edge removal when the vertex capacity gets saturated. Therefore, we only need to deal with diverging views related to the removed (R) state:

$$E \succ R; \qquad S \succ R; \qquad I \succ R;$$

4.5.4 The GREEDYMR algorithm

In this section we present a second matching algorithm based on a greedy strategy: GREEDYMR. As previously, we analyze the centralized version and then we show how to adapt it in MapReduce.

The centralized greedy algorithm processes sequentially each edge in order of decreasing weight. It includes an edge $e(u, v)$ in the solution if $b(u) > 0 \wedge b(v) > 0$. In this case, it subtracts 1 from both $b(u)$ and $b(v)$.

It is immediate that the greedy algorithm produces a feasible solution. In addition, it has a factor $\mathbf{1/2}$ approximation guarantee. We believe that this is a well-known result, however, we were not able find a reference. Thus, for completeness we include a proof in Section 4.5.5.

GREEDYMR is a MapReduce adaptation of this centralized algorithm. We note that the adaptation is not straightforward due to the access to the globally-shared variables $b(v)$ that hold node capacities.

GREEDYMR works as follows. In the `map` phase each node v proposes its $b(v)$ edges with maximum weight to its neighbors. In the `reduce` phase, each node computes the intersection between its own proposals and the proposals of its neighbors. The set of edges in the intersection is

included in the solution. Then, each node updates its capacity. If it becomes 0, the node is removed from the graph. Pseudocode for GREEDYMR is shown in Algorithm 3.

In contrast with STACKMR, GREEDYMR is not guaranteed to terminate in a poly-logarithmic number of iterations. As a simple worst-case input instance, consider a path graph $u_1u_2, u_2u_3, ...u_{k-1}u_k$ such that $w(u_i, u_{i+1}) \leq w(u_{i+1}, u_{i+2})$. GREEDYMR will face a chain of cascading updates that will cause a linear number of MapReduce iterations. However, as shown in Section 4.6, in practice GREEDYMR yields quite competitive results compared to STACKMR.

Finally, GREEDYMR maintains a feasible solution at each step. The advantage is that it can be terminated at any time and return the current solution. This property makes GREEDYMR especially attractive in our application scenarios, where content can be delivered to the users immediately while the algorithm continues running in the background.

Algorithm 3 GREEDYMR

while E is non empty **do**
  for all $v \in V$ in parallel **do**
    Let $\widehat{L_v}$ be the set of $b(v)$ edges incident to v with maximum weight;
    Let F be $\widehat{L_v} \cap \widehat{L_U}$ where $U = \{u \in V : \exists\, e(v,u) \in E\}$ is the set f vertexes sharing an edge with v;
    Update $M \leftarrow M \cup F$;
    Update $E \leftarrow E \setminus F$;
    Update $b(v) \leftarrow b(v) - b_F(v)$;
    If $b(v) = 0$ remove v from V and remove all edges incident to v from E;
  end for
end while
return M;

4.5.5 Analysis of the GREEDYMR algorithm

The following theorem proves the approximation guarantee of GREEDYMR. We believe this result to be well-known, however we could not find a reference. Thus, we give a proof for completeness and self containment. The greedy algorithm can be equivalently described as the follow-

ing process: at each step each node v marks an edge with largest weight. If an edge is marked by both its end nodes then it enters the solution and residual node capacities are updated. As soon as the residual capacity of a node becomes zero all its edges are deleted and ignored from further consideration. These steps are iterated until the set of edges of the input graph becomes empty.

Theorem 2. *The greedy algorithm produces a solution with approximation guarantee $\frac{1}{2}$ for the weighted b-matching problem.*

Proof. Let O be an optimum solution for a given problem instance and let A be the solution yielded by the greedy algorithm. For every node v, let O_v and A_v denote the sets of edges $O \cap \Delta_G(v)$ and $A \cap \Delta_G(v)$, respectively, where $\Delta_G(v)$ is the set of edges in G incident to v. The total weight of a set of edges T is denoted by $w(T)$. We say that a node is *saturated* if exactly $b(v)$ edges of v are in the greedy solution A and we let S denote the set of saturated nodes.

For every node v, we consider the sets $\widehat{O}_v \subseteq O_v \setminus A$, defined as follows: each edge $e(u,v) \in O \setminus A$ is assigned to a set $\widehat{O}_v$, for which v is a saturated node and the weight of any edge in A_v is larger than $w(e)$. Ties are broken arbitrarily. There must be such a node v, for otherwise e would be included in A. The idea of the proof is to relate the weight of edge e with the weights of the edges of A_v, which prevent e from entering the solution. From the definition of the $\widehat{O}_v$'s it follows that

$$w(O \setminus A) = \sum_{v \in S} w(\widehat{O}_v). \tag{4.6}$$

For every saturated node v we have that $|O_v| \leq b(v) = |A_v|$. From this and from the definition of the $\widehat{O}_v$'s we have that

$$\sum_{v \in S} w(A_v \setminus O) \geq \sum_{v \in S} w(\widehat{O}_v). \tag{4.7}$$

From Equations (4.6) and (4.7) we obtain

$$2w(A) \geq w(A \cap O) + \sum_{v \in S} w(A_v \setminus O) \geq w(A \cap O) + \sum_{v \in S} w(\widehat{O}_v) \geq w(O),$$

which concludes the proof. □

The analysis is tight as proved by the following example. Consider a cycle consisting of three nodes u, v, z and three edges uv, vz, zu. Let $b(u) = b(z) = 1$ and let $b(v) = 2$. Moreover, let $w(uv) = w(vz) = 1$ while $w(zu) = (1+\epsilon)$ where $\epsilon > 0$. The greedy algorithm would select the edge whose weight is $1 + \epsilon$, while the weight of the optimum solution is 2.

4.6 Experimental evaluation

Flickr. We extract two datasets from flickr, a photo-sharing Web site. Table 4.1 shows statistics for the `flickr-small` and `flickr-large` datasets. In these datasets items represent photos and consumers represent users. In each dataset, each user has posted at least 10 photos, and each photo has been considered at least 5 times as a favorite.

Recall from our discussion in Section 4.4 that the capacity $b(u)$ of each user u should be set in proportion to the login activity of the user in the system. Unfortunately, the login activity is not available in the datasets. So we decide to use the number of photos $n(u)$ that the user u has posted as a *proxy* to his activity. We then use a parameter $\alpha > 0$ to set the capacity of each user u as $b(u) = \alpha\, n(u)$. Higher values of the parameter α simulate higher levels of activity in the system.

Next we need to specify the capacity of photos. Since our primary goal is to study the matching algorithm, specifying the actual capacities is beyond the scope of the work. Thus we use as a *proxy*, the number of favorites $f(p)$ that each photo p has received. The intuition is that we want to favor good photos in order to increase user satisfaction. Following Section 4.4, we set the capacity of each photo to

$$b(p) = f(p) \frac{\sum_u \alpha n(u)}{\sum_q f(q)}.$$

In order to estimate edge similarities, we represent each photo by its tags, and each user by the set of all tags he or she has used. Then we compute the similarity between a photo and a user as the cosine similarity of the their tag vectors. We compute all edges whose similarity is larger than a threshold σ by employing SSJ-2R.

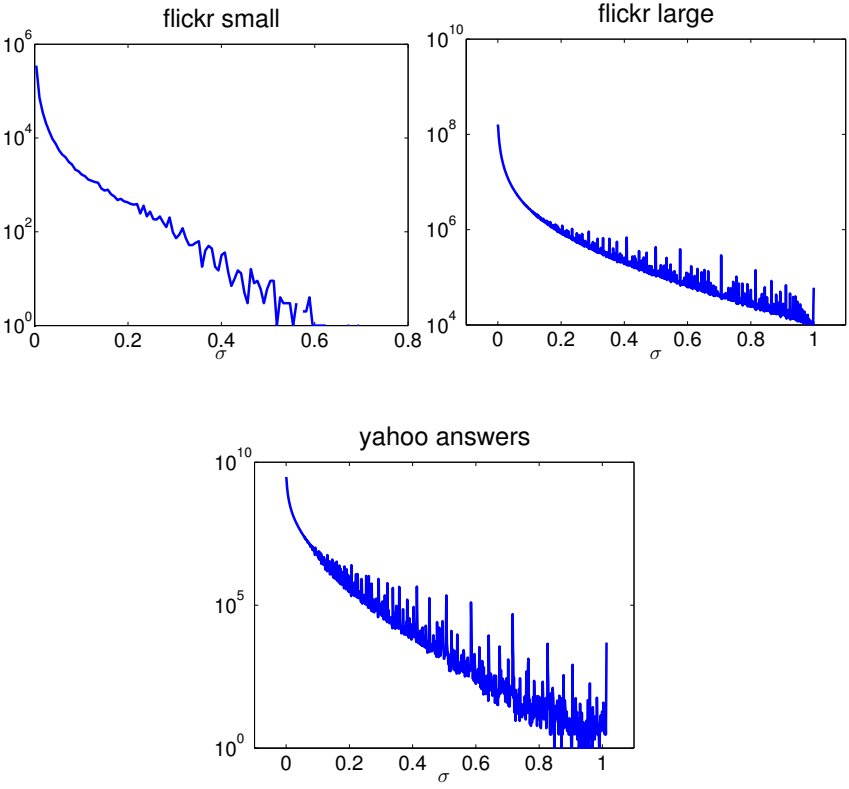

Figure 4.3: Distribution of edge similarities for the datasets.

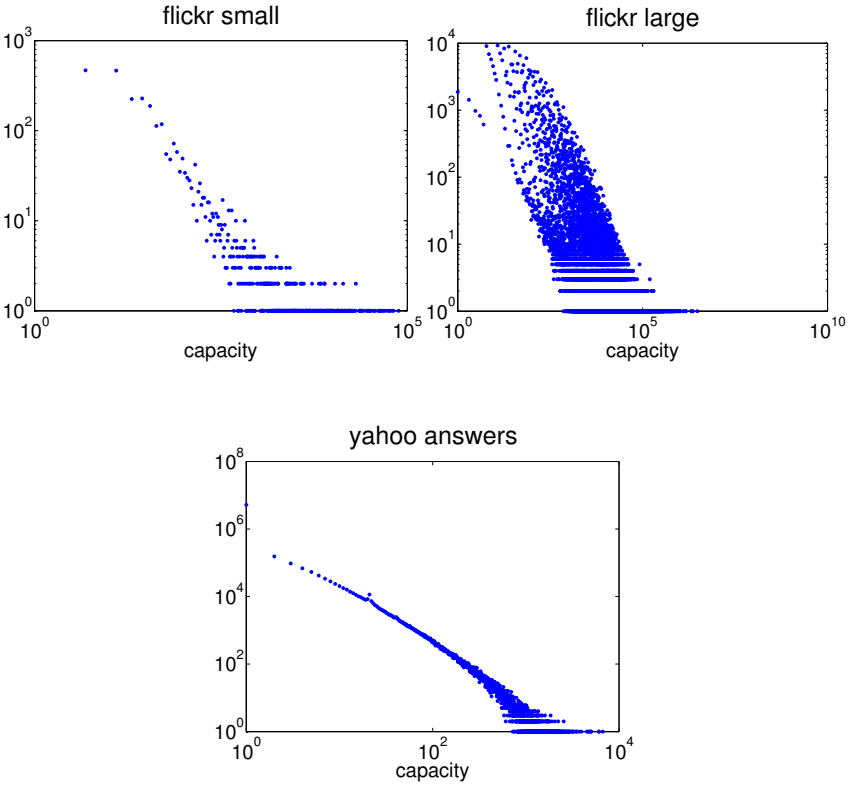

Figure 4.4: Distribution of capacities for the three datasets.

Yahoo! Answers. We extract one dataset from Yahoo! Answers, a Web question-answering portal. In `yahoo-answers`, consumers represent users, while items represent questions. The motivating application is to propose unanswered questions to users. Matched questions should fit the interests of the user. To identify user interests, we represent users by the weighted set of words in their answers. We preprocess the answers to remove punctuation and stop-words, stem words, and apply `tf·idf` weighting. We treat questions in the same manner.

As before, we extract a bipartite graph with edge weights representing the similarity between questions and users. We employ again a threshold σ to sparsify the graph, and present results for different density levels. In this case, we set user capacities $b(u)$ by employing the number of answers $n(u)$ provided by each user u as a proxy to the activity of the user. We use the same parameter α as for the `flickr` datasets to set $b(u) = \alpha\, n(u)$. However, for this dataset we use a constant capacity for all questions, in order to test our algorithm under different settings. For each question q we set

$$b(q) = \frac{\sum_u \alpha n(u)}{|Q|}.$$

The distributions of node capacities are shown in Figure 4.4 while the distributions of edge similarities are shown in Figure 4.3.

Variants. We also experiment with a number of variants of the STACKMR algorithm. In particular, we vary the edge-selection strategy employed in the first phase of the maximal b-matching algorithm (`marking`). The STACKMR algorithm proposes to its neighbors edges chosen uniformly at random. In order to favor heavier edges in the matching, we modify the selection strategy to propose the $\lceil \frac{1}{2}\epsilon b(v) \rceil$ edges with the largest weight. We call this variant STACKGREEDYMR. We also experiment with a third variant, in which we choose edges randomly but with probability proportional to their weights. We choose not to show the results for this third variant because it always performs worse than STACKGREEDYMR

Measures. We evaluate the proposed algorithms in terms of both *quality* and *efficiency*. Quality is measured in terms of b-matching value achieved, and efficiency in terms of the number of MapReduce iterations required.

Table 4.1: Dataset characteristics. $|T|$: number of items; $|C|$: number of users; $|E|$: total number of item-user pairs with non zero similarity.

Dataset	$\|T\|$	$\|C\|$	$\|E\|$
`flickr-small`	2 817	526	550 667
`flickr-large`	373 373	32 707	1 995 123 827
`yahoo-answers`	4 852 689	1 149 714	18 847 281 236

We evaluate our algorithms by varying the following parameters: the similarity threshold σ, which controls the number of edges that participate in the matching; the factor α in determining capacities; and the slackness parameter ϵ used by StackMR.

Results. Sample results on the quality and efficiency of our matching algorithms for the three datasets, `flickr-small`, `flickr-large`, and `yahoo-answers`, are shown in Figures 4.5, 4.6, and 4.7, respectively. For each plot in these figures, we fix the parameters ϵ and α and we vary the similarity threshold σ. Our observations are summarized as follows.

Quality. GreedyMR consistently produces matchings with higher value than the two stack-based algorithms. Since GreedyMR has better approximation guarantee, this result is in accordance with theory. In fact, GreedyMR achieves better results even though the stack algorithms have the advantage of being allowed to exceed node capacities. However, as we will see next, the violations of the stack-based algorithms are very small, ranging from practically 0 to at most 6%. In the `flickr-large` dataset, GreedyMR produces solutions that have on average 31% higher value than the solutions produced by StackMR. In `flickr-small` and `yahoo-answers`, the improvement of GreedyMR is 11% and 14%, respectively. When comparing the two stack algorithms, we see that StackGreedyMR is slightly better than StackMR. Again the difference is more pronounced on the `flickr-large` dataset.

We also observe that in general the b-matching value increases with the number edges. This behavior is expected, as the number of edges increase the algorithms have more flexibility. Since we add edges by lowering the edge weight threshold, the gain in the b-matching value tends to saturate. The only exception to this rule is for StackGreedyMR on the

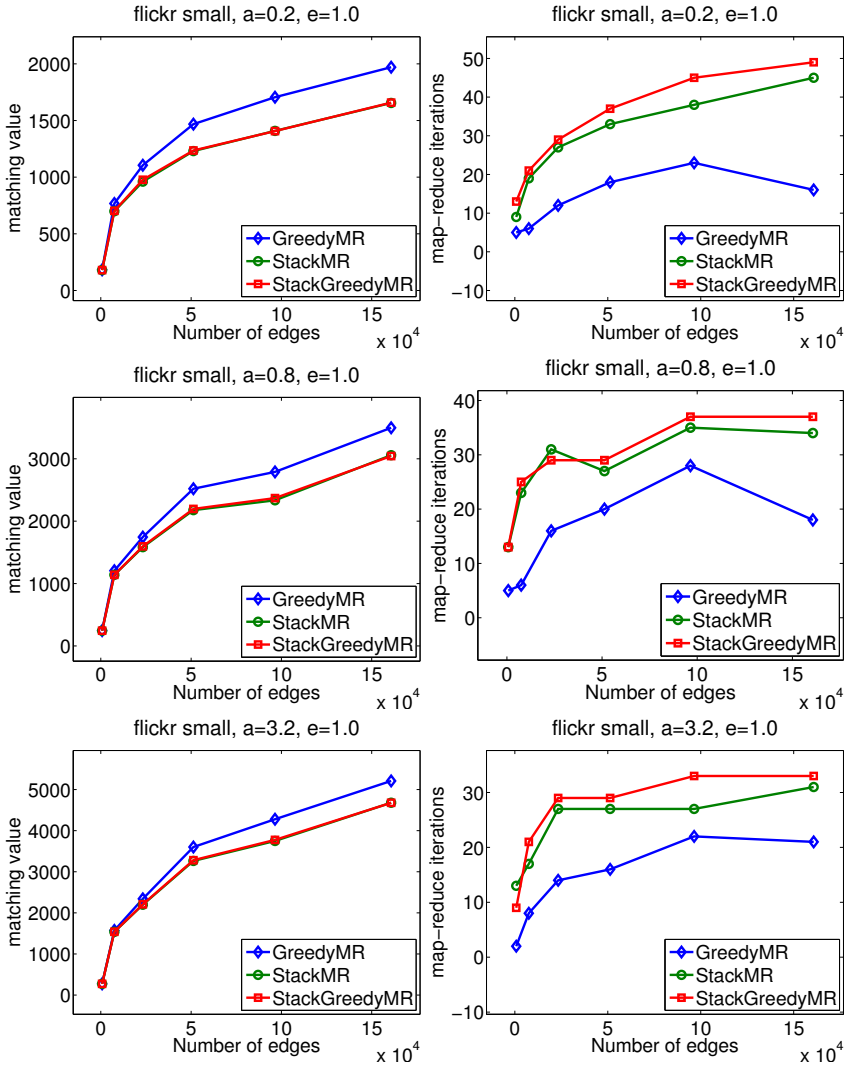

Figure 4.5: `flickr-small` dataset: matching value and number of iterations as a function of the number of edges.

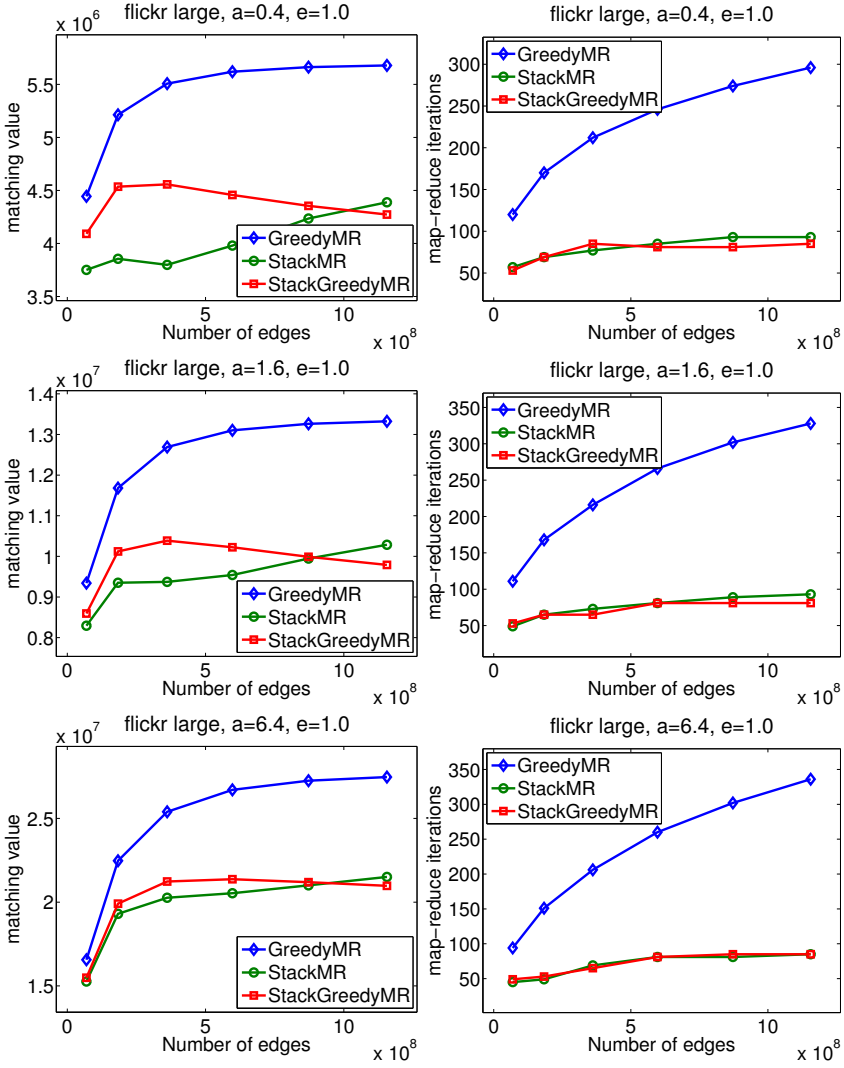

Figure 4.6: `flickr-large` dataset: matching value and number of iterations as a function of the number of edges.

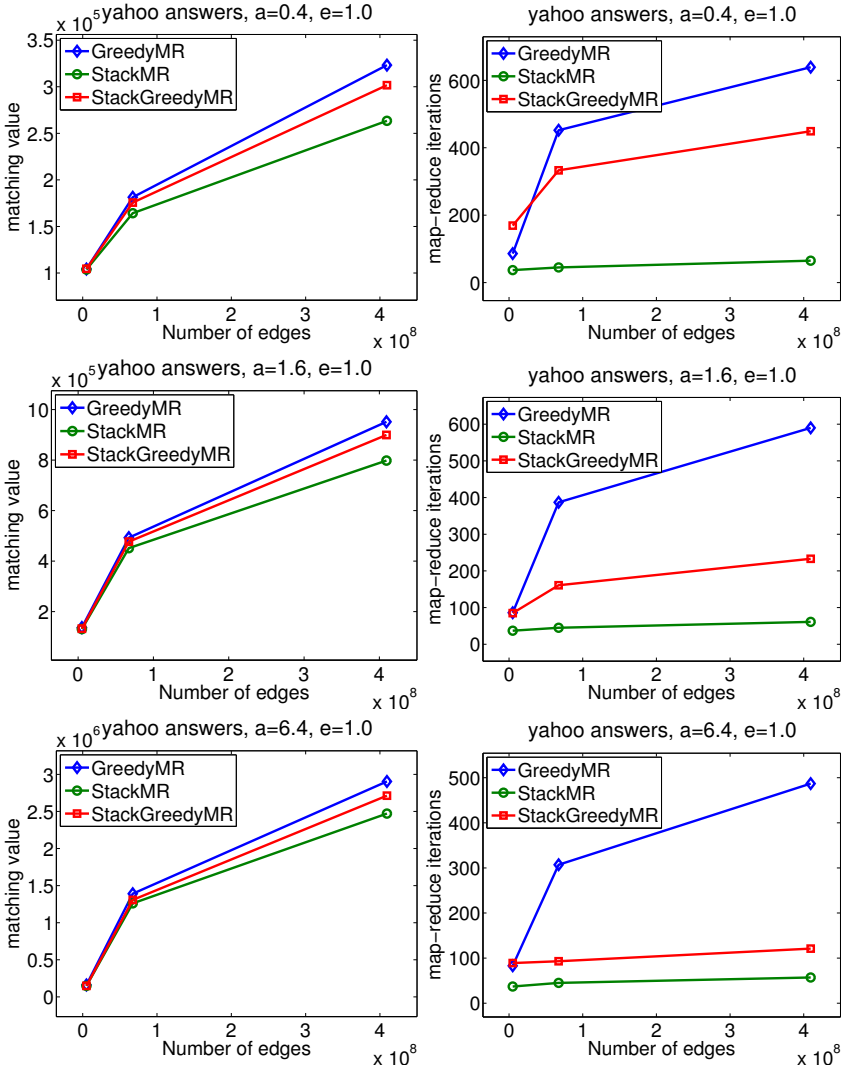

Figure 4.7: `yahoo-answers` dataset: matching value and number of iterations as a function of the number of edges.

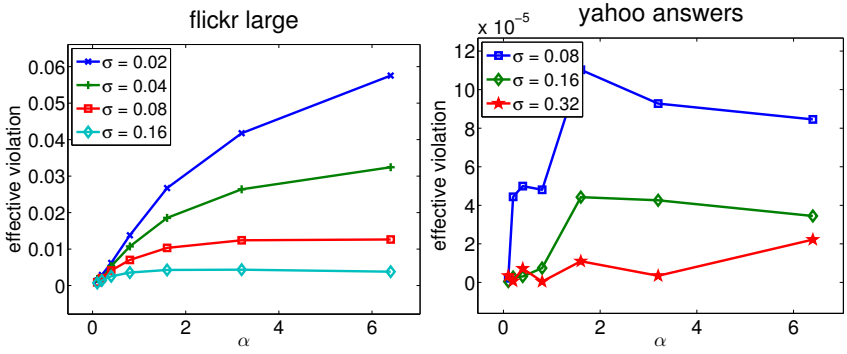

Figure 4.8: Violation of capacities for STACKMR.

`flickr-large` dataset. We believe this is due to the uneven capacity distribution for the `flickr-large` dataset, see Figure 4.4. Our belief is supported by fact that the decrease is less visible for higher values of α.

Efficiency. Our findings validate the theory also in terms of efficiency. In most settings the stack algorithms perform better than GREEDYMR. The only exception is the `flickr-small` dataset. This dataset is very small, so the stack algorithms incur excessive overhead in computing maximal matchings. However, the power of the stack algorithms is best proven on the larger datasets. Not only they need less MapReduce steps than GREEDYMR, but they also scale extremely well. The performance of STACKMR is almost unaffected by increasing the number of edges.

Capacity violations. As explained in Section 4.5.2, the two stack-based algorithms can exceed the capacity of the nodes by a factor of $(1 + \epsilon)$. However, in our experiments the algorithms exhibit much lower violations than the worst case. We compute the average violation as follows.

$$\epsilon' = \frac{1}{|V|} \sum_{v \in V} \frac{\max\{|M(v)| - b(v), 0\}}{b(v)},$$

where $|M(v)|$ is the degree of node v in the matching M, and $b(v)$ is the capacity for each node v in V. Figure 4.8 shows capacity violations for STACKMR. The violations for STACKGREEDYMR are similar. When

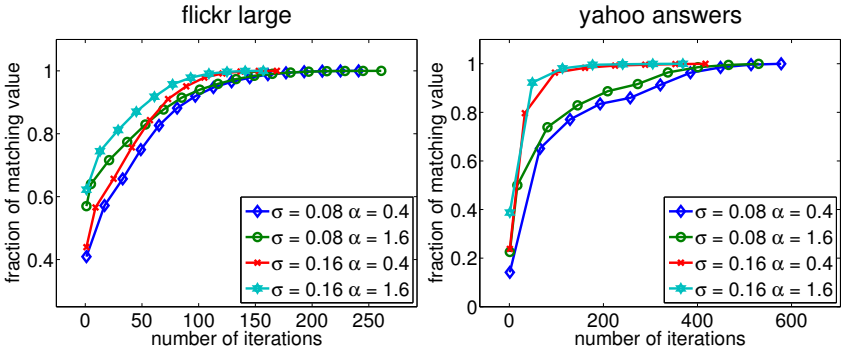

Figure 4.9: Normalized value of the b-matching achieved by the GREEDY-MR algorithm as a function of the number of MapReduce iterations.

$\epsilon = 1$, for the `flickr-large` dataset the violation is as low as 6% in the worst case. As expected, more violations occur when more edges are allowed to participate in the matching, either by increasing the number of edges (by lowering σ) or the capacities of the nodes (by increasing α). On the other hand, for the `yahoo-answers` datasets, using the same $\epsilon = 1$, the violations are practically zero for any combination of the other parameters. One reason for the difference between the violations in these two datasets may be the capacity distributions, as shown in Figure 4.4. For all practical purposes in our scenarios, these violations are negligible.

Any-time stopping. An interesting property of GREEDYMR is that it produces a feasible but suboptimal solution at each iteration. This allows to stop the algorithm at any time, or to query the current solution and let the algorithm continue in the background. Furthermore, GREEDY-MR converges very fast to a good global solution. Figure 4.9 shows the value of the solution found by GREEDYMR as a function of the iteration. For the three datasets, `flickr-small`, `flickr-large`, and `yahoo-answers`, the GREEDYMR algorithm reaches 95% of its final b-matching value within 28.91%, 44.18%, and 29.35% of the total number of iterations required, respectively. The latter three numbers are averages over all the parameter settings we tried in each dataset.

4.7 Conclusions

Graph problems arise continuously in the context of Web mining. In this chapter we investigate the graph b-matching problem and its application to content distribution. Our goal is to help users effectively experience Web 2.0 sites with large collections of user-generated content.

We described two iterative MapReduce algorithms, STACKMR and GREEDYMR. Both algorithms rely on the same base communication pattern. This pattern is able to support a variety of computations and provides a scalable building block for graph mining in MR.

To the best of our knowledge, STACKMR and GREEDYMR are the first solutions to graph b-matching in MR. Both algorithms have provable approximation guarantees and scale to realistic-sized datasets. In addition STACKMR provably requires only a poly-logarithmic number of MapReduce iterations. On the other hand GREEDYMR has a better approximation guarantee and allows to query the solution at any time.

We evaluated our algorithms on two large real-world datasets from the Web and highlighted the tradeoffs between quality and performance between the two solutions. GREEDYMR is a good solution for practitioners. In our experiments it consistently found the best b-matching value. Moreover, it is easy to implement and reason about. Nevertheless, STACKMR has high theoretical and practical interest because of its better running time. It scales gracefully to massive datasets and offers high-quality results.

Chapter 5

Harnessing the Real-Time Web for Personalized News Recommendation

Real-time Web is an umbrella term that encompasses several social microblogging services. It can be modeled as a graph of nodes exchanging streams of updates via publish-subscribe channels. As such it crosses the boundaries between graphs and streams as defined in the taxonomy of Chapter 1. Famous examples include Facebook's news feed and Twitter.

In this chapter we tackle the problem of mining the real-time Web to suggest articles from the news stream. We propose T.REX, a new methodology for generating personalized recommendations of news articles by leveraging the information in users' Twitter persona. We use a mix of signals to model relevance of news articles for users: the content of the tweet stream of the users, the profile of their social circles, and recent topic popularity in news and Twitter streams.

We validate our approach on a real-world dataset from Yahoo! news and one month of tweets that we use to build user profiles. We model the task as click prediction and learn a personalized ranking function from click-through data. Our results show that a mix of various signals from the real-time Web is an effective indicator of user interest in news.

5.1 Introduction

Information overload is a term referring to the difficulty in taking decisions or understanding an issue when there is more information than one can handle. Information overload is not a new problem. The term itself precedes the Internet by several years Toffler (1984). However, digital and automated information processing has aggravated the problem to unprecedented levels, transforming it into one of the crucial issues in the modern information society.

In this work we focus on one of the most common daily activities: reading news articles online. Every day the press produces thousands of news articles covering a wide spectrum of topics. For a user, finding relevant and interesting information in this ocean of news is a daunting task and a perfect example of information overload.

News portals like Yahoo! news and Google news often resort to recommender systems to help the user find relevant pieces of information. In recent years the most successful recommendation paradigm has been *collaborative filtering* (Adomavicius and Tuzhilin, 2005). Collaborative filtering requires the users to rate the items they like. The rating can be explicit (e.g., 'like', '+1', number of stars) or implicit (e.g., click on a link, download). In both cases, collaborative filtering leverages a closed feedback loop: user preferences are inferred by looking only at the user interaction with the system itself. This approach suffers from a data sparsity problem, namely it is hard to make recommendations for users for whom there is little available information or for brand new items, since little or no feedback is available for such "cold" items.

At the same time, at an increasingly rate, web users access news articles via micro-blogging services and the so-called real-time web. By subscribing to feeds of other users, such as friends, colleagues, domain experts and enthusiasts, as well as to organizations of interest, they obtain timely access to relevant and personalized information. Yet, information obtained by micro-blogging services is not complete as highly-relevant events could easily be missed by users if none of their contacts posts about these events.

We propose to recommend news articles to users by combining two sources of information, news streams and micro-blogs, and leveraging the best features of each. News streams have high *coverage* as they aggregate a very large volume of news articles obtained from many different news agencies. On the other hand, information obtained by micro-blogging services can be exploited to address the problems of *information filtering and personalization*, as users can be placed in the context of their social circles and personal interests.

Our approach has the following advantages. First, we are able to leverage the information from social circles in order to offer personalized recommendations and overcome the data-sparsity problem. If we do not have enough information for a user, there should be significantly more information from the social circle of the user, which is presumably relevant. Second, more than once it has been reported that news break out earlier on the real-time web than in traditional media, and we would like to harness this property to provide timely recommendations.

With more than 200 million users, Twitter is currently the most popular real-time web service. Twitter is an emerging agora where users publish short text messages, also known as tweets, and organize themselves into social networks. Interestingly, in many cases, news have been published and commented on Twitter before any other news agency, as in the case of Osama Bin-Laden's death in 2011,[1] or Tiger Wood's car crash in 2009.[2] Due to its popularity, traditional news providers, such as magazines, news agencies, have become Twitter users: they exploit Twitter and its social network to disseminate their contents.

Example. In Figure 5.1 we show the normalized number of tweets and news articles regarding "Osama Bin Laden" during the time period between May 1st and 4th, 2011. For the news, we also report the number of clicks in the same period. We notice that the number of relevant tweets ramps up earlier than news, meaning that the information spread through Twitter even before any press release. Later on, while the number of tweets decreases quickly, and stabilizes around a relatively small

[1] `http://www.bbc.co.uk/news/technology-13257940`

[2] `http://techcrunch.com/2009/11/27/internet-twitter-tiger-woods`

number of tweets, the number of published news continues to be quite large. The number of clicks on news articles follows somewhat the number of published news, even though there is a significant delay until the time the users actually click and read an article on the topic. Figure 5.2 presents similar information for the "Joplin tornado." In this case, even though there is a great activity both on Twitter and on news streams, users start reading the news only after one day. About 60% of the clicks on the news occur starting from 10 hours after its publication. □

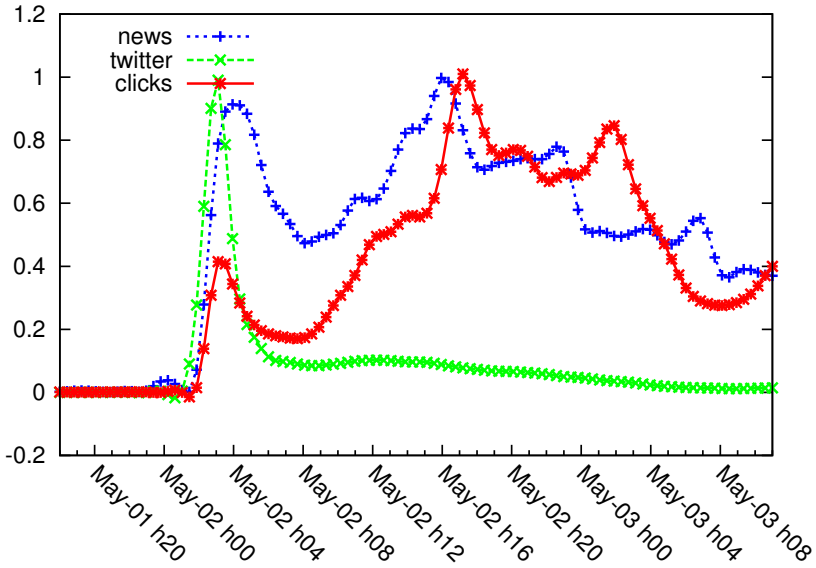

Figure 5.1: *Osama Bin Laden* trends on Twitter and news streams.

The goal of this work is to reduce the delay between the publication of a news and its access by a user. We aim at helping the users in finding relevant news as soon as they are published. Our envisioned application scenario is a feature operating on a news aggregator like Yahoo! news or Google news. Our objective is to develop a recommendation system that provides the users with fresh, relevant news by leveraging their tweet

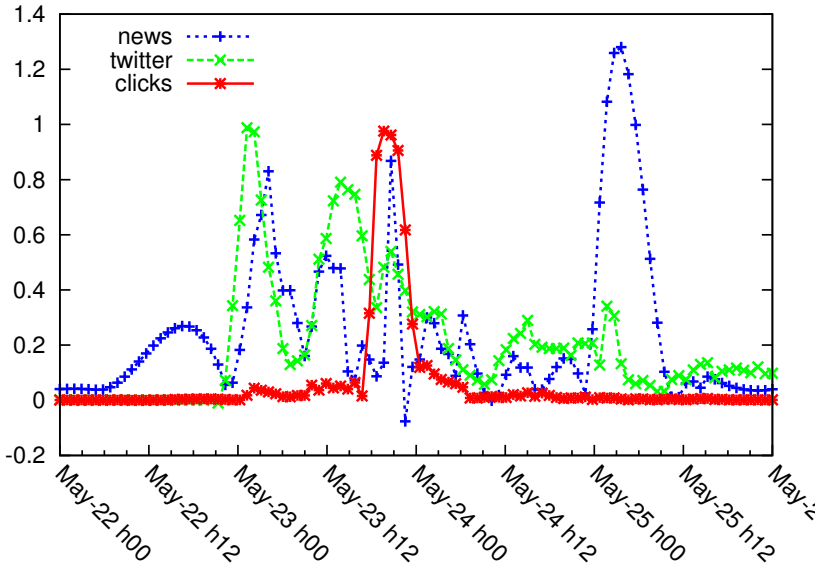

Figure 5.2: *Joplin tornado* trends on Twitter and news streams.

stream. Ideally, the users log into the news portal and link their Twitter account to their portal account in order to provide access to their tweet stream. The portal analyzes the tweet stream of the users and provides them personalized recommendations.

Solving this problem poses a number of research challenges. First, the volume of tweets and news is significantly large. It is necessary to design a scalable recommender system able to handle millions of users, tweets and news. Second, both tweets and news are unbounded streams of items arriving in real-time. The set of news from which to choose recommendations is highly dynamic. In fact, news are published continuously and, more importantly, they are related to events that cannot be predicted in advance. Also, by nature, news have a short life cycle, since they are replaced by updated news on the same topic, or because they become obsolete. Therefore, a recommender system should be able to find

relevant news early, before they lose their value over time. Third, the nature of tweets, short and jargon-like, complicates the task of modeling the interests of users.

Finally, personalization should leverage user profiles to drive the user to less popular news. But it should not prevent from suggesting news of general interest, even when unrelated to the user profile, e.g., the Fukushima accident.

In summary, our contributions are the following.

- We propose an adaptive, online recommendation system. In contrast, typical recommendation systems operate in offline mode.

- We present a new application of usage of tweets. Twitter has mainly been used to identify trending topics and analyzing information spread. Its application to recommendation has received little interest in the community. Our main recommendation system is called T.REX, for *twitter-based news recommendation system*.

- Our system provides personalized recommendations by leveraging information from the tweet stream of users, as well as from the streams of their social circle. By incorporating social information we address the issue of data sparsity. When we do not have enough information for a user, for example for a new user or if a user rarely tweets, the available information in the social circle of the user provides a proxy to his interests.

The rest of this chapter is organized as follows. In Section 5.3 we formalize the news recommendation problem, and introduce our proposed personalized news ranking function. In Section 5.5 we describe the learning process of the ranking function based on click data. In Section 5.6 we present the experimental setting used in the evaluation of our algorithms and our results. Finally, in Section 5.2 we discuss other related work.

5.2 Related work

Recommender systems can be roughly divided in two large categories: content-based and collaborative filtering (Goldberg et al., 1992).

The large-scale system for generating personalized news recommendations proposed by Das et al. (2007) falls in the second category. The authors exploit a linear combination of three different content agnostic approaches based only on click-through data: clustering of users based on minhashing, probabilistic latent semantic indexing of user profiles and news, and news co-visitation count. Even tough the system can update its recommendation immediately after a new click is observed, we have seen that click information arrives with a significant delay, and therefore it may fail in detecting emerging topics early.

Using Twitter to provide fresh recommendations about emerging topics has received a large deal of attention recently. A number of studies attempts to understand the Twitter social network, the information-spreading process, and to discover emerging topics of interesest over such a network (Bakshy et al., 2011; Cataldi et al., 2010; Java et al., 2007).

Chen et al. (2010) propose a URL-recommender system from URLs posted in Twitter. A user study shows that both the user content-based profile, and the user social neighborhood plays a role, but, the most important factor in the recommendation performance is given by the social ranking, i.e., the number of times a URL is mentioned among the neighborhood of a given user. The work presented by Garcia Esparza et al. (2010) exploits data from a micro-blogging movie review service similar to Twitter. The user profile is built on the basis of the user's posts. Similarly, a movie profile generated from the posts associated to the movie. The proposed prototype resembles a content-based recommender system, where users are matched against recommended items.

A small user study by Teevan et al. (2011) reports that 49% of users search while looking for information related to news, or to topics gaining popularity, and in general to "to keep up with events." The analysis of a larger crawl of Twitter shows that about 85% of the Twitter posts are about headlines or persistent news (Kwak et al., 2010). As a result,

this abundance of news-related content makes Twitter the ideal source of information for the news recommendation task.

Akcora et al. (2010) use Twitter to detect abrupt opinion changes. Based on an *emotion-word corpus*, the proposed algorithm detects opinion changes, which can be related to publication of some interesting news. No actual automatic linking to news is produced. The system proposed by Phelan et al. (2011) is a content-based approach that uses tweets to rank news using `tf·idf`. A given a set of tweets, either public or of a friend, is used to build a user profile, which is matched against the news coming from a set of user-defined RSS feeds. McCreadie et al. (2010) show that, even if useful, URL links present in blog posts may not be sufficient to identify interesting news due to their sparsity. Their approach exploits user posts as if they were votes for the news on a given day. The association between news and posts is estimated by using a divergence from randomness model. They show that a gaussian weighting scheme can be used profitably to predict the importance of a news on a given day, given the posts of a few previous days.

Our approach pursues the same direction, with the goal of exploiting Twitter posts to predict news of interest. To this end, rather than analyzing the raw text of a tweet, we chose to extract the entities discussed in tweets. Indeed, Twitter highlights in its Web interfaces the so called *trending topics*, i.e., set of words occurring frequently in recent tweets. Asur et al. (2011) crawled all the tweets containing the keywords identifying a Twitter trending topic in the 20 minutes before the topic is detected by Twitter. The authors find that the popularity of a topic can be described as a multiplicative growth process with noise. Therefore the cumulative number of tweets related to a trending topic increases linearly with time.

Kwak et al. (2010) conduct an interesting study on a large crawl of Twitter, and analyze a number of features and phenomena across the social network. We highlight a few interesting findings. After the initial break-up, the cumulative number of tweets of a trending topic increases linearly, as suggested by Asur et al. (2011), and independently of the number of users. Almost 80% of the trending topics have a single activity

period, i.e., they occur in a single burst, and 93% of such activity periods last less than 10 days. Also, once a tweet is published, half of its re-tweets occur within an hour and 75% within one day. Finally, once re-tweeted, tweets quickly spread four hops away. These studies confirm the fast information spreading occurring on Twitter.

A basic approach for building topic-based user profiles from tweets is proposed by Michelson and Macskassy (2010). Each capitalized non-stopword is considered an entity. The entity is used to query Wikipedia and the categories of the retrieved page are used to update the list of topics of interest for the user who authored the tweet.

Abel et al. (2011) propose a more sophisticated user model to support news recommendation for Twitter users. They explore different ways of modeling use profiles by using hashtags, topics or entities and conclude that entity based modeling gives the best results. They employ a simple recommender algorithm that uses cosine similarity between user profiles and tweets. The authors recommend tweets containing news URLs and re-tweets are used as ground truth.

Based on these results, we choose to use Wikipedia to build an entity based user model. We use the SPECTRUM system (Paranjpe, 2009) to map every single tweet to a bag of entities, each corresponding to a Wikipedia page. We apply the same entity extraction process to the stream of news with the goal of overcoming the vocabulary mismatch problem. Our proposed model borrows from the aforementioned works by exploiting a blend of content-based and social-based profile enrichment. In addition it is able to discover emerging trends on Twitter by measuring entity popularity and taking into account aging effects. Finally, we propose to learn the ranking function directly from available data.

5.3 Problem definition and model

Our goal is to harness the information present in tweets posted by users and by their social circles in order to make relevant and timely recommendation of news articles. We proceed by introducing our notation and define formally the problem that we consider in this work. For quick

Table 5.1: Table of symbols.

Symbol	Definition
$\mathcal{N} = \{n_0, n_1, \ldots\}$	Stream of news
n_i	i-th news article
$\mathcal{T} = \{t_0, t_1, \ldots\}$	Stream of tweets
t_i	i-th tweet
$\mathcal{U} = \{u_0, u_1, \ldots\}$	Set of users
$\mathcal{Z} = \{z_0, z_1, \ldots\}$	Set of entities
$\tau(n_i)$	Timestamp of the i-th news article n_i
$\tau(t_i)$	Timestamp of the i-th tweet t_i
$c(n_i)$	Timestamp of the click on n_i
S	Social network matrix
S^*	Social-influence network matrix
A	User $\times$ tweet authorship matrix
T	Tweet $\times$ entity relatedness matrix
N	Entity $\times$ news relatedness matrix
$M = T \cdot N$	Tweet $\times$ news relatedness matrix
$\Gamma = A \cdot M$	User $\times$ news content relatedness
$\Sigma = S^* \cdot A \cdot M$	User $\times$ news social relatedness
Z	Entity popularity
$\Pi = Z \cdot N$	News popularity
$R_\tau(u, n)$	Ranking score of news n for user u at time τ

reference, our notation is also summarized in Table 5.1.

Definition 2 (News stream). *Let $\mathcal{N} = \{n_0, n_1, \ldots\}$ be an unbounded stream of news arriving from a set of news sources, where news article n_i is published at time $\tau(n_i)$.*

Definition 3 (Tweet stream). *Let $\mathcal{T} = \{t_0, t_1, \ldots\}$ be an unbounded stream of tweets arriving from the set of Twitter users, where tweet t_i is published at time $\tau(t_i)$.*

Problem 3 (News recommendation problem). *Given a stream of news $\mathcal{N}$, a set of users $\mathcal{U} = \{u_0, u_1, \ldots\}$ and their stream of tweets $\mathcal{T}$, find the top-k most relevant news for user $u \in \mathcal{U}$ at time τ.*

We aim at exploiting the tweet and news streams to identify news

of general interest, but also at exploiting a Twitter-based user profile to provide personalized recommendations. That is, for any user $u \in \mathcal{U}$ at any given time τ, the problem requires to rank the stream of past news in $\mathcal{N}$ according to a user-dependent relevance criteria. We also aim at incorporating time recency into our model, so that our recommendations favor the most recently published news articles.

We now proceed to model the factors that affect the relevance of news for a given user. We first model the social-network aspect. In our case, the social component is induced by the Twitter *following* relationship. We define S to be the social network adjacency matrix, were $S(i, j)$ is equal to 1 divided by the number of users followed by user u_i if u_i follows u_j, and 0 otherwise. We also adopt a functional ranking (Baeza-Yates et al., 2006) that spreads the interests of a user among its neighbors recursively. By limiting the maximum hop distance d, we define the social influence in a network as follows.

Definition 4 (Social influence S^*)**.** *Given a set of users $\mathcal{U} = \{u_0, u_1, \ldots\}$, organized in a social network where each user may express an interest to the content published by another user, we define the social influence model S^* as the $|\mathcal{U}| \times |\mathcal{U}|$ matrix where $S^*(i, j)$ measures the interest of user u_i to the content generated by user u_j and it is computed as*

$$S^* = \left(\sum_{i=1}^{i=d} \sigma^i S^i \right),$$

where S is the row-normalized adjacency matrix of the social network, d is the maximum hop-distance up to which users may influence their neighbors, and σ is a damping factor.

Next we model the profile of a user based on the content that the user has generated. We first define a binary authorship matrix A to capture the relationship between users and the tweets they produce.

Definition 5 (Tweet authorship A)**.** *Let A be a $|\mathcal{U}| \times |\mathcal{T}|$ matrix where $A(i, j)$ is 1 if u_i is the author of t_j, and 0 otherwise.*

The matrix A can be extended to deal with different types of relationships between users and posts, e.g., weigh differently *re-tweets*, or *likes*.

In this work, we limit the concept of authorship to the posts actually written by the user.

It is worth noting that a tweet stream for a user is composed of tweets authored by the user and by people in the social neighborhood of the user. This is a generalization of the "home timeline" as known in Twitter.

We observe that news and tweets often happen to deal with the same topic. Sometimes a given topic is pushed into Twitter by a news source, and then it spread throughout the social network. However sometimes a given topic is first discussed by Twitter users, and it is later reflected in the news, which may or may not be published back on Twitter. In both cases, it is important to discover which are the current trending topics in order to promptly recommend news of interest. We model the relationship between tweets and news by introducing an intermediate layer between the two streams. This layer is populated by what we call *entities*.

Definition 6 (Tweets-to-news model M)**.** *Let $\mathcal{N}$ be a stream of news, $\mathcal{T}$ a stream of tweets, and $\mathcal{Z} = \{z_0, z_1, \ldots\}$ a set of entities. We model the relationship between tweets and news as a $|\mathcal{T}| \times |\mathcal{N}|$ matrix M, where $M(i, j)$ is the relatedness of tweet t_i to news n_j, and it is computed as*

$$M \;=\; T \;\cdot\; N,$$

where

T *is a $|\mathcal{T}| \times |\mathcal{Z}|$ row-wise normalized matrix with $T(i, j)$ representing the relatedness of tweet t_i to entity z_j;*

N *is a $|\mathcal{Z}| \times |\mathcal{N}|$ column-wise normalized matrix with $N(i, j)$ representing the relatedness of entity z_i to news n_j.*

The set of entities $\mathcal{Z}$ introduces a middle layer between the stream of news and the stream of tweets that allows us to generalize our analysis. First, this layer allows to overcome any vocabulary mismatch problem between the two streams, since the streams are mapped onto the entity space. Second, rather than monitoring the relevance of a specific news or a tweet, we propose to measure the relevance of an entity.

A number of techniques can be used to extract entities from news and tweets. A naïve approach is to let each term in the dictionary play

the role of an entity. In this case $T(t, z)$ can be estimated as the number of occurrences of the term z in the tweet t, or as a `tf·idf` score, and similarly for N. Alternatively, probabilistic latent semantic indexing can map tweets and news onto a set of latent topics (Hofmann, 1999).

In this work we follow a third approach: we use an existing entity-extraction system. In particular we use the SPECTRUM system, which was proposed by Paranjpe (2009). Given any fragment of text, the SPECTRUM system identifies entities related to Wikipedia articles. Therefore, we assume that $\mathcal{Z}$ consists of the set of all titles of Wikipedia articles. This choice has some interesting advantages. First, once an entity is detected, it is easy to propose a meaningful *label* to the user. Second, it allows to include additional external knowledge into the ranking, such as geographic position, categorization, number of recent edits by Wikipedia users. Although we choose to use `Spectrum`, we note that our model is independent of the specific entity extraction technique employed.

Example. Consider the following tweet by user KimAKelly: "Miss Liberty is closed until further notice." The words "Miss Liberty" are mapped to the entity/Wikipedia page *Statue of Liberty*, which is an interesting topic, due to the just announced renovation. This allows to rank high news regarding the Statue of Liberty, e.g. "Statue of Liberty to Close for One Year after 125th Anniversary" by Fox news. Potentially, since the Wikipedia page is geo-referenced, it is possible to boost the ranking of the news for users living nearby, or interested in the entity *New York*. □

Example. Consider the following tweet by user NASH55GARFIELD: "We don't know how we're gonna pay social security benefits to the elderly & we're spending 27.25 mill $ to renovate the Statue of Liberty!!!". The following entities/Wikipedia pages are extracted: *welfare* (from "social benefits"), *social security* and *Statue of Liberty*. This entity extraction suggests that news regarding the announced Statue of Liberty renovation, or the U.S. planned medicare cuts are of interests to the user. □

The three matrices S^*, A, M can be used to measure the relatedness of the news in $\mathcal{N}$ to the tweets posted by a user or her social neighborhood.

Definition 7 (Content-based relatedness Γ)**.** *Given the tweet authorship matrix A and the tweets-to-news model M for a stream of news $\mathcal{N}$ and a stream of*

tweets $\mathcal{T}$ authored by users $\mathcal{U}$, the relatedness between $\mathcal{U}$ and $\mathcal{N}$ is defined as

$$\Gamma = A \cdot M,$$

where Γ is a $|\mathcal{U}| \times |\mathcal{N}|$ matrix, where $\Gamma(u_i, n_j)$ is the relevance of news n_j for user u_i.

According to the definition of Γ, a news article n_j is relevant for user u_i, if the news article discusses entities that have been discussed in the past by the user in her own tweets.

Definition 8 (Social-based relatedness Σ)**.** *Given the tweet authorship matrix A, the tweets-to-news model M, and the social network S for a stream of news $\mathcal{N}$ and a stream of tweets $\mathcal{T}$ authored by users $\mathcal{U}$, the relatedness between $\mathcal{U}$ and $\mathcal{N}$ is defined as*

$$\Sigma = S^* \cdot A \cdot M,$$

where Σ is a $|\mathcal{U}| \times |\mathcal{N}|$ matrix, where $\Sigma(u_i, n_j)$ is the relevance of news n_j for user u_i.

According to the definition of social-based relatedness Σ, the relevance of a news article is computed by taking into account the tweets authored by neighboring users.

The matrices Γ and Σ measure content-based similarity, with no reference to popularity or freshness of tweets, news articles, and entities. In order to provide timely recommendations and catch up with trending news, we introduce a popularity component, which combines the "hotness" of entities in the news stream and the tweet stream.

Definition 9 (Entity-based news popularity Π)**.** *Given a stream of news $\mathcal{N}$, a set of entities $\mathcal{Z}$, and their relatedness matrix N, the popularity of $\mathcal{N}$ is defined as*

$$\Pi = Z \cdot N,$$

where Z is a row-wise vector of length $|\mathcal{Z}|$ and $Z(i)$ is a measure of the popularity of entity z_i. The resulting Π is a row-wise vector of length $|\mathcal{N}|$, where $\Pi(j)$ measures the popularity of the news article n_j.

The vector Z holds the popularity of each entity z_i. The counts of the popularity vector Z need to be updated as new entities of interest arise in the news stream $\mathcal{N}$ and the tweet stream $\mathcal{T}$. An important aspect

in updating the popularity counts is to take into account recency: new entities of interest should dominate the popularity counts of older entities. In this work, we choose to update the popularity counts using an exponential decay rule. We discuss the details in Section 5.3.1. However, note that the popularity update is independent of our recommendation model, and any other decaying function can be used.

Finally, we propose a ranking function for recommending news articles to users. The ranking function is linear combination of the scoring components described above. We plan to investigate the effect of non-linear combinations in the future.

Definition 10 (Recommendation ranking $R_\tau(u, n)$)**.** *Given the components Σ_τ, Γ_τ and Π_τ, resulting form a stream of news $\mathcal{N}$ and a stream of tweets $\mathcal{T}$ authored by users $\mathcal{U}$ up to time τ, the recommendation score of a news article $n \in \mathcal{N}$ for a user $u \in \mathcal{U}$ at time τ is defined as*

$$R_\tau(u, n) \;=\; \alpha \cdot \Sigma_\tau(u, n) \;+ \beta \cdot \Gamma_\tau(u, n) \;+ \gamma \cdot \Pi_\tau(n),$$

where α, β, γ are coefficients that specify the relative weight of the components.

At any given time, the recommender system produces a set of news recommendation by ranking a set of candidate news, e.g., the most recent ones, according to the ranking function R. To motivate the proposed ranking function we note similarities with popular recommendation techniques. When $\beta = \gamma = 0$, the ranking function R resembles collaborative filtering, where user similarity is computed on the basis of their social circles. When $\alpha = \gamma = 0$, the function R implements a content-based recommender system, where a user is profiled by the bag-of-entities occurring in the tweets of the user. Finally, when $\alpha = \beta = 0$, the most popular items recommended, regardless of the user profile.

Note that Σ, Γ, Π and R are all time dependent. At any given time τ the social network and the set of authored tweets vary, thus affecting Σ and Γ. More importantly, some entities may abruptly become popular, hence of interest to many user. This dependency is captured by Π. While the changes in Σ and Γ derive directly from the tweet stream $\mathcal{T}$ and the social network S, the update of Π is non-trivial, and plays a fundamental role in the recommendation system that we describe in the next section.

5.3.1 Entity popularity

We complete the description of our model by discussing the update of the entity popularity counts. We motivate our approach by empirically observing how the user interest for particular entities decays over time.

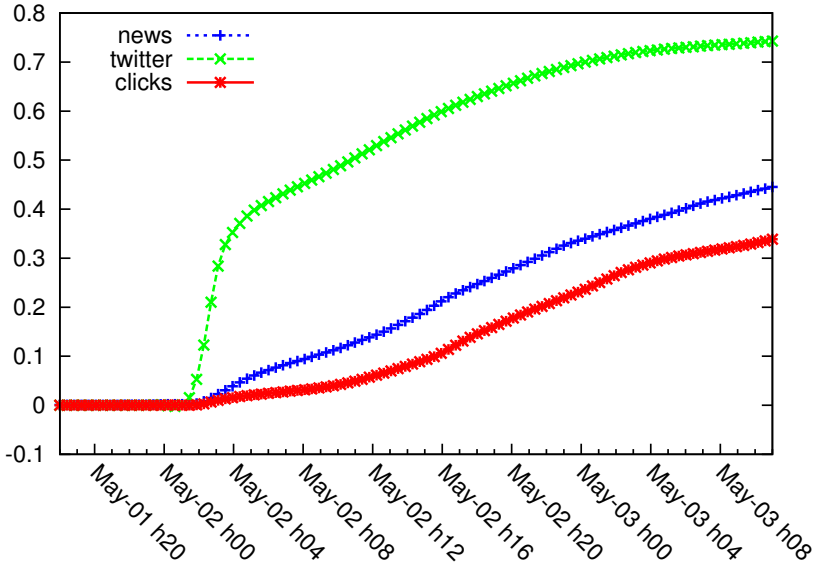

Figure 5.3: Cumulative *Osama Bin Laden* trends (news, Twitter and clicks).

Figure 5.3 shows the cumulative distribution of occurrences for the same entity of Figure 5.1. By using cumulative distribution, the fact the entity appears much earlier in Twitter than in the news becomes more evident. If we consider the number of clicks on news as a surrogate of user interest, we notice that with a delay of about one day the entity receives a great deal of attention. This delay is probably due to the fact that the users have not been informed about the event yet. On the other hand, we observe that after two days the number of clicks drop, possibly the interest of users has been saturated or diverted to other events.

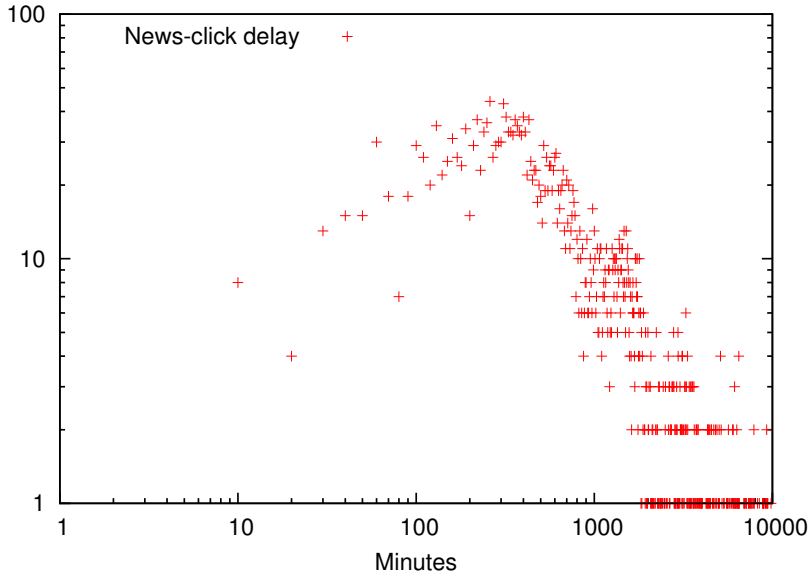

Figure 5.4: News-click delay distribution.

We note that the example above is not an exception, rather it describes the typical behavior of users with respect to reading news articles online. Figure 5.4 shows the scatter plot of the distribution of delay between the time that news articles are published the when they are clicked by users. The figure considers all the news articles and all the clicks in our dataset as described in Section 5.6.1.

Only a very small number of news is clicked within the first hour from their publication. Nonetheless, 76.7% of the clicks happen within one day (1440 minutes) and 90.1% within two days (2880 minutes). Analogous observations can be made by looking at Figure 5.5, which shows the cumulative distribution of the delay and resembles a typical Pareto distribution. The slope of the curve is very steep up to about 2000 minutes (33 hours), however it flattens out quite rapidly soon after.

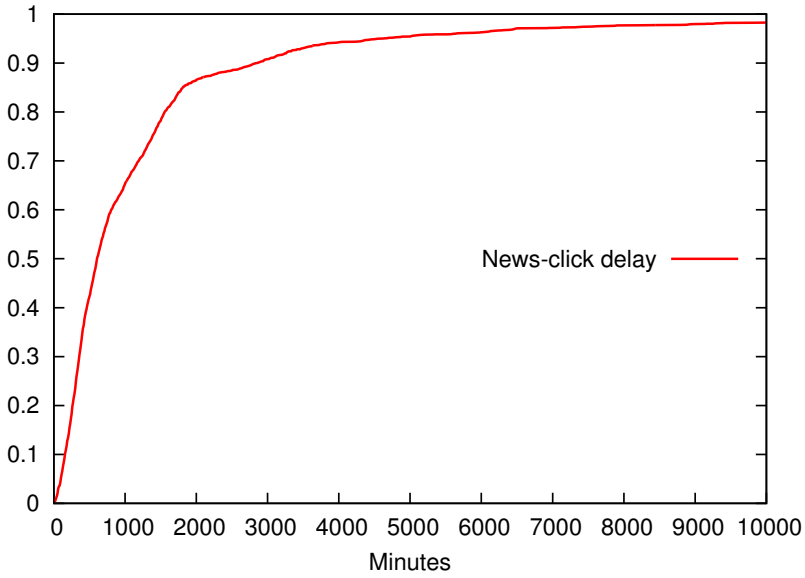

Figure 5.5: Cumulative news-click delay distribution.

From these empirical observations on the delay between news publishing and user clicking we can draw the following conclusions:

1. The increase in the number of tweets and news related to a given entity can be used to predict the increase of interest of the users.

2. News become stale after two days.

The first observation motivates us to inject a popularity component in our recommendation model. The second observation suggests updating popularity scores using a decaying function. We choose an exponentially-decaying function, which has the advantage of allowing to count efficiently frequent items in the data stream model. This choice allows to monitor entity popularity in high-speed data streams by using only a limited amount of memory (Cormode et al., 2008).

Definition 11 (Popularity-update rule)**.** *Given a stream of news $\mathcal{N}$ and a stream of tweets $\mathcal{T}$ at time τ, the popularity vector Z is computed at the end of every time window of fixed width ω as follows*

$$Z_\tau = \lambda Z_{\tau-1} + w_\mathcal{T} H_\mathcal{T} + w_\mathcal{N} H_\mathcal{N}.$$

The vectors $H_\mathcal{T}$ and $H_\mathcal{N}$ are estimates of the expected number of mentions of the entity occurring in tweets and news, respectively, during the latest time window. They are also called hotness *coefficients. The weights $w_\mathcal{T}$ and $w_\mathcal{N}$ measure the relative impact of news and tweets to the popularity count, and $\lambda < 1$ is an exponential forgetting factor.*

The popularity-update rule has the effect of promptly detecting entities becoming suddenly popular, and spreading this popularity in the subsequent time windows. According to our experimental evidence, we fix λ so that a signal is becomes negligible after two days. We set both coefficients $w_\mathcal{T}$ and $w_\mathcal{N}$ to 0.5 to give equal weight to tweets and news.

Asur et al. (2011) show that the number of tweets for a trending topic grows linearly over time. So we compute the expected counts of the entity in tweets and news $H_\mathcal{T}$ and $H_\mathcal{N}$ by using the first order derivative of their cumulative counts, measured at the last time window $\tau - 1$.

5.4 System overview

Figure 5.6 shows a general overview of the T.REX system. T.REX maintains a user model for each user that is composed by the content relatedness and social relatedness components. Conceptually both components process tweets at high speed to extract and count entities. This operation can be done efficiently and in parallel at high speed. Efficiently keeping frequency moments in a stream is a classical problem in the literature of streaming algorithms (Alon et al., 1999).

The social component also holds truncated PageRank values for the followees of each users. These values are used to weight the importance of the followee profile when computing recommendation scores for the user. Truncated PageRank values can be updated periodically, or when the social graph has been modified more than a given threshold.

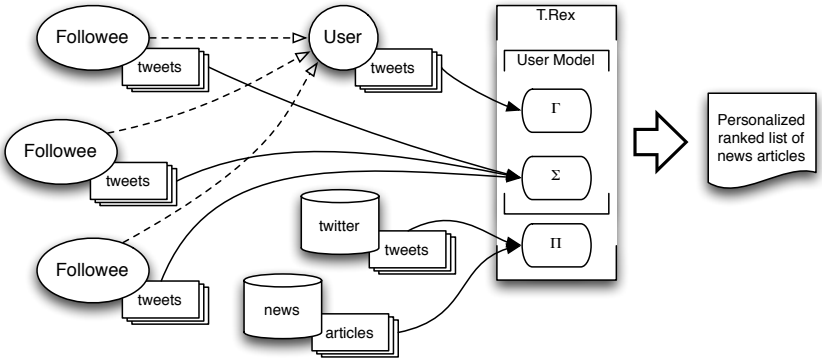

Figure 5.6: Overview of the T.REX system.

PageRank is a typical batch operation and can be easily implemented in MR. Given that we use a truncated version of PageRank, this computation is even cheaper because the values can spread only d hops away, therefore convergence will be faster. Parameters for the ranking function can be computed offline and updated likewise in a batch fashion. The users can possibly be clustered into groups and different parameters can be learned for each group. We leave the study of the effects of clustering for future work. Learning the parameters and the potential clustering are best seen as offline processes, given that the general preferences of the user are not supposed to change too rapidly.

However processing high speed streams is a poor fit for MR. High speed counting is necessary both for the user model and for entity popularity. Even though MR has been extended to deal with incremental and continuous computations (Condie et al., 2009), there are more natural matches for computing on streams. One reason is that the fault tolerance guarantees of MR are too strong in this case. We would rather loose some tweets than wait for them to be reprocessed upon a crash, as long as the system is able to keep running. Freshness of data is a key property to preserve when mining streams, even at the cost of some accuracy.

Therefore we propose to use the actors model to parallelize our sys-

tem (Agha, 1986). In particular, we describe how to implement it on S4.

We use two different type of PEs to update the model: one for the content component and one for entity popularity. The social component can be implemented by aggregating the values from various content components with pre-computed weights, so it does not need a separate PE. The weights can be updated in batches and computed in MR.

A content PE receives tweets keyed on the author, extracts entities from the tweets and updates the counts for the entities seen so far. Then it sends messages keyed on the entity to popularity PEs in order to update their entity popularity counts.

The entity extraction operation is performed also on incoming news by using a keyless PE. This PE just performs entity extraction on the incoming news articles and forwards the mention counts downstream. The news articles with the extracted entities are kept in a pool as long as they do not become stale, after which they are discarded.

A popularity PE receives entity count updates keyed on the entity. It updates its own counts by using the timestamp of the mention and an efficient procedure for exponentially decayed counters Cormode et al. (2008). As popularity is updated at fixed intervals, caching and batching can be used to reduce the network load of the system. For instance updates can be sent only at time window boundaries.

Figure 5.7 depicts an overview of the ranking process at query time. When the user logs into the system, the pool of news articles is ranked according to the user model model and the entity popularity information. The search for relevant news can be speeded up by employing an inverted index of the entities in each news, and indeed the process is very similar to answering a query in a Web search engine.

All the standard techniques for query processing found in the information retrieval literature can be used. For example, it is possible to employ early exit optimizations (Cambazoglu et al., 2010). Ranking of news can proceed in parallel at different PEs, and finally only the top-k news articles from each PE need to be aggregated again at a single place.

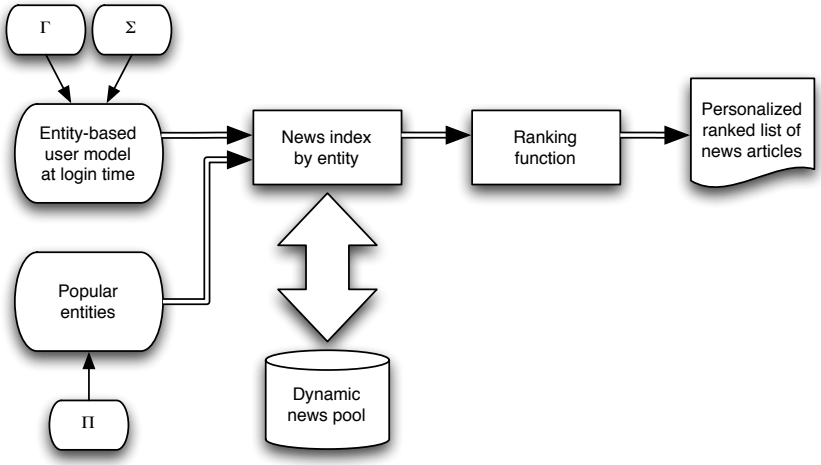

Figure 5.7: T.REX news ranking dataflow.

5.5 Learning algorithm

The next step is to estimate the parameters of the relevance model that we developed in the previous section. The parameters consist of the coefficients α, β, γ used to adjust the relative weight of the components of the ranking function R_τ. Another parameter is the damping factor used in the computation of social influence, but we empirically observed that it does not influence the results very much, so we used a default value of $\sigma = 0.85$ common in PageRank. We consider only direct neighbors by setting the maximum hop distance $d = 1$. This choice is for practical reasons, since crawling Twitter is rate limited and thus very slow.

Our approach is to learn the parameters of the model by using training data obtained from action logs. In particular, we use click-log data from Yahoo! toolbar as a source of user feedback, The Yahoo! toolbar anonymously collects clicks of several millions of users on the Web, including clicks on news. Our working hypothesis is that a high-quality news recommender ranks high the news that will be clicked by users. We

actually formulate our recommendation task according to the learning-to-rank framework (Joachims, 2002).

Consider a user $u \in \mathcal{U}$ and a news article $n \in \mathcal{N}$ with publication timestamp $\tau(n) \leq \tau$, where τ is the current time. We say that the news article n should be ranked in the i-th position of the recommendation list for user u, if it will be the i-th article to be clicked in the future by user u.

This formulation allows to define precedence constraints on the ranking function R_τ. Let $c(n)$ be the time at which the news n is clicked, and let $c(n) = +\infty$ if n is never clicked. At time τ, for two news n_i, n_j with $\tau(n_i) \leq \tau$ and $\tau(n_j) \leq \tau$, we obtain the following constraint:

$$\text{If } \tau \leq c(n_i) < c(n_j) \text{ then } R_\tau(u, n_i) > R_\tau(u, n_j). \tag{5.1}$$

The click stream from Yahoo! toolbar identifies a large number of constraints according to Equation (5.1) that the optimal ranking function must satisfy. As for the learning-to-rank problem Joachims (2002), finding the optimal ranking function is an **NP**-hard problem. Additionally, considering that some of the constraints could be contradictory, a feasible solution may not exist. As usual, the learning problem is translated to a Ranking-SVM optimization problem.

Problem 4 (Recommendation Ranking Optimization)**.**

$$\begin{aligned}
&\textit{Minimize: } V\left(\overrightarrow{\omega} \equiv \langle \alpha, \beta, \gamma \rangle, \overrightarrow{\xi}\right) = \frac{1}{2}\|\overrightarrow{\omega}\|^2 + C\sum \xi_{ij\tau} \\
&\textit{Subject to: } R_\tau(u, n_i) > R_\tau(u, n_j) + 1 - \xi_{ij\tau} \\
&\qquad \textit{for all } \tau, n_i \in \mathcal{N}, n_j \in \mathcal{N} \textit{ such that} \\
&\qquad\quad \tau(n_i) \leq \tau, \tau(n_j) \leq \tau, \tau \leq c(n_i) < c(n_j) \\
&\qquad \xi_{ij\tau} \geq 0
\end{aligned}$$

As shown by Joachims (2002), this optimization problem can be solved via classification SVM. In the following sections, we show how to generate the training set of the SVM classifier in order to keep a reasonably low amount of training instances, so as to speed-up convergence and support the scalability of the solution.

5.5.1 Constraint selection

The formulation of Problem 4 includes a potentially huge number of constraints. Every click occurring after time τ on a news n, generates a new constraints involving n and every other non-clicked news published before time τ. Such a large number of constraints also includes relationships on "stale" news articles, e.g., a non-clicked news article published weeks or months before τ. Clicks are the signals driving the learning process. So it is important to select pairs of clicked news articles that eliminate biases such as the one caused by stale news articles. Clearly, the more constraints are taken into consideration during the learning process, the more robust is the final model. On the other hand, increasing the number of constraints affects the complexity of the minimization algorithm. We propose the following strategy in order to select only the most interesting constraints and thus simplify the optimization problem.

First, we evaluate the ranking function only at the time instants when a click happens. This selection does not actually change the set of constraints of Problem 4, but it helps in the generation of the constraints by focussing on specific time instants. If the user u clicks at the news article n_i at time $c(n_i)$, then the news article n_i must be the most relevant at that time. The following condition must hold.

$$R_{c(n_i)}(u, n_i) > R_{c(n_i)}(u, n_j) + 1 - \xi_{ijc(n_i)}$$
$$\text{for all } n_j \in \mathcal{N} \text{ such that } \tau(n_j) \leq c(n_i). \tag{5.2}$$

Whenever a click occurs at time $c(n_i)$, we add a set of constraint to the ranking function such that the clicked news article gets the largest score among any other news article published before time $c(\tau)$.

Second, we restrict the number of news articles to be compared with the clicked one. If a news article was published a long time before the click, then it can be easily filtered out, as it would not help the learning process. As we have shown in Section 5.3.1, users lose interest into news articles after a time interval $\widehat{\tau}$ of two days. We can make use of this

threshold into Equation (5.2) as follows:

$$R_{c(n_i)}(u, n_i) > R_{c(n_i)}(u, n_j) + 1 - \xi_{ijc(n_i)} \quad \text{for all } n_j \in \mathcal{N}\left[c(n_i) - \widehat{\tau}, c(n_i)\right], \tag{5.3}$$

where $\mathcal{N}\left[c(n_i) - \widehat{\tau}, c(n_i)\right]$ is the set of news articles published between time $c(n_i) - \widehat{\tau}$ and $c(n_i)$.

Finally, we further restrict $\mathcal{N}\left[c(n_i) - \widehat{\tau}, c(n_i)\right]$ by considering only the articles which that are relevant according to at least one of the three score components Σ_τ, Γ_τ, Π_τ. Let $\mathsf{Top}(k, \chi, \tau_a, \tau_b)$ be the set of k news articles with largest rank in the set $\mathcal{N}\left[\tau_a, \tau_b\right]$ according to the score component χ. We include into the optimization Problem 4 only the constraints:

$$\begin{aligned} R_{c(n_i)}(u, n_i) > R_{c(n_i)}(u, n_j) + 1 - \xi_{ijc(n_i)} \\ \text{for all } n_j \text{ s.t. } n_j \in \mathsf{Top}\left(k, \Sigma_{c(n_i)}, c(n_i) - \widehat{\tau}, c(n_i)\right) \text{ or} \\ n_j \in \mathsf{Top}\left(k, \Gamma_{c(n_i)}, c(n_i) - \widehat{\tau}, c(n_i)\right) \text{ or} \\ n_j \in \mathsf{Top}\left(k, \Pi_{c(n_i)}, c(n_i) - \widehat{\tau}, c(n_i)\right). \end{aligned} \tag{5.4}$$

By setting $k = 10$ we are able to reduce the number of constraints from more than 25 million to approximately 250 thousand, thus significantly reducing the training time of the learning algorithm.

5.5.2 Additional features

The system obtained by learning the model parameters using the SVM-Rank method is named T.REX, and forms our main recommender. Additionally, we attempt to improve the accuracy of T.REX by incorporating more features. To build this improved recommender we use exactly the same learning framework: we collect more features from the same training dataset and we learn their importance using the SVM-Rank algorithm. We choose three additional features: *age*, *hotness* and *click count*, which we describe below.

The *age* of a news article n is the time elapsed between the current time and the time n was published, that is, $\tau - \tau(n)$.

The *hotness* of a news article is a set of features extracted from the vectors $H_\mathcal{T}$ and $H_\mathcal{N}$, which keep the popularity counts for the entities in

our model. For each news article n we compute the average and standard deviation of the vectors $H_{\mathcal{T}}$ and $H_{\mathcal{N}}$ over all entities extracted from article n. This process gives us four hotness features per news article.

Finally, the *click count* of a news article is simply the number of times that the article has been clicked by any users in the system up to time τ.

The system that we obtain by training our ranking function on these additional features is called T.REX+.

5.6 Experimental evaluation

5.6.1 Datasets

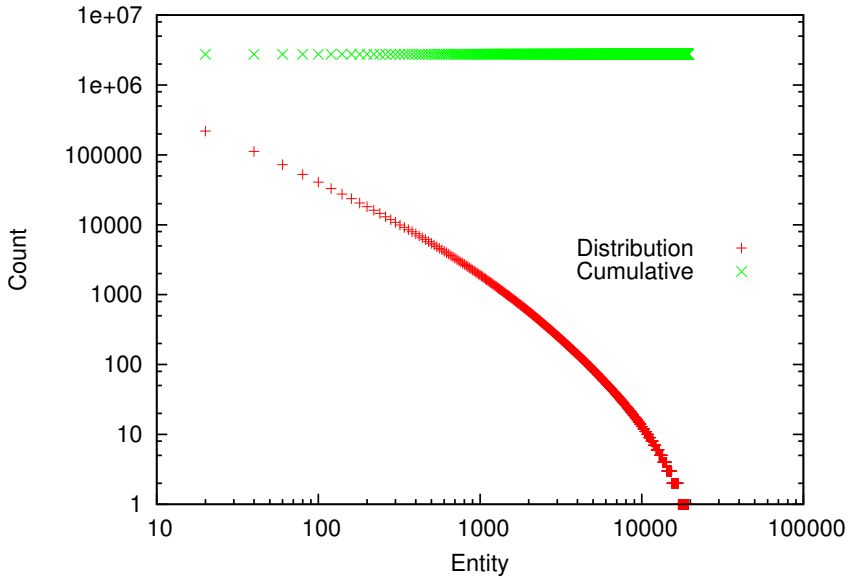

Figure 5.8: Distribution of entities in Twitter.

To build our recommendation system we need the following sources of information: Twitter stream, news stream, the social network of users,

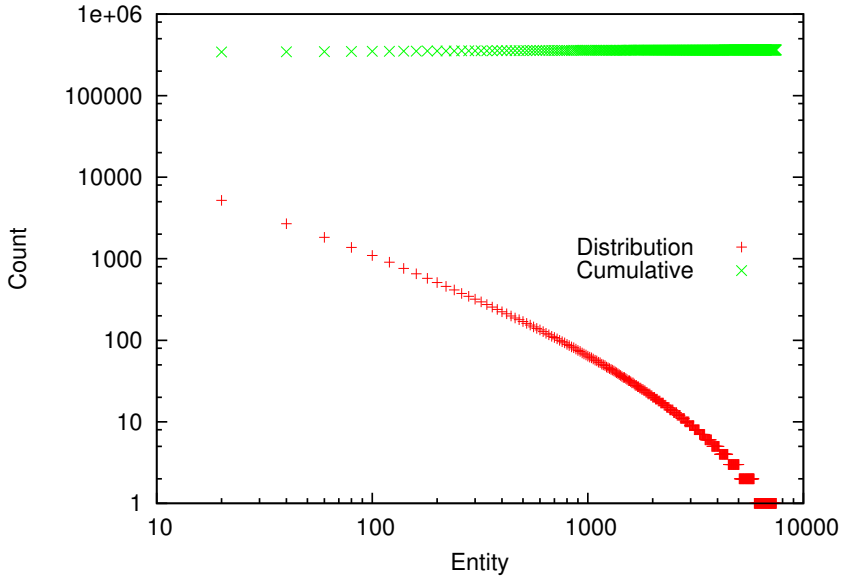

Figure 5.9: Distribution of entities in news.

and click-through data. We extract this information from three different data sources: Twitter, Yahoo! news, and Yahoo! toolbar, respectively.

Twitter: We obtain tweets from Twitter's public API by crawling users' timelines. We collect all tweets posted during May 2011 by our $3,214$ target users (identified as described below). We also collect a random sample of tweets to track entity popularity across all Twitter. We extract entities from tweets using the `Spectrum` method described by Paranjpe (2009). Overall we obtain about 1 million tweets in English, for which we are able to extract entities. Figure 5.8 shows the distribution of entities in the Twitter dataset. The curve has a truncated power law shape.

Yahoo! news: We collect all news articles aggregated in the English site of Yahoo! news during May 2011. From the news articles we extract entities by using again the `Spectrum` algorithm. Overall we have about 28.5 million news articles, from which we keep only the articles that contain

at least one entity contained in one of the tweets. In total we obtain about 40 thousand news articles. Figure 5.9 shows the distribution of entities in the Yahoo! news dataset. The shape of the distribution is similar to the one of Twitter, a truncated power law distribution.

Yahoo! toolbar: We collect click information from the Yahoo! toolbar logs. The Yahoo! toolbar is a browser add-on that provides a set of browsing functionalities. The logs contain information about the browsing behavior of the users who have installed the toolbar, such as, user cookie id, url, referral url, event type, and so on. We collect all toolbar data occurring in May 2011. We use the first 80% in chronological order of the toolbar clicks to train our system.

Using a simple heuristic we identify a small set of users for whom we can link their toolbar cookie id, with their Twitter user id. The heuristic is to identify which Twitter account a user is visiting more often, discarding celebrity accounts and non-bijective mappings. The underlying assumption is that users visit more often their own accounts. In total we identify of set U_0 of 3,214 test users. The dataset used in this work is the projection of all the data sources on the users of the set U_0, that is, for all the users in U_0 we collect all their tweets and all their clicks on Yahoo! news. Additionally, by employing snowball sampling on Twitter we obtain the set of users U_1 who are followed by the set of users U_0. Then, by collecting all the tweets of the users in U_1, we form the social component for our set of users of interest U_0.

5.6.2 Test set

Evaluating a recommendation strategy is a complex task. The ideal evaluation method is to deploy a live system and gather click-through statistics. While such a deployment gives the best accuracy, it is also very expensive and not always feasible.

User studies are a commonly used alternative evaluation method. However, getting judgements from human experts does not scale well to a large number of recommendations. Furthermore, these judgement are often biased because of the small sample sizes. Finally, user studies

cannot be automated, and they are impractical if more than one strategy or many parameters need to be tested.

For these reasons, we propose an automated method to evaluate our recommendation algorithm. The proposed evaluation method exploits the available click data collected by the Yahoo! toolbar, and it is similar in spirit with the learning process. Given a stream of news and a stream of tweets for the user, we identify an event that a user clicks at a news article n_*. Assume that such a click occurs at time $\tau = c(n_*)$. Suppose the user just logged in the system at time τ. Then the recommendation strategy should rank the news article n_* as high as possible.

To create a test instance, we collect all news articles that have been published within the time interval $[\tau - \tau^*, \tau]$, and select a subset as described by Equation (5.4): we pick the news with largest scores according to the components of content, social, popularity, and number of clicks until time τ. In total we generate a pool of $k = 1{,}000$ candidate news. All these news articles are ranked using our ranking function R_τ, and we then examine what is the ranking of the article n_* in the list.

We use the last 20%, in chronological order, of the Yahoo! toolbar log to test the T.REX and T.REX+ algorithms, as well as all the baselines.

5.6.3 Evaluation measures

Evaluating a ranking of items is a standard problem in information retrieval. In our setting we have only one correct answer per ranking — the news article n_* clicked by the user — and we care about the position of this item in the ranking, which we evaluate using the Mean Reciprocal Rank (MRR) measure Voorhees (1999):

$$\mathrm{MRR} = \frac{1}{|Q|} \sum_{i=1}^{Q} \frac{1}{r(n_*^i)},$$

where $r(n_*^i)$ is the rank of the news article n_*^i in the i-th event of the evaluation and Q is the set of all tests. The MRR measure is commonly used to evaluate question-answering systems, where only the first correct answer matters. MRR gives higher scores to correct results that are higher

in the ranking and reduces the importance of correct predictions in the lower parts of the ranking.

We also compute the precision of our system at different levels. For our task we define the *precision-at-k* (P@k) as the fraction of rankings in which the clicked news is ranked in the top-k positions. We compute P@k for $k = 1$, 5, and 10.

There are cases in which the sparsity of the data prevents our system from providing recommendations. Therefore, we also evaluate the coverage of our recommendations. We define the coverage as the fraction of clicks for which we were able to provide some recommendation.

Finally, we note that when the user clicks on a news, the user is actually expressing interest in the entities mentioned in the news. We can argue that if a user clicks on a report on Fukushima from CNN, the user might equally be interested in a report on the same topic from BBC. In this case, we would like to predict the entities that the user is interested in, rather than the news articles themselves.

We use Jaccard overlap between the sets of entities mentioned in two news articles to measure the similarity between the articles themselves. We compute the similarity between each news article in the ranking and the clicked news. Then, we assign a relevance level to each news by binning the computed similarity into five levels. This gives rise to a ranking task with multiple levels of relevance. We use Discounted Cumulated Gain (DCG) (Järvelin and Kekäläinen, 2002) to measure the performance of the strategies on this newly defined task:

$$\mathrm{DCG}\left[i\right] = \begin{cases} G\left[i\right] & \text{if } i = 1; \\ \mathrm{DCG}\left[i-1\right] + \frac{G[i]}{log_2 i} & \text{if } i > 1, \end{cases}$$

where $G[i]$ is the relevance of the document at position i in our ranking. We compute the DCG for the top 20 results averaged over all the rankings produced. If a strategy is not able to produce a ranking, we pad the ranking with non-relevant documents.

Table 5.2: MRR, precision and coverage.

Algorithm	MRR	P@1	P@5	P@10	Coverage
RECENCY	0.020	0.002	0.018	0.036	1.000
CLICKCOUNT	0.059	0.024	0.086	0.135	1.000
SOCIAL	0.017	0.002	0.018	0.036	0.606
CONTENT	0.107	0.029	0.171	0.286	0.158
POPULARITY	0.008	0.003	0.005	0.012	1.000
T.REX	0.107	0.073	0.130	0.168	1.000
T.REX+	0.109	0.062	0.146	0.189	1.000

5.6.4 Baselines

We compare our system with other five ranking baselines:

RECENCY: it ranks news articles by time of publication (most recent first);

CLICKCOUNT: it ranks news articles by click count (highest count first);

SOCIAL: it ranks news articles by using T.REX with $\beta = \gamma = 0$;

CONTENT: it ranks news articles by using T.REX with $\alpha = \gamma = 0$;

POPULARITY: it ranks news articles by using T.REX with $\alpha = \beta = 0$.

5.6.5 Results

We report MRR, precision and coverage results in Table 5.2. The two variants of our system, T.REX and T.REX+, have the best results overall.

T.REX+ has the highest MRR of all the alternatives. This result means that our model has a good overall performance across the dataset. CONTENT has also a very high MRR. Unfortunately, the coverage achieved by the CONTENT strategy is very low. This issue is mainly caused by the sparsity of the user profiles. It is well know that most of Twitter users belong to the "silent majority," and do not tweet very much.

The SOCIAL strategy is affected by the same problem, albeit to a much lesser extent. The reason for this difference is that SOCIAL draws from a large social neighborhood of user profiles, instead of just one. So it

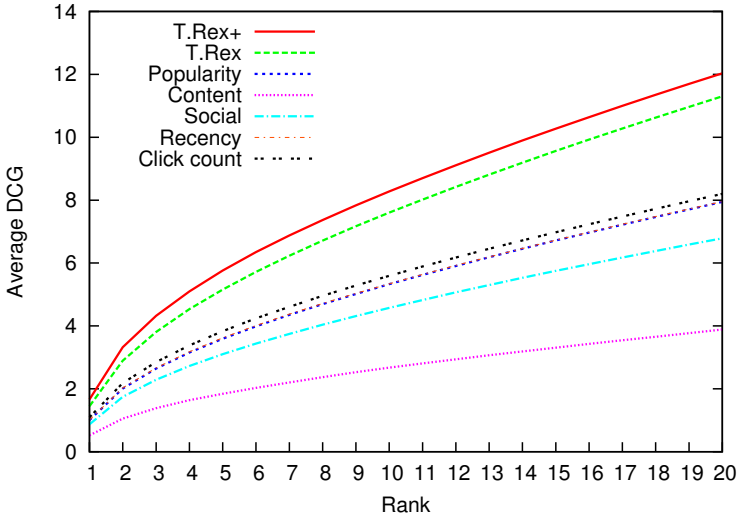

Figure 5.10: Average discounted cumulated gain on related entities.

has more chances to provide a recommendation. The quality of the recommendation is however quite low, probably because the social-based profile only is not able to catch the specific user interests.

It is worth noting that in almost 20% of the cases T.REX+ was able to rank the clicked news in the top 10 results. Ranking by the CLICKCOUNT strategy is quite efficient despite being very simple. Interestingly, adding the CLICKCOUNT feature to our model does not improve the results significantly. In fact, the difference in P@10 between T.REX and T.REX+ is only about 2%. This suggests that the information contained in the click counts is already captured by our model, or cannot be effectively exploited by a linear combination.

Figure 5.10 shows the DCG measure for the top 20 positions in the rankings. From the figure we can see that, compared to the other strategies, T.REX and T.REX+ are able to suggest more news articles about the same entities mentioned in the clicked news. This result is expected, as our system is specifically designed to exploit the concept of entity.

5.7 Conclusions

The real-time Web can provide timely information on important events, and mining it online with stream processing allows to yield maximum benefit from it. In this chapter we proposed an application to online news recommendation. Our goal is to help the user keep up with the torrent of news by providing personalized and timely suggestions.

We described T.REX, an online recommender that combines stream and graph mining to harness the information available on Twitter. T.REX creates personalized entity-based user models from the information in the tweet streams coming from the user and his social circle, and further tracks entity popularity in Twitter and news streams. T.REX constantly updates the models in order to continuously provide fresh suggestions.

We framed our problem as a click-prediction task and we tested our system on real-world data from Twitter and Yahoo! news. We combined the signals from the two streams by using a learning-to-rank approach with training data extracted from Yahoo! toolbar logs. T.REX is able to predict with good accuracy the news articles clicked by the users and rank them higher on average. Our results show that the real-time Web is a useful resource to build powerful predictors of user interest.

Chapter 6

Conclusions

"In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox."

Admiral Grace Hopper,
developer of the first compiler.

Data Intensive Scalable Computing is an emerging technology in the data analysis field that can be used to capitalize on massive datasets coming from the Web. In particular MapReduce and streaming are the two most promising emerging paradigms for analyzing, processing and making sense of large heterogeneous datasets. While MapReduce offers the capability to analyze large batches of stored data, streaming solutions offer the ability to continuously process unbounded streams of data online.

"It is not who has the best algorithms that wins, it is who has more data" is a well known aphorism. The importance of big data is widely recognized both in academia and in industry, and the Web is the largest public data repository available. Information extracted from the Web has the potential to have a big impact in our society.

Data on the Web is heterogeneous and is best modeled in different ways. To fully exploit the power of parallelism offered by DISC systems, algorithms need to take into account the characteristics of their input data. Therefore, designing algorithms for large scale Web mining is an interesting and challenging research goal.

In this thesis we have explored how to tackle the big data problem for the Web by employing Data Intensive Scalable Computing. This problem has two sides. On the one hand we need effective methodologies to make sense of the complex relationships that underly data on the Web. On the other hand we need efficient techniques that enable processing of large datasets. This thesis sheds some light at the crossing of these two areas of research, and investigates the synergies and difficulties in performing large scale Web mining using Data Intensive Scalable Computing. Our work builds on previous research in Data Mining, Information Retrieval and Machine Learning to make sense of the complex data available on the Web. We also employ the products of Distributed Systems and Database research to manage the Web-scale datasets. We have tackled three classical problems in Web mining and provided solutions for MapReduce and streaming.

In Chapter 3 we addressed the issue of similarity on bags of Web pages. Finding similar items in a bag of documents is a challenging problem that arises in many applications in the areas of Web mining. The size of Web-related problems mandates the use of parallel approaches in order to achieve reasonable computing times. We presented SSJ-2 and SSJ-2R, two algorithms specifically designed for the MapReduce programming paradigm. We showed that designing efficient algorithms for MR is not trivial and requires careful blending of several factors to effectively capitalize on the available parallelism. We analyzed and compared our algorithms with the state-of-the-art theoretically and experimentally on a sample of the Web. SSJ-2R borrows from previous approaches based on inverted index, and embeds pruning strategies that have been used only in sequential algorithms so far. Designing algorithms specifically for MapReduce by rethinking classical algorithms gives substantial performance advantages over black box parallelization. By applying this principle SSJ-2R achieves scalability without sacrificing precision by exploiting each MR phase to perform useful work. SSJ-2R outperforms the state-of-the-art for similarity join in MapReduce by almost five times.

In Chapter 4 we explored graph b-matching and proposed its application to the distribution of user-generated content from Web 2.0 sites.

Our goal is to help users experience websites with large collections of user-generated content. We described StackMR and GreedyMR, two iterative MapReduce algorithms with different performance and quality properties. These algorithms represent the first solution to the graph b-matching problem in MapReduce. They rely on the same base communication pattern for iterative graph mining in MR which can support a variety of algorithms and provides a scalable building block for graph mining in MR. Both algorithms have provable approximation guarantees and scale to massive datasets. In addition StackMR provably requires only a poly-logarithmic number of MapReduce iterations. On the other hand GreedyMR has a better approximation guarantee and allows to query the solution at any time. We evaluated our algorithms on two large real-world datasets from the Web and highlighted the tradeoffs between quality and performance between the two solutions.

In Chapter 5 we investigated online news recommendation for real-time Web users. Our goal is to help the user keep up with the torrent of news by providing personalized and timely suggestions. We devised T.Rex, an online recommender that combines stream and graph mining to harness the information available on Twitter. The proposed system processes all the incoming data online in a streaming fashion and is amenable to parallelization on stream processing engines like S4. This feature allows to provide always fresh recommendations and to cope with the large amount of input data. T.Rex extracts several signals from the real-time Web to predict user interest. It is able to harnesses information extracted from user-generated content, social circles and entity popularity in Twitter and news streams. Our results show that the real-time Web is a useful resource to build powerful predictors of user interest.

The research results presented in this thesis open the way to some interesting questions that deserve further investigation. It is reasonable to ask whether all of the useful part of the Web can be modeled as bags, graphs, streams or a mix of these. Furthermore, the Web is highly dynamic and changes rapidly. Therefore new kinds of data might require new models and new algorithms. On the other hand new DISC systems are emerging and they might provide a good fit for some of the algo-

rithms presented here, especially for graph and iterative computations. Finally, the algorithms presented in Chapter 3 and Chapter 4 can be generalized to contexts different from the Web. Finding the limits of applicability of these algorithms is an interesting line of research.

Some direct extensions of the present work are also worth of notice. For instance computing top-k and incremental versions of the similarity self-join problem, or exploring the effects of clustering users and employing entity categories on the news recommendation problem. Finally, live deployments of the systems described in Chapter 4 and 5 could provide useful insight on the quality of the suggestions made. This would in turn open new spaces for further research on user engagement.

We believe that the topic investigated in this thesis has the potential for great practical impact in our society. We hope that the contributions presented here will be valuable for the future researchers in the field.

Appendix A

List of Acronyms

API	Application Programming Interface
CERN	European Council for Nuclear Research
GPS	Global Positioning System
HOP	Hadoop Online Prototype
LHC	Large Hadron Collider
NAS	Network Attached Storage
P2P	Peer-to-Peer
RFID	Radio-Frequency Identification
SAN	Storage Area Network
SMP	Symmetric Multi-Processor
BSP	Bulk Synchronous Parallel
DAG	Direct Acyclic Graph
DBMS	Database Management System
DHT	Distributed Hash Table

DIKW	Data Information Knowledge Wisdom
DISC	Data Intensive Scalable Computing
GFS	Google File System
HDFS	Hadoop Distributed File System
MPI	Message Passing Interface
MR	MapReduce
OLAP	Online Analysis Processing
OLTP	Online Transaction Processing
PDBMS	Parallel Database Management System
PE	Processing Element
PVM	Parallel Virtual Machine
SQL	Structured Query Language
UDF	User Defined Function

References

Fabian Abel, Qi Gao, G.J. Houben, and Ke Tao. Analyzing User Modeling on Twitter for Personalized News Recommendations. In *UMAP '11: 19th International Conference on User Modeling, Adaption and Personalization*, pages 1–12. Springer, 2011. 106

G Adomavicius and A Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, pages 734–749, 2005. 99

Foto N Afrati and Jeffrey D. Ullman. A New Computation Model for Cluster Computing. 2009. URL `http://ilpubs.stanford.edu:8090/953/`. 27, 53

Foto N. Afrati and Jeffrey D. Ullman. Optimizing Joins in a Map-Reduce Environment. In *EDBT '10: 13th International Conference on Extending Database Technology*, pages 99—-110, 2010. 34

Foto N. Afrati and Jeffrey D. Ullman. Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE Transactions on Knowledge and Data Engineering*, 23 (9):1282 – 1298, 2011. 34

Foto N. Afrati, Anish Das Sarma, David Menestrina, Aditya Parameswaran, and Jeffrey D. Ullman. Fuzzy Joins Using MapReduce. Technical report, Stanford InfoLab, July 2011a. URL `http://ilpubs.stanford.edu:8090/1006`. 34

Foto N. Afrati, Dimitris Fotakis, and Jeffrey D. Ullman. Enumerating Subgraph Instances Using Map-Reduce. Technical report, Stanford University, 2011b. URL `http://ilpubs.stanford.edu:8090/1020/`. 35

Charu C. Aggarwal. *Data Streams: Models and Algorithms*. Springer, 2007. ISBN 0387287590. 35

Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, December 1986. ISBN 0-262-01092-5. 23, 32, 118

Eugene Agichtein, Carlos Castillo, Debora Donato, Aristides Gionis, and Gilad Mishne. Finding High-Quality Content in Social Media. In ACM, editor, *WSDM '08: 1st International Conference on Web Search and Data Mining*, pages 183–194, 2008. 74

C G Akcora, M A Bayir, M Demirbas, and H Ferhatosmanoglu. Identifying breakpoints in public opinion. In *SOMA '10: 1st Workshop on Social Media Analytics*, pages 62–66. ACM, 2010. 105

Noga Alon, Yossi Matias, and Mario Szegedy. The Space Complexity of Approximating the Frequency Moments. *Journal of Computer and System Sciences*, 58 (1):137–147, 1999. 30, 116

Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67: April Spring Joint Computer Conference*, pages 483–485. ACM, April 1967. 12

C Anderson. The Petabyte Age: Because more isn't just more—more is different. *Wired*, 16(07), July 2008. 2

Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. OPTICS: Ordering points to identify the clustering structure. In *SIGMOD '99: 25th ACM International Conference on Management of Data*, SIGMOD '99, pages 49–60, New York, NY, USA, 1999. ACM. ISBN 1-58113-084-8. 38

Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *VLDB '06: 32nd International Conference on Very Large Data Bases*, pages 918–929. VLDB Endowment, 2006. 40, 42

S Asur, B A Huberman, G Szabo, and C Wang. Trends in Social Media: Persistence and Decay. *Arxiv*, 2011. 105, 116

Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS '02: 21st Symposium on Principles of Database Systems*, pages 1–30, New York, New York, USA, June 2002. ACM Press. ISBN 1581135076. 30

R Baeza-Yates, P Boldi, and C Castillo. Generalizing PageRank: Damping Functions for Link-Based Ranking Algorithms. *SIGIR '06: 29th International Conference on Research and Development in Information Retrieval*, pages 308—-315, August 2006. 108

Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval: The Concepts and Technology behind Search (2nd Edition)*. Addison-Wesley, 2011. ISBN 0321416910. 6

Eytan Bakshy, Jake M Hofman, Duncan J Watts, and Winter A Mason. EveryoneâĂŹs an Influencer : Quantifying Influence on Twitter. In *WSDM '11: 4th International Conference on Web Search and Data Mining*, WSDM '11, pages 65–74, 2011. 104

N Bansal, A Blum, and S Chawla. Correlation Clustering. *Machine Learning Journal*, 56(1):89–113, 2004. 38

Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling Up All Pairs Similarity Search. In *WWW '07: 16th International Conference on World Wide Web*, pages 131–140, New York, New York, USA, 2007. ACM. ISBN 9781595936547. 41

Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A Comparison of Join Algorithms for Log Processing in MapReduce. In *SIGMOD '10: 36th International Conference on Management of Data*, pages 975–986. ACM Press, 2010. ISBN 9781450300322. 34, 40

George E P Box and Norman R Draper. *Empirical model-building and response surface*, page 424. John Wiley & Sons, Inc., 1986. 2

M. Brocheler, A. Pugliese, and VS Subrahmanian. COSI: Cloud Oriented Subgraph Identification in Massive Social Networks. In *ASONAM '10: International Conference on Advances in Social Networks Analysis and Mining*, pages 248—-255. IEEE, 2010. 35

A Z Broder, S C Glassman, M S Manasse, and G Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8-13), 1997. 41

Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI '06: 7th Symposium on Operating Systems Design and Implementation*, pages 335–350, November 2006. 21

B. Barla Cambazoglu, Hugo Zaragoza, Olivier Chapelle, Jiang Chen, Ciya Liao, Zhaohui Zheng, and Jon Degenhardt. Early exit optimizations for additive machine learned ranking systems. In *WSDM '10: 3rd International Conference on Web Search and Data Mining*, page 411. ACM Press, February 2010. ISBN 9781605588896. 118

Mario Cataldi, Università Torino, Luigi Di Caro, Claudio Schifanella, and Luigi Di Caro. Emerging Topic Detection on Twitter based on Temporal and Social Terms Evaluation. In *MDMKDD '10: 10th International Workshop on Multimedia Data Mining*, pages 4:1—-4:10, 2010. 104

R Chaiken, B Jenkins, P ÅLarson, B Ramsey, D Shakib, S Weaver, and J Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. *VLDB Endowment*, 1(2):1265–1276, August 2008. 22, 23

Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *PLDI '10: SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 363–375. ACM, May 2010. ISBN 978-1-4503-0019-3. 24

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. BigTable: A distributed storage system for structured data. In *OSDI '06: 7th Symposium on Operating Systems Design and Implementation*, pages 205–218, November 2006. 22

Denis Charles, Max Chickering, Nikhil R Devanur, Kamal Jain, and Manan Sanghi. Fast algorithms for finding matchings in lopsided bipartite graphs with applications to display ads. In *EC '10: 11th Conference on Electronic Commerce*, pages 121–128. ACM, 2010. 71

Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A Primitive Operator for Similarity Joins in Data Cleaning. In *ICDE '06: 22nd International Conference on Data Engineering*, page 5. IEEE Computer Society, 2006. 41

Jilin Chen, Rowan Nairn, Les Nelson, Michael Bernstein, and Ed H Chi. Short and Tweet : Experiments on Recommending Content from Information Streams. In *CHI '10: 28th International Conference on Human Factors in Computing Systems*, pages 1185–1194. ACM, 2010. 104

S Chen and S W Schlosser. Map-Reduce Meets Wider Varieties of Applications. Technical Report IRP-TR-08-05, Intel Research, Pittsburgh, May 2008. URL `http://www.pittsburgh.intel-research.net/~chensm/papers/IRP-TR-08-05.pdf`. 36

Flavio Chierichetti, Ravi Kumar, and Andrew Tomkins. Max-Cover in Map-Reduce. In *WWW '10: 19th International Conference on World Wide Web*, pages 231–240, New York, New York, USA, 2010. ACM Press. ISBN 9781605587998. 35

Paul Christiano, Jonathan A Kelner, Aleksander Madry, Daniel A Spielman, and Shang-Hua Teng. Electrical Flows, Laplacian Systems, and Faster Approximation of Maximum Flow in Undirected Graphs. *STOC '10: 43rd ACM Symposium on Theory of Computing*, pages 273—-282, 2010. 71

Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R Bradski, Andrew Y Ng, and Kunle Olukotun. Map-Reduce for Machine Learning on Multicore. *Advances in Neural Information Processing Systems*, 19:281–288, 2006. 36

E F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970. 10

Jonathan Cohen. Graph Twiddling in a Map Reduce World. *Computing in Science and Engineering*, 11(4):29–41, 2009. 35

T Condie, N Conway, P Alvaro, J M Hellerstein, K Elmeleegy, and R Sears. MapReduce Online. Technical Report UCB/EECS-2009-136, University of California, Berkeley, October 2009. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-136.html`. 29, 117

B F Cooper, R Ramakrishnan, U Srivastava, A Silberstein, P Bohannon, H A Jacobsen, N Puz, D Weaver, and R Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *VLDB Endowment*, 1(2):1277–1288, August 2008. 22

Graham Cormode, Flip Korn, and Srikanta Tirthapura. Exponentially Decayed Aggregates on Data Streams. In *ICDE '08: 24th International Conference on Data Engineering*, pages 1379 – 1381. IEEE, 2008. 115, 118

Mache Creeger. Cloud Computing: An Overview. *ACM Queue*, 7(5):2:3—-2:4, June 2009. 13

Abhinandan Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google News Personalization: Scalable Online Collaborative Filtering. In *WWW '07: 16th International Conference on World Wide Web*, pages 271–280. ACM, AVM, 2007. 104

J Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, January 2010. 13

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data processing on Large Clusters. In *OSDI '04: 6th Symposium on Opearting Systems Design and Implementation*, pages 137–150. USENIX Association, December 2004. 15, 23, 24

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008. 24

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP '07: 21st Symposium on Operating Systems Principles*, pages 205–220. ACM, October 2007. 22

D J DeWitt and M Stonebraker. MapReduce: A major step backwards, January 2008. URL `http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards http://databasecolumn.vertica.com/database-innovation/mapreduce-ii`. 13

D J DeWitt, S Ghandeharizadeh, D A Schneider, A Bricker, H.-I. Hsiao, and R Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990. 11

David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992. 11

Tamer Elsayed, Jimmy Lin, and Douglas W Oard. Pairwise document similarity in large collections with MapReduce. In *HLT '08: 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies*, pages 265–268. Association for Computational Linguistics, June 2008. 39, 42, 43

M Ester, H P Kriegel, J Sander, and X Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD '96: 2nd International Conference on Knowledge Discovery and Data mining*, volume 1996, pages 226–231. AAAI Press, 1996. 38

J Feigenbaum, S Kannan, A McGregor, S Suri, and J Zhang. On Graph Problems in a Semi-Streaming Model. *Theoretical Computer Science*, 348(2-3):207–216, 2005. 30

Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, Clifford Stein, and Zoya Svitkina. On the Complexity of Processing Massive, Unordered, Distributed Data. *Arxiv*, 2007. 30

Ted Fischer, Andrew Goldberg, David Haglin, and Serge Plotkin. Approximating Matchings in Parallel. *Information Processing Letters*, 46(3):115–118, 1993. 71

H Gabow. An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems. In *STOC '83: 15th ACM Symposium on Theory of Computing*, pages 448–456, 1983. 69, 71

Sandra Garcia Esparza, Michael P O'Mahony, Barry Smyth, Sandra Garcia Esparza, and Michael P O Mahony. On the Real-Time Web as a Source of Recommendation Knowledge. In *RecSys '10: 4th ACM Conference on Recommender Systems*, RecSys'10, pages 305–308. ACM, January 2010. 104

N Garg, T Kavitha, A Kumar, K Mehlhorn, and J Mestre. Assigning papers to referees. *Algorithmica*, 58(1):119–136, 2010. 71

Oscar Garrido, Stefan Jarominek, Andrzej Lingas, and Wojciech Rytter. A Simple Randomized Parallel Algorithm for Maximal f-Matchings. *Information Processing Letters*, 57(2):83–87, 1996. 69, 71, 76, 77, 79, 80, 81

Gartner. Gartner Says Solving 'Big Data' Challenge Involves More Than Just Managing Volumes of Data, 2011. URL http://www.gartner.com/it/page.jsp?id=1731916. 3

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP '03: 19th ACM symposium on Operating Systems Principles*, pages 29–43. ACM, October 2003. 22

J Ginsberg, M H Mohebbi, R S Patel, L Brammer, M S Smolinski, and L Brilliant. Detecting influenza epidemics using search engine query data. *Nature*, 457 (7232):1012–1014, 2008. 5

Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity Search in High Dimensions via Hashing. In *VLDB '99: 25th International Conference on Very Large Data Bases*, pages 518–529, 1999. 42

Andrew V Goldberg and Satish Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45(5):783–797, 1998. 69, 71

David Goldberg, David Nichols, Brian M Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, December 1992. 104

A Gulli and A Signorini. The Indexable Web is More than 11.5 billion pages. In *WWW '05: Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, pages 902–903, New York, NY, USA, 2005. ACM. ISBN 1-59593-051-5. 6

John L Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31 (5):532–533, May 1988. 12

Alon Halevy, Peter Norvig, and Fernando Pereira. The Unreasonable Effectiveness of Data. *IEEE Intelligent Systems*, 24(2):8–12, March 2009. ISSN 1541-1672. 5

Ben Hammersley. My speech to the IAAC, 2011. URL http://www.benhammersley.com/2011/09/my-speech-to-the-iaac/. 6

Joseph M Hellerstein. Programming a Parallel Future. Technical Report UCB/EECS-2008-144, University of California, Berkeley, November 2008. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-144.html. 5

Joseph M Hellerstein, Peter J Haas, and Helen J Wang. Online aggregation. In *SIGMOD '97: Proceedings of the 23rd ACM International Conference on Management of Data*, pages 171–182. ACM, May 1997. 29

M.R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. Technical report, Digital Systems Research Center, 1998. URL `http://www.eecs.harvard.edu/~michaelm/E210/datastreams.pdf`. 30

Djoerd Hiemstra and Claudia Hauff. MapReduce for information retrieval evaluation: âĂIJLetâĂŹs quickly test this on 12 TB of dataâĂİ. In *CLEF '10: 2010 Conference on Multilingual and Multimodal Information Access Evaluation*, pages 64–69, September 2010. ISBN 3-642-15997-4, 978-3-642-15997-8. 34

T Hofmann. Probabilistic latent semantic indexing. *SIGIR '99: 22nd International Conference on Research and Development in Information Retrieval*, January 1999. 110

Patrick Hunt, Mahadev Konar, F.P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIXATC '10: USENIX Annual Technical Conference*, pages 1–14. USENIX Association, 2010. 21

Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys '07: 2nd European Conference on Computer Systems*, pages 59–72. ACM, March 2007. 23

Adam Jacobs. The Pathologies of Big Data. *Communications of the ACM*, 52(8): 36–44, August 2009. ISSN 00010782. 3

Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems*, 20(4):422–446, October 2002. ISSN 1046-8188. 127

Akshay Java, Xiaodan Song, Tim Finin, and Belle Tseng. Why We Twitter: Understanding Microblogging Usage and Communities. In *WebKDD-SNA-KDD '07: Joint 9th WebKDD and 1st SNA-KDD Workshop on Web Mining and Social Network Analysis*, pages 56–65. ACM, 2007. 104

T Jebara, J Wang, and S F Chang. Graph construction and b-matching for semi-supervised learning. In *ICML '09: 26th International Conference on Machine Learning*, pages 441–448, 2009. 71

Tony Jebara and Vlad Shchogolev. B-Matching for Spectral Clustering. In *ECML '06: 17th European Conference on Machine Learning*, pages 679–686, 2006. 71

Thorsten Joachims. Optimizing search engines using clickthrough data. In *KDD '02: 8th International Conference on Knowledge Discovery and Data Mining*, pages 133–142. ACM, 2002. 120

U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *ICDM '09: 9th International Conference on Data Mining*, pages 229–238. IEEE Computer Society, December 2009. 35

U. Kang, CE Charalampos E. Tsourakakis, Ana Paula Appel, Christos Faloutsos, and Jure Leskovec. HADI: Mining Radii of Large Graphs. *ACM Transactions on Knowledge Discovery from Data*, 5(2):1–24, 2011. 35

H Karloff, S Suri, and S Vassilvitskii. A Model of Computation for MapReduce. In *SODA '10: Symposium on Discrete Algorithms*, pages 938–948. ACM, January 2010. 27, 53

Christos Koufogiannakis and Neal Young. Distributed Fractional Packing and Maximum Weighted b-Matching via Tail-Recursive Duality. In *DISC '09: 23rd International Symposium on Distributed Computing*, pages 221–238, 2009. 71

Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *WWW '10: 19th International Conference on World Wide Web*, pages 591–600, New York, New York, USA, 2010. ACM, ACM Press. ISBN 9781605587998. 104, 105

Frank La Rue. Report of the Special Rapporteur on the promotion and protection of the right to freedom. Technical Report May, United Nations, 2011. 5

Avinash Lakshman and Prashant Malik. Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35—-40, April 2010. ISSN 01635980. 22

Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998. 21

Silvio Lattanzi, Benjamin Moseley, S. Suri, and Sergei Vassilvitskii. Filtering: A Method for Solving Graph Problems in MapReduce. In ACM, editor, *SPAA '11: 23rd Symposium on Parallelism in Algorithms and Architectures*, pages 85—-94, 2011. 35, 71

Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y Chang. PFP: Parallel FP-Growth for Query Recommendation. In *RecSys '08: 2nd ACM Conference on Recommender Systems*, pages 107–114. ACM, October 2008. 34

J Lin, D Metzler, T Elsayed, and L Wang. Of Ivory and Smurfs: Loxodontan MapReduce experiments for Web search. In *TREC '09: 18th Text REtrieval Conference*, November 2009. 34

Jimmy Lin and Chris Dyer. Data-Intensive Text Processing with MapReduce. *Synthesis Lectures on Human Language Technologies*, 3(1):1–177, 2010. ISSN 19474040. 34

Jimmy Lin and Michael Schatz. Design Patterns for Efficient Graph Algorithms in MapReduce. In *MLG '10: 8th Workshop on Mining and Learning with Graphs*, pages 78–85. ACM, 2010. 35

Bing Liu. *Web data mining*. Springer, 2007. 6, 7

Chao Liu, Hung-chih Yang, Jinliang Fan, Li-Wei He, and Yi-Min Wang. Distributed Nonnegative Matrix Factorization for Web-Scale Dyadic Data Analysis on MapReduce. In *WWW '10: 19th International Conference on World Wide Web*, pages 681—-690, New York, New York, USA, April 2010. ACM Press. ISBN 9781605587998. 35

Zhiyuan Liu, Yuzhou Zhang, Edward Y. Chang, and Maosong Sun. PLDA+: Parallel Latent Dirichlet Allocation with Data Placement and Pipeline Processing. *ACM Transactions on Intelligent Systems and Technology*, 2(3):26, April 2011. ISSN 2157-6904. 34

Zvi Lotker, Boaz Patt-Shamir, and Seth Pettie. Improved distributed approximate matching. In *SPAA '08: 20th Symposium on Parallelism in Algorithms and Architectures*, pages 129–136, 2008. 71

Mike Loukides. What is Data Science? Technical report, O'Reilly Radar, 2010. URL `http://cdn.oreilly.com/radar/2010/06/What_is_Data_Science.pdf`. 3

Mike Loukides. The Evolution of Data Products. Technical report, O'Reilly Radar, 2011. URL `http://cdn.oreilly.com/radar/2011/09/Evolution-of-Data-Products.pdf`. 5

Grzegorz Malewicz, Matthew H Austern, Aart J C Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *SIGMOD '10: 36th International Conference on Management of Data*, SIGMOD '10, pages 135–145, New York, NY, USA, June 2010. ACM. ISBN 978-1-4503-0032-2. 23

G.S. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. In *VLDB '02: 28th International Conference on Very Large Data Bases*, pages 346–357. VLDB Endowment, 2002. 31

R M C McCreadie, C Macdonald, and I Ounis. Comparing Distributed Indexing: To MapReduce or Not? In *LSDS-IR '09: 7th Workshop on Large-Scale Distributed Systems for Information Retrieval*, pages 41–48, 2009a. 34

Richard McCreadie, Craig Macdonald, and Iadh Ounis. On single-pass indexing with MapReduce. In *SIGIR '09: 32nd International Conference on Research and development in Information Retrieval*, pages 742–743. ACM, July 2009b. 34, 46

Richard McCreadie, Craig Macdonald, and Iadh Ounis. News Article Ranking: Leveraging the Wisdom of Bloggers. *RIAO '10: Adaptivity, Personalization and Fusion of Heterogeneous Information*, pages 40–48, January 2010. 105

Richard McCreadie, Craig Macdonald, and Iadh Ounis. MapReduce indexing strategies: Studying scalability and efficiency. *Information Processing & Management*, pages 1–18, February 2011. ISSN 03064573. 34, 46

Peter Mell and Tim Grance. Definition of Cloud Computing, October 2009. URL `http://csrc.nist.gov/groups/SNS/cloud-computing`. 13

Matthew Michelson and Sofus A Macskassy. Discovering Users âĂŹ Topics of Interest on Twitter : A First Look. In *AND '10: 4th Workshop on Analytics for Noisy Unstructured Text Data*, pages 73–80, 2010. 106

Leonardo Neumeyer, Bruce Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *ICDMW '10: 10th International Conference on Data Mining Workshops*, pages 170–177. IEEE, 2010. 15, 23, 31

Michael G Noll and Christoph Meinel. Building a Scalable Collaborative Web Filter with Free and Open Source Software. In *SITIS '08: 4th IEEE International Conference on Signal Image Technology and Internet Based Systems*, pages 563–571. IEEE Computer Society, November 2008. 36

Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD '08: 34th International Conference on Management of Data*, pages 1099–1110. ACM, June 2008. 15, 24

Iadh Ounis, Gianni Amati, Vassilis Plachouras, Ben He, Craig Macdonald, and Christina Lioma. Terrier: A High Performance and Scalable Information Retrieval Platform. In *OSIRâĂŹ06: 2nd International Workshop on Open Source Information Retrieval*, pages 18–25, 2006. 34

Rasmus Pagh and Charalampos E. Tsourakakis. Colorful Triangle Counting and a MapReduce Implementation. *Arxiv*, pages 1–9, 2011. 35

Alessandro Panconesi and Mauro Sozio. Fast primal-dual distributed algorithms for scheduling and matching problems. *Distributed Computing*, 22(4):269–283, March 2010. ISSN 0178-2770. 69, 70, 75, 81

Patrick Pantel, Eric Crestan, Arkady Borkovsky, Ana-Maria Popescu, and Vishnu Vyas. Web-Scale Distributional Similarity and Entity Set Expansion. In *EMNLP '09: 2009 Conference on Empirical Methods in Natural Language Processing*, page 938. Association for Computational Linguistics, 2009. ISBN 9781932432626. 34

Spiros Papadimitriou and Jimeng Sun. DisCo: Distributed Co-clustering with Map-Reduce: A Case Study towards Petabyte-Scale End-to-End Mining. In *ICDM '08: 8th International Conference on Data Mining*, pages 512–521. IEEE Computer Society, December 2008. 35

Deepa Paranjpe. Learning document aboutness from implicit user feedback and document structure. In *CIKM '09: 18th Conference on Information and Knowledge Mining*, pages 365–374, New York, New York, USA, 2009. ACM, ACM Press. ISBN 9781605585123. 106, 110, 124

M Penn and M Tennenholtz. Constrained multi-object auctions and b-matching. *Information Processing Letters*, 75(1-2):29–34, 2000. 71

Owen Phelan, K. McCarthy, Mike Bennett, and Barry Smyth. Terms of a Feather : Content-Based News Recommendation and Discovery Using Twitter. *Advances in Information Retrieval*, 6611(07):448–459, 2011. 105

Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, October 2005. 23

Anand Rajaraman and Jeffrey D Ullman. *Mining of Massive Datasets*. Stanford University, 2010. 24, 35

Benjamin Reed and Flavio P. Junqueira. A simple totally ordered broadcast protocol. In *LADIS '08: 2nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '08, pages 2:1—-2:6, New York, NY, USA, September 2008. ACM. 21

Jennifer Rowley. The wisdom hierarchy: representations of the DIKW hierarchy. *Journal of Information Science*, 33(2):163–180, April 2007. 4

Sunita Sarawagi and Alok Kirpal. Efficient set joins on similarity predicates. In *SIGMOD '04: 30th International Conference on Management of Data*, pages 743–754. ACM, 2004. 42

M C Schatz. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363, June 2009. 36

Sangwon Seo, Edward J. Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. HAMA: An Efficient Matrix Computation with the

MapReduce Framework. In *CloudCom '10: 2nd International Conference on Cloud Computing Technology and Science*, pages 721–726. IEEE, November 2010. ISBN 978-1-4244-9405-7. 23

Y Shi. Reevaluating Amdahl's Law and Gustafson's Law. October 1996. URL `http://www.cis.temple.edu/~shi/docs/amdahl/amdahl.html`. 12

M Stonebraker, C Bear, U Cetintemel, M Cherniack, T Ge, N Hachem, S Harizopoulos, J Lifter, J Rogers, and S Zdonik. One size fits all? Part 2: Benchmarking results. In *CIDR '07: 3rd Conference on Innovative Data Systems Research*, January 2007a. 10

M Stonebraker, S Madden, D J Abadi, S Harizopoulos, N Hachem, and P Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *VLDB '07: 33rd International Conference on Very Large Data Bases*, pages 1150–1160. ACM, September 2007b. 10

Michael Stonebraker. The Case for Shared Nothing. *IEEE Data Engineering Bulletin*, 9(1):4–9, March 1986. 11

Michael Stonebraker and UÇğur Çetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *ICDE '05: 21st International Conference on Data Engineering*, pages 2–11. IEEE Computer Society, Ieee, April 2005. ISBN 0-7695-2285-8. 10

Michael Stonebraker, Daniel Abadi, David J DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MapReduce and Parallel DBMSs: Friends or Foes? *Communications of the ACM*, 53(1):64–71, January 2010. 13

Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW '11: 20th International Conference on World Wide Web*, pages 607–614. ACM, 2011. ISBN 9781450306324. 35

Jaime Teevan, Daniel Ramage, and Merredith Ringel Morris. # TwitterSearch : A Comparison of Microblog Search and Web Search. In *WSDM '11: 4th International Conference on Web Search and Data Mining*, pages 35–44, 2011. 104

Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *VLDB Endowment*, 2 (2):1626–1629, August 2009. 24

Alvin Toffler. *Future shock*. Random House Publishing Group, 1984. 99

Charalampos E Tsourakakis, U Kang, Gary L Miller, and Christos Faloutsos. DOULION: Counting Triangles in Massive Graphs with a Coin. In *KDD '09: 15th International Conference on Knowledge Discovery and Data Mining*, pages 837–846. ACM, April 2009. 35

Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990. ISSN 00010782. 23

Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using MapReduce. In *SIGMOD '10: 36th International Conference on Management of Data*, pages 495–506, New York, New York, USA, 2010. ACM Press. ISBN 9781450300322. 39, 44

Werner Vogels. Eventually Consistent. *ACM Queue*, 6(6):14—-19, October 2008. ISSN 15427730. 22

Ellen M Voorhees. The TREC-8 Question Answering Track Report. In *TREC '99: 8th Text REtrieval Conference*, 1999. 126

Mirjam Wattenhofer and Roger Wattenhofer. Distributed Weighted Matching. In *Distributed Computing*, pages 335–348, 2004. 71

Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In *WWW '08: 17th International Conference on World Wide Web*, pages 131–140. ACM, 2008. 42, 45

Chuan Xiao, Wei Wang, Xuemin Lin, and Haichuan Shang. Top-k Set Similarity Joins. In *ICDE '09: 25th International Conference on Data Engineering*, pages 916–927. IEEE Computer Society, 2009. 40

Yahoo! and Facebook. Yahoo! Research Small World Experiment, 2011. URL `http://smallworld.sandbox.yahoo.com/`. 7

H Yang, A Dasdan, R L Hsiao, and D S Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD '07: 33rd International Conference on Management of Data*, pages 1029–1040. ACM, June 2007. 28

J.H. Yoon and S.R. Kim. Improved Sampling for Triangle Counting with MapReduce. In *ICHIT '11: 5th International Conference on Convergence and Hybrid Information Technology*, pages 685–689. Springer, 2011. 35

Y Yu, M Isard, D Fetterly, M Budiu, U Erlingsson, P K Gunda, and J Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI '08: 8th Symposium on Operating System Design and Implementation*, December 2008. 24

Bin Zhou, Daxin Jiang, Jian Pei, and Hang Li. OLAP on search logs: an infrastructure supporting data-driven applications in search engines. In *KDD '09: 15th International Conference on Knowledge Discovery and Data Mining*, pages 1395–1404. ACM, June 2009. 34