Analysis and Mining of Large Dynamic Graphs: case of graph clustering

Wissem Inoubli

UNIVERSITY OF TUNIS EL MANAR
LIPAH - LR11ES14

P H D T H E S I S

To obtain the degree of

Doctor of Philosophy

in Computer Science

Defended by

Wissem INOUBLI

Analysis and Mining of Large Dynamic Graphs: case of graph clustering

publicly defended on January, $07^{th}, 2021$

Committee:

Reviewers:	
Pr. Lotfi Ben Romdhane	University of Sousse, Tunisia
Pr. Osmar Zaiane	University of Alberta, Canada
Examiners:	
Pr. Anis Yazidi	Oslo Metropolitan University, Norway
Pr. Mohamed Mohssen Gamoudi	University of manouba, Tunisia
Advisors:	
Dr. Sabeur Aridhi	University of lorraine, France
Pr. Amel Borgi	University of Tunis El Manar, Tunisia

Acknowledgments

This thesis would not have been possible without the help, support and patience of my advisors, Dr. Sabeur Aridhi, Prof. Engelbert Mephu Nguifo, Prof. Mondher Maddouri, Pr. Amel Borgi and Dr. Haithem Mezni, not to mention their advice and vast knowledge in big data, graph mining and machine learning. Special thanks are directed to Prof. Lotfi Ben Romdhane and Prof. Osmar Zaiane for having accepted to review my thesis manuscript and for their kind efforts to comply with all the administrative constraints. My thanks go as well to Prof. Mohamed Mohhsen Gamoudi and Prof. Anis Yazidi for having accepted to act as examiners of my thesis.

Special thanks to Prof. Sadok ben yahia for his support and the help in administrative procedures.

I would also like to thank my father, mother, brother, sister and all my family. They have always been supporting and encouraging me with their best wishes. Their faith in me allowed me to be as ambitious as I wanted and helped me a lot through the past years. It was under their watchful eye that I gained so much drive and ability to tackle challenges head on.

Also an acknowledgments to the University of Clermont Auvergne, LIPAH laboratory, LORIA laboratory and LIMOS laboratory. Especially, LIMOS for providing us the opportunity to perform the experiment parts of this thesis on the GALACTICA data center, the Tunisian Ministry of Higher Education and LIPAH laboratory for the financial support of my PhD scholarships and LORIA laboratory for having hosted me during research internships.

At last but certainly not least, I would like to thank my friends, they were always there cheering me up and standing by me through both good and bad times.

Nancy, 2020, Wissem Inoubli

List of Figures

2.1 MapReduce architecture 15
2.2 HDFS architecture 16
2.3 Spark system overview 18
2.4 WordCount example with Spark 21
2.5 Topology of a Storm program and architecture 22
2.6 WordCount example with Storm 23
2.7 Samza architecture 24
2.8 WordCount example with Samza 24
2.9 Flink architecture 25
2.10 WordCount with Flink 26
2.11 *Batch Mode* scenario 30
2.12 *Stream mode* scenario 32
2.13 Architecture of our personalized monitoring tool 33
2.14 Impact of the size of the data on the average processing time: case of small datasets 34
2.15 Impact of the size of the data on the average processing time: case of big datasets 34
2.16 Impact of the number of machines on the average processing time (WordCount workload with 50 Gb of data) 35
2.17 Impact of iterative processing on the average processing time . . 36
2.18 Impact of the number of iterations on the average processing time (Kmeans workload with 10 million examples) 37
2.19 Impact of HDFS block size on the runtime (Kmeans workload with 10 million examples and 10 iterations) 38
2.20 Impact of the cluster manager on the performance of the studied frameworks 39
2.21 Impact of bandwidth on the performance of the studied frameworks 39
2.22 Impact of parallelism parameters on the performance of Flink . . 40

2.23 Impact of the memory size on the performance of Flink 41
2.24 Impact of the number of workers on the performance of Spark . . 41
2.25 Impact of the memory size on the performance of Spark 42
2.26 Impact of the number of slots on the performance of Hadoop . . 43
2.27 Impact of the memory size on the performance of Hadoop 43

3.1 Vertex-centric programming model 57
3.2 Edge-centric programming model 58
3.3 GAS programming model 58
3.4 Block-centric programming model flow chart 59

4.1 Running example ($\varepsilon = 0.7$ and $\mu = 3$). 87
4.2 Illustrative example 91
4.3 Partitioned graph G used in the running example in Section 4.1.2 93
4.4 Impact of the graph size on the processing time of both SCAN and DSCAN 97
4.5 Impact of the number of workers on the running time (with $\varepsilon = 0.5$, $\mu = 3$). 98
4.6 Impact of ε size on the running time of DSCAN 99
4.7 Performance of DSCAN phases 100
4.8 Impact of the partitionning step on DSCAN response time (ε=0.5, μ=3) 101

5.1 Different graph representations from static to dynamic graph . . 107
5.2 Partitioned graph G (P1, P2, P3), affected vertices, edges when deleting vertex 3 115
5.3 Impact of the number of updates on the processing time of the pSCAN, ppSCAN and DISCAN (ε=0.5, μ=3) 122
5.4 Impact of the n workers' number on the processing time of DISCAN (ε=0.5, μ=3) 123
5.5 Impact of the batch size on the running time (ε=0.5, μ=3) . . . 123
5.6 Impact of the vertices' type (internal or external) on the processing time (ε=0.5, μ=3) 124

5.7 Performance of DISCAN steps (ε=0.5, μ=3) 125

List of Tables

2.1 Comparative study of popular Big Data frameworks 27
2.2 Graph datasets. 31

3.1 Graph processing models comparative study 60
3.2 Graph processing frameworks comparative study 66
3.3 Comparative study on the graph clustering algorithms 73
3.4 Comparative study on existing structural graph clustering approaches 76

4.1 Graph datasets 96

Contents

1 Introduction **1**

1.1 Context and motivations . . . 2

1.1.1 Big Data emergence . . . 2

1.1.2 From Big Data to Big Graphs . . . 2

1.1.3 Challenges of big graph processing . . . 3

1.2 Contributions . . . 4

1.2.1 First axis: a comparative study on Big Data frameworks . 5

1.2.2 Second axis: distributed structural graph clustering . . . 5

1.2.3 Third axis: dynamic and distributed structural graph clustering . . . 5

1.3 Outline . . . 6

I Background and related works **7**

2 Big Data Frameworks **9**

2.1 Introduction to Big Data . . . 11

2.2 Popular Big Data frameworks . . . 11

2.2.1 Presentation of Big Data frameworks . . . 13

2.2.2 Apache Hadoop . . . 13

2.2.2.1 Hadoop system overview . . . 13

2.2.2.2 WordCount example with Hadoop . . . 17

2.2.3 Apache spark . . . 18

2.2.3.1 Spark system overview . . . 18

2.2.3.2 WordCount example with Spark . . . 20

2.2.4 Apache Storm . . . 20

2.2.4.1 Storm system overview . . . 20

2.2.4.2 WordCount example with Storm . . . 22

2.2.5 Apache Samza . . . 23

2.2.5.1 Samza system overview . . . 23
2.2.5.2 WordCount example with Samza . . . 24
2.2.6 Apache Flink . . . 25
2.2.6.1 Flink system overview . . . 25
2.2.6.2 WordCount example with Flink . . . 26
2.2.7 Categorization of Big Data frameworks . . . 26
2.3 Comparative study of Big Data frameworks . . . 29
2.3.1 Experimental environment . . . 29
2.3.1.1 Experimental environment . . . 29
2.3.2 Experimental results . . . 33
2.3.2.1 Batch mode . . . 33
2.3.2.2 Stream mode . . . 44
2.3.2.3 Summary of the evaluation . . . 44
2.4 Real-world applications and best practices . . . 45
2.4.1 Real-world applications . . . 45
2.4.1.1 Healthcare applications . . . 45
2.4.1.2 Recommendation systems . . . 46
2.4.1.3 Social media . . . 47
2.4.1.4 Smart cities . . . 47
2.4.2 Best practices . . . 48
2.4.2.1 Stream processing . . . 48
2.4.2.2 Micro-batch processing . . . 49
2.4.2.3 Machine learning algorithms . . . 49
2.4.2.4 Big graph processing . . . 50
2.5 Conclusion . . . 50

3 Graph mining 53
3.1 Graph: definitions . . . 55
3.2 Related work on big graph: processing models and frameworks . . . 56
3.2.1 Graph programming models . . . 56
3.2.1.1 Vertex-centric model . . . 56
3.2.1.2 Edge-centric model . . . 57

3.2.1.3 Gather-Apply-Scatter model (GAS) 57
3.2.1.4 Block-centric model 59
3.2.2 Summary of graph processing models 60
3.3 Related work on big graph processing frameworks 61
3.3.1 Graph processing frameworks 61
3.3.2 Graph processing frameworks: an overview 62
3.3.3 Summary of graph processing framworks 66
3.4 Related work on graph clustering 67
3.4.1 Modularity based algorithm 68
3.4.2 Markov Clustering (MCL) 69
3.4.3 Label propagation algorithm 70
3.4.4 Spectral clustering 71
3.4.5 Structural graph clustering 71
3.4.6 Summary of graph clustering algorithms 72
3.4.7 Related work on structural graph clustering 74
3.5 Conclusion 77

II Contributions 79

4 Distributed Structural Graph Clustering Algorithm 81
4.1 Strctural graph clustering 83
4.1.1 Definitions and basic concepts 83
4.1.2 Running example 85
4.2 DSCAN: A Distributed Algorithm for Large-Scale Graph Clustering 87
4.2.1 Distributed graph partitioning 87
4.2.2 Distributed clustering 88
4.2.2.1 Illustrative example 93
4.2.2.2 Time complexity analysis of DSCAN 94
4.2.2.3 Communication complexity 95
4.3 Experiments 95
4.3.1 Experimental protocol 95
4.3.2 Experimental data 96

4.3.3 Experimental results related to DSCAN 96
4.4 Conclusion 102

5 Distributed and Incremental Structural Graph Clustering Algorithm 105
5.1 Introduction to dynamic graphs 107
5.2 Dynamic graphs: definitions 107
5.3 DISCAN: Proposed incremental graph clustering 108
5.3.1 Distributed graph update 109
5.3.1.1 Adding a new vertex 109
5.3.1.2 Adding a new edge 110
5.3.1.3 Delete a vertex 110
5.3.1.4 Delete an edge 112
5.3.2 Distributed graph clustering 114
5.3.3 Time complexity analysis of DISCAN 119
5.3.3.1 Communication complexity 119
5.4 Experimental result 119
5.5 Conclusion 125

6 Conclusion and prospects 129
6.1 Summary of contributions 130
6.1.1 Comparative study on big data frameworks 130
6.1.2 Distributed structural graph clustering 130
6.1.3 Dynamic graph clustering 131
6.2 Future works and prospects 131
6.2.1 First axis: improvement of the partitioning method proposed by DSCAN 131
6.2.2 Second axis: improvement of the communication 132
6.2.3 Third axis: evolving graphs 132

Bibliography 135

Chapter 1

Introduction

Contents

1.1 Context and motivations 2

1.1.1 Big Data emergence 2

1.1.2 From Big Data to Big Graphs 2

1.1.3 Challenges of big graph processing 3

1.2 Contributions 4

1.2.1 First axis: a comparative study on Big Data frameworks . . 5

1.2.2 Second axis: distributed structural graph clustering 5

1.2.3 Third axis: dynamic and distributed structural graph clustering 5

1.3 Outline 6

Goals

This chapter summarizes the contents and outlines of the thesis plan. First, we highlight the emergence of Big Data and big graph analysis. Then, we state issues to be addressed in the fallowing chapters.

1.1 Context and motivations

1.1.1 Big Data emergence

Every day, 2.5 terabytes of data are generated in the digital world. In the last five years, the size of data is expected to multiply by 50 times [Gandomi 2015a]. According to Statista [1], the global volume of data in the world will be multiplied by 3.7 between 2020 and 2025. Then it will be multiplied by 3.5 every five years until 2035 to reach 2,142 zettabytes of data. For instance, Google receives 40,000 requests every second, 72 videos are uploaded every minute to YouTube, and 217 new smartphone users are counted every minute [Gandomi 2015a]. These data come to us various sources, including smartphones (log files, calls, etc.), social networks, videos and digital images, customer transactions, shape or movement sensors of connected objects, etc.

The development and access to these data caused the emergence of a new phenomenon called "Big Data". This phenomenon had a particular impact on companies that have to handle terabytes or even petabytes of data requiring, specific infrastructures for their creation, storage, processing, analysis, and recovery. In other words, they manage and process a lot of data that come in streams or in batches. Such data exceed the capacity of traditional resources such as relational databases, sequential processing, classic development tools, etc. In addition, traditional data models, like the relational database model, are unable to support these volumes of data. Consequently, advanced data models have been introduced such as NoSQL databases model and graph-based models.

1.1.2 From Big Data to Big Graphs

Data can take several representations. Apart from the relational model, the graph model has gained increasing interest to represent various types of data, it has been extremely used in different applications, especially computer networks [Faloutsos 1999], social networks [Bonchi 2011], [Liu 2009] and bioinformatics [Saidi 2009]. These applications use the graph representation to describe the

[1] https://fr.statista.com/

relationships between the data. In bioinformatics, the structure of the proteins can be considered as a graph. Amino acids represent the nodes whereas their interactions seen edges. In fact, studying protein graphs allows us to better understand the structures of proteins through the extraction of information that is not visible using the classical representation structure of protein data. In the area of social network analysis, graph representation is used to model the interactions between users in a social network. Analyzing these interactions can help, for example, to identify the transmission routes of a rumor or a joke [Liu 2009].
In another way, some graphs are naturally dynamic in terms of their structures or their properties (i.e. some vertices and/or edges can be deleted from or added to the initial graph, and some properties of the graph can be changed).
In the next section, some challenges related to this thesis, mainly big graph processing and mining, will be presented and discussed.

1.1.3 Challenges of big graph processing

The two fields of big graph analytics and large-scale graph processing represent a continuity of Big Data challenges. In the last years, big graphs analytics have become among the first research tracks in the database research community. Also, big graphs attract the attention of companies in the context of industrial projects. Facebook and Google have proposed their own graph processing systems and have used graphs in several use cases.
As a result, large graphs can contain terabytes of compressed data when they are stored on disks. Therefore a single computer will not be able to process such volume of data. This incapacity causes a lot of problems mainly in the context of data processing and indexing, despite the fact that graphs represent an important source of knowledge useful for business. Therefore, many algorithms and solutions have been proposed to explore and exploit large graphs in order to analyze them and to discover hidden insights. Multiple applications of large graphs are present today inside technologies and daily routine such as social networks, protein interactions and road networks. Recently, the discovery of dense subgraphs in large graphs has become one of the major issues in the graph field. This is known

as the problem of graph clustering and community detection, and it is gradually attracting the attention of a larger research community. The graph clustering aims mainly to detect groups or hidden structures in a given graph [Rossetti 2018]. For example, in social networks (e.g., Twitter), groups of users with specific conditions can be considered to be communities [Rossetti 2018]. In a collaborative network (for example the co-authorship database DBLP), a cluster can be a group of researchers with similar research interests. In computational biology, calculating functional clusters of genes can help biologists to better study the gene networks. real-world graphs are characterized not only by their huge size but also by their heterogeneity. Besides, the dynamic nature of several real-world graphs is also a major issue. For example, Facebook has put about 86,400 objects per second in 2013 [Armstrong 2013], while Twitter traffic [Sengupta 2016] can reach 143 thousand tweets per second (we can also consider graph updates), and email communication network [Sengupta 2016] can exceed 2.5 millions of new emails per second [2]. This speed of graph updates makes graph analytics algorithms (e.g., clustering in our case) need real-time and obviously scalable processing.

1.2 Contributions

This thesis deals with distributed and incremental structural graph clustering. Firstly, we present an experimental study on data processing. This comparative study is divided into two parts: (1) theoretical study and (2) experimental study. Secondly, we propose a distributed algorithm for graph clustering. The proposed approach is based on the structural graph clustering [Xu 2007] and implemented on top of the BLADYG [Aridhi 2017] framework. Thirdly, we extend the proposed algorithm in order to support the dynamic graphs.

[2] https://www.internetlivestats.com/one-second/email-band

1.2.1 First axis: a comparative study on Big Data frameworks

In this contribution, we study the most popular frameworks for large-scale data processing in both stream and batch processing. We, first, present a brief description of some major Big Data framework: Hadoop, Spark, Storm, Samza and Flink. We also presented a categorization of these frameworks according to some features as being mainly related to the processing mode, data source, Machine learning compatibility and others. We also conducted an extensive comparative study of the above-mentioned frameworks on a cluster of machines and we highlighted best practices with respect to each big data framework's capacity. This comparative study has been published in [Inoubli 2018b].

1.2.2 Second axis: distributed structural graph clustering

In this part of our thesis, after an extensive literature review on both graph clustering algorithms and large graph processing frameworks, we propose a novel algorithm for graph clustering in a distributed setting. The proposed algorithm is implemented on top of BLADYG framework. It supports both centralized and distributed graphs. In a centralized graph setting, the proposed algorithm starts by splitting the graph into subgraphs, using our proposed graph partitioning. In the distributed graph setting, the proposed algorithm takes the already distributed graph as input and lunch directly the graph clustering task. The proposed algorithm is based on the structural graph clustering algorithm SCAN [Xu 2007]. We compare our approach with four other methods based on SCAN. We have shown through an experimental study that the proposed algorithm is scalable even with big distributed graphs.

1.2.3 Third axis: dynamic and distributed structural graph clustering

The third axis of this work deals with dynamic graphs. We extend our distributed graph clustering algorithm in order to support big and dynamic graphs. Thus, an

incremental graph clustering has been proposed. The experimental studies have proved that the new extension is scalable and faster than the baseline algorithms.

1.3 Outline

This thesis is made up of six chapters organized as follows. In Chapter 2, we introduce the Big Data emergence in the last years and we present an overview of popular Big Data frameworks. We also conduct an experimental study on the reviewed Big Data frameworks.
In Chapter 3, we first survey large graph processing systems and graph programming models. Then, we discuss existing graph clustering algorithms and we present a structural graph clustering algorithms.
In Chapter 4, we introduce our method for structural graph clustering. Then, we present the conducted experimental study and we compare the proposed algorithm to four similar clustering algorithms.
In Chapter 5, we extend the proposed distributed algorithm in order to support dynamic graphs. For this, we purpose an incremental and distributed structural graph clustering algorithm that exploits the previous results to generate new ones in each update. Through the conducted experimental study, we have shown that the incremental algorithm significantly improves the execution time.
In Chapter 6, we conclude the present manuscript by summarizing our contributions and highlighting some prospects.

Part I

Background and related works

CHAPTER 2

Big Data Frameworks

Contents

2.1 Introduction to Big Data 11

2.2 Popular Big Data frameworks 11

2.2.1 Presentation of Big Data frameworks 13

2.2.2 Apache Hadoop 13

2.2.2.1 Hadoop system overview 13

2.2.2.2 WordCount example with Hadoop 17

2.2.3 Apache spark 18

2.2.3.1 Spark system overview 18

2.2.3.2 WordCount example with Spark 20

2.2.4 Apache Storm 20

2.2.4.1 Storm system overview 20

2.2.4.2 WordCount example with Storm 22

2.2.5 Apache Samza 23

2.2.5.1 Samza system overview 23

2.2.5.2 WordCount example with Samza 24

2.2.6 Apache Flink 25

2.2.6.1 Flink system overview 25

2.2.6.2 WordCount example with Flink 26

2.2.7 Categorization of Big Data frameworks 26

2.3 Comparative study of Big Data frameworks 29

2.3.1 Experimental environment 29
2.3.1.1 Experimental environment 29
2.3.2 Experimental results 33
2.3.2.1 Batch mode 33
2.3.2.2 Stream mode 44
2.3.2.3 Summary of the evaluation 44
2.4 Real-world applications and best practices 45
2.4.1 Real-world applications 45
2.4.1.1 Healthcare applications 45
2.4.1.2 Recommendation systems 46
2.4.1.3 Social media 47
2.4.1.4 Smart cities 47
2.4.2 Best practices 48
2.4.2.1 Stream processing 48
2.4.2.2 Micro-batch processing 49
2.4.2.3 Machine learning algorithms 49
2.4.2.4 Big graph processing 50
2.5 Conclusion 50

Goals The main objective of this chapter is to discuss the Big Data concept from both theoretical and practical perspectives. The comprehension of this concept can be considered as a primary stage before proceeding to a specialized study of its components like graphs. This background chapter starts with a brief introduction to Big Data. Afterward, we describe popular frameworks that are recently developed to handle imposed challenges by this concept. A deep experimental study of some Apache frameworks is conducted in order to concretize their performance to store, analyze and process Big Data. Finally, the usefulness of Big Data in real-world applications is then demonstrated through some examples
Keywords: Big Data, Spark, MapReduce, Hadoop, Stream processing.

2.1 Introduction to Big Data

In recent decades, increasingly large amounts of data are generated from a variety of sources. The size of generated data per day on the Internet has already exceeded two exabytes [Gandomi 2015b]. Within one minute, 72 h of videos are uploaded to Youtube, around 30.000 new posts are created on the Tumblr blog platform, more than 100.000 tweets are shared on Twitter and more than 200.000 pictures are posted on Facebook [Gandomi 2015b]. Big Data problems lead to several research questions such as (1) how to design scalable environments, (2) how to provide fault tolerance and (3) how to design efficient solutions.
Most existing tools for storage, processing and analysis of data are inadequate for massive volumes of heterogeneous data. Consequently, there is an urgent need for more advanced and adequate Big Data solutions. Many definitions of Big Data have been proposed throughout the literature. Most of them agreed that Big Data problems share four main characteristics, referred to as the four V's (Volume, Variety, Veracity and Velocity) [Oguntimilehin 2014].
The volume refers to the size of available datasets that typically require distributed storage and processing. The variety refers to the fact that Big Data is composed of several different types of data such as text, sound, image and video. The veracity refers to the biases, noise and abnormality in data. The velocity deals with the place at which data flows in from various sources like social networks, mobile devices and the Internet of Things (IoT).

2.2 Popular Big Data frameworks

In the literature, a set of surveys was established in relation to Big Data frameworks. From the ten discussed surveys above, only six have experimentally studied some of the Big Data frameworks.

In [6], the authors compared several MapReduce implementations like Hadoop [Li 2015], Twister [Ekanayake 2010] and LEMO-MR [Fadika 2010] on many workloads. Particularly, the performance and scalability of the studied frameworks have been evaluated. In [Veiga 2016], an experimental study on Spark, Hadoop and

Flink has been conducted. Mainly, the impact of some configuration parameters of the studied frameworks (e.g., number of mappers and reducers in Hadoop, number of threads in the case of Spark and Flink) on the runtime while running several workloads was studied. In [Shi 2015], the authors conducted an experimental study on Spark and Hadoop. They developed two profiling tools : (1) a study of the resource utilization for both MapReduce and Spark; (2) a breakdown of the task execution time for in-depth analysis. The conducted experiments showed that Spark is about 2.5x, 5x, and 5x faster than MapReduce, for WordCount, k-means, and PageRank workloads, respectively.
Some other works like [Singh 2014] [Chen 2014b] [Chen 2014a] [Liu 2014] tried to highlight Big Data fundamentals. They discussed the challenges related to Big Data applications and they presented the main features of some Big Data processing frameworks. Two works have compared Spark and Flink from theoretical and/or experimental point of view [García-Gil 2017] [Marcu 2016]. Scalability and impact of the size on disk, as well as the performance of specific functionalities of the compared frameworks have been considered.
In [García-Gil 2017], the authors discussed the main difference between Spark and Flink and presented an empirical study of both frameworks in the case of machine learning applications. In [Marcu 2016], Marcu et al. studied the impact of different architectural choices and parameter configurations on the perceived performance in the case of batch processing is studied. The performance of the studied frameworks has been evaluated with several representative batch and iterative workloads. The work presented in [Zhang 2015b] deals with in-memory Big Data management and processing frameworks. The authors provided a review of several in-memory data management and processing proposals and systems, including both data storage systems and data processing frameworks. They also presented some key factors that need to be considered in order to achieve efficient in-memory data management and processing, such as RDD for in-memory data persistence, immutable objects to improve response time, and data placement optimization. In [Zhang 2017], the authors conducted an experimental study on Storm and Flink in a stream processing context. The aim of the conducted study is to understand how current design aspects of modern stream processing sys-

tems interact with modern processors when running different types of applications. However, the study mainly focuses on evaluating the common design aspects of stream processing systems on scale-up architectures, rather than comparing the performance of individual systems.
We mention that most of the above-presented surveys are limited in terms of both the evaluated features of Big Data frameworks and the number of considered frameworks. For example, in [Zhang 2017], only stream processing frameworks are considered while in [Dede 2014] [Veiga 2016] [García-Gil 2017] [Marcu 2016], only batch processing frameworks are considered. We highlight that our experimental survey as described in Section 1.3. differs from the above-presented works by the fact that it compares the studied frameworks in the case of both batch and stream processing. It also deals with several representative batch and iterative workloads which are not considered in most existing surveys. Add to that, additional parameters (e.g., memory, threads) are configured to better evaluate the discussed frameworks. Moreover, monitoring capacities differentiate our work from the existing surveys. In fact, a personalized tool is implemented for different tests to effectively monitor resource usage.

2.2.1 Presentation of Big Data frameworks

This part is dedicated to a theoretical study of the most popular Big Data frameworks. For a better explanation of their principles, we use the Word- Count program as a running example. This latter consists on reading a set of text files and counting how often words occur. Snapshots of the codes used to implement the WordCount example with the studied frameworks are available online.

2.2.2 Apache Hadoop

2.2.2.1 Hadoop system overview

Hadoop is an Apache project founded in 2008 by Doug Cutting at Yahoo and Mike Cafarella at the University of Michigan [Polato 2014]. Hadoop consists of two main components: (1) Hadoop Distributed File System (HDFS) for data storage

and (2) Hadoop MapReduce, an implementation of the MapReduce programming model [Dean 2008]. In what follows, we discuss the MapReduce programming model, HDFS and Hadoop MapReduce.

- MapReduce Programming Model : it is a programming model that was designed to deal with parallel processing of large datasets. MapReduce has been proposed by Google in 2004 [Dean 2008] as an abstraction that allows performing simple computations while hiding the details of parallelization, distributed storage, load balancing and enabling fault tolerance. The central features of the MapReduce programming model are two functions, written by a user: Map and Reduce. The Map function takes a single key–value pair as input and produces a list of intermediate key–value pairs. The intermediate values associated with the same intermediate key are grouped together and passed to the Reduce function. The Reduce function takes as input an intermediate key and a set of values for that key. It merges these values together to form a smaller set of values. The system overview of MapReduce is illustrated in Fig. 2.1.

  As shown in Fig. 2.1, the basic steps of a MapReduce program are as follows:

  1. Data reading: in this phase, the input data is transformed into a set of key–value pairs. The input data may come from various sources such as file systems, database management systems or main memory (RAM). The input data is split into several fixed-size chunks. Each chunk is processed by one instance of the Map function.
  2. Map phase: for each chunk having the key–value structure, the corresponding Map function is triggered and produces a set of intermediate key–value pairs.
  3. Combine phase: this step aims to group together all intermediate key–value pairs associated with the same intermediate key.
  4. Partitioning phase: following their combination, the results are distributed across the different Reduce functions.

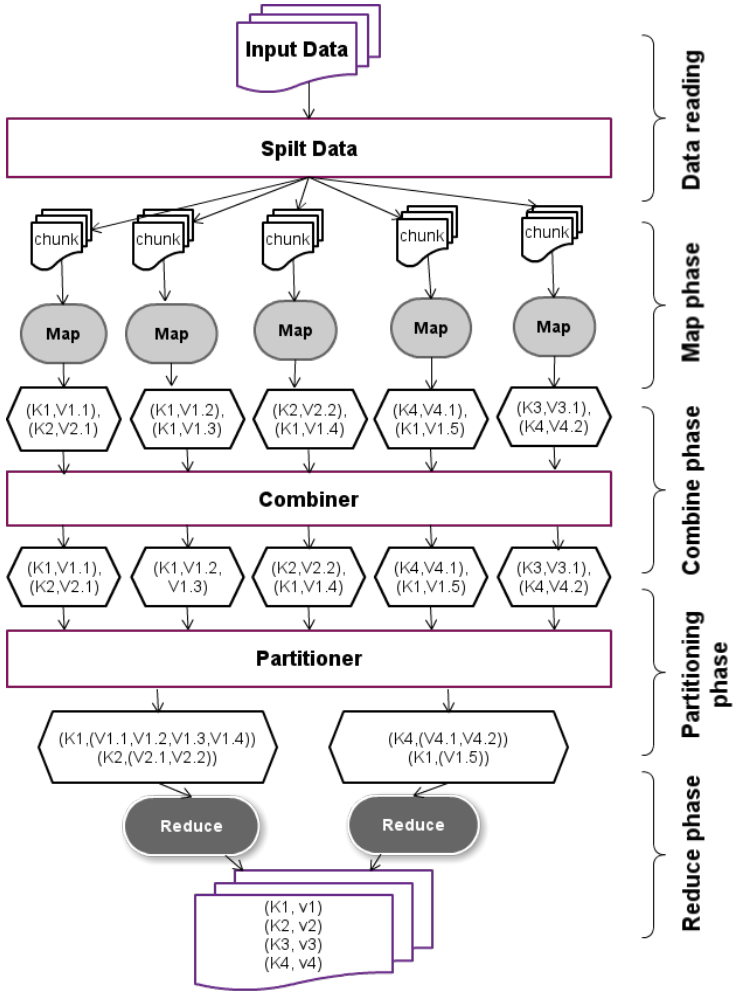


Figure 2.1: MapReduce architecture

5. Reduce phase: the Reduce function merges key–value pairs having the same key and computes a final result.

- **HDFS**: it consists of an open source implementation of the distributed Google File System (GFS) [Ghemawat 2003]. It provides a scalable distributed file system for storing large files over distributed machines in a reliable and efficient way [White 2012]. In Fig. 2.2, we show the abstract architecture of HDFS and its components.

  It consists of a master/slave architecture with a Name Node being master

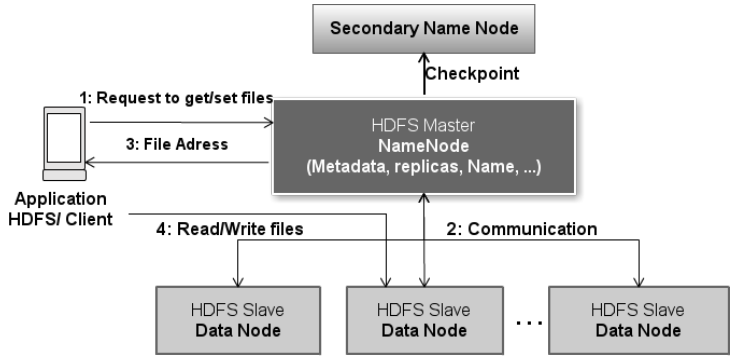


Figure 2.2: HDFS architecture

and several Data Nodes as slaves. The Name Node is responsible for allocating physical space to store large files sent by the HDFS client. If the client wants to retrieve data from HDFS, it sends a request to the Name Node. The Name Node will seek their location in its indexing system and subsequently sends their address back to the client. The Name Node returns to the HDFS client the meta-data (filename, file location, etc.) related to the stored files. A secondary Name Node periodically saves the state of the Name Node. If the Name Node fails, the secondary Name Node takes over automatically.

- **Hadoop MapReduce:** There are two main versions of Hadoop MapReduce. In the first version called MRv1, Hadoop MapReduce is essentially based on two components: (1) the Task Tracker that aims to supervise the execution of the Map/Reduce functions and (2) the Job Tracker which represents the master part and allows resource management and job scheduling/monitoring. The Job Tracker supervises and manages the Task Trackers [21]. In the second version of Hadoop called YARN, the two major features of the Job Tracker have been split into separate daemons: (1) a global Resource Manager and (2) per-application Application Master. In Fig. 2.2, we illustrate the overall architecture of YARN. As shown in this figure, the Resource Manager receives and runs MapReduce jobs. The per-application Application Master obtains resources from the Resource Manager and works

with the Node Manager(s) to execute and monitor the tasks. In YARN, the Resource Manager (respectively the Node Manager) replaces the Job Tracker (respectively the Task Tracker) [Li 2015]. Note that other well-known cluster managers are heavily used by Big Data systems. Taking as examples Mesos [Hindman 2011] and Zookeeper [Skeirik 2013].

- Mesos is an open source cluster manager that ensures dynamic resources sharing and provides efficient resources management for distributed frameworks [Hindman 2011]. It is based on a master/slave architecture. The master node relies on a daemon, called master process. This later manages all executor daemons deployed in the slave nodes, on which user tasks are distributed and executed.

- Apache ZooKeeper is an open source and fault-tolerant coordinator for large distributed systems [Skeirik 2013]. It provides a centralized service for maintaining the cluster's configuration and management. It also ensures the data or service synchronization in distributed applications. Unlike YARN or Mesos, Zookeeper is based on a cooperative control architecture, where the same service is deployed in all machines of the cluster. Each client or application can request the Zookeeper service by connecting to any machine in the cluster.

2.2.2.2 WordCount example with Hadoop

A WordCount program in Hadoop consists of a MapReduce job that counts the number of occurrences of each word in a file stored in the HDFS. The Map task maps the text data in the file and counts each word in the data chunk provided to the Map function (see Fig. 2.1). The result of the Map tasks are passed to Reduce function which combines and reduces the data to generate the final result.

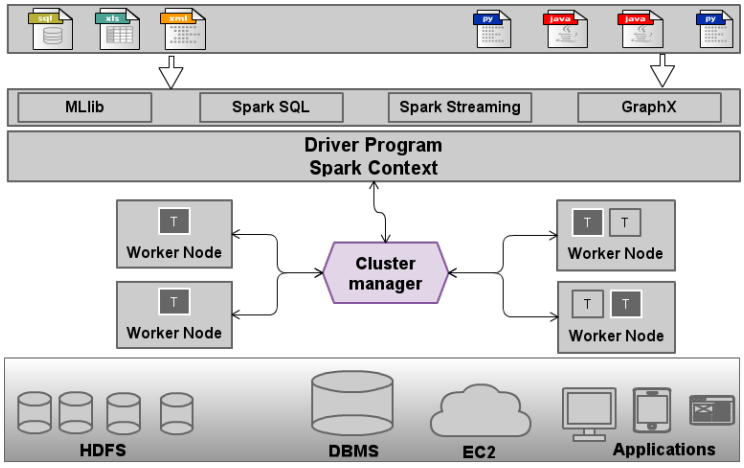


Figure 2.3: Spark system overview

2.2.3 Apache spark

2.2.3.1 Spark system overview

Apache Spark is a powerful processing framework that provides ease of use tool for efficient analytics of heterogeneous data. It was originally developed at UC Berkeley in 2009 [Zaharia 2010]. Spark has several advantages compared to other Big Data frameworks like Hadoop and storm. Spark is used by many companies such as Yahoo, Baidu, and Tencent. A key concept of Spark is Resilient Distributed Datasets (RDDs). An RDD is basically an immutable collection of objects spread across a Spark cluster. In Spark, there are two types of operations on RDDs: (1) transformations and (2) actions. Transformations consist in the creation of new RDDs from existing ones using functions like map, filter, union and join. Actions consist of the final result of RDD computations. In Fig. 2.3, we present an overview of the Spark architecture. A Spark cluster is based on a master/slave architecture with three main components:

- **Driver Program:** this component represents the slave node in a Spark cluster. It maintains an object called SparkContext that manages and supervises running applications.
- **Cluster Manager:** this component is responsible for orchestrating the

workflow of the application assigned by Driver Program to workers. It also controls and supervises all resources in the cluster and returns their state to the Driver Program.

- **Worker Nodes:** each Worker Node represents a container of one operation during the execution of a Spark program.

Spark offers several Application Programming Interfaces (APIs) [Zaharia 2010]:

- **SparkCore:** Spark Core is the underlying general execution engine for the Spark platform. All other features and extensions are built on top of it. Spark Core provides in-memory computing capabilities and a generalized execution model to support a wide variety of applications, as well as Java, Scala, and Python APIs for ease of development.

- **SparkStreaming:** Spark Streaming enables powerful interactive and analytic applications across both streaming and historical data while inheriting Spark's ease of use and fault tolerance characteristics. It can be used with a wide variety of popular data sources including HDFS, Flume [Chambers 2010], Kafka [Garg 2013], and Twitter [Zaharia 2010].

- **SparkSQL:** Spark offers a range of features to structure data retrieved from several sources. It allows subsequently to manipulate them using the SQL language [Armbrust 2015].

- **SparkMLLib:** Spark provides a scalable machine learning library that delivers both high-quality algorithms (e.g., multiple iterations to increase accuracy) and high speed (up to 100x faster than MapReduce) [Zaharia 2010].

- **GraphX:** GraphX [Xin 2013] is a Spark API for graph-parallel computation (e.g., PageRank algorithm and collaborative filtering). At a high-level, GraphX extends the Spark RDD abstraction by introducing the Resilient Distributed Property Graph: a directed multigraph with properties attached to each vertex and edge. To support graph computation, GraphX provides a set of fundamental operators (e.g., subgraph, joinVertices, and

MapReduceTriplets) as well as an optimized variant of the Pregel API [Malewicz 2010a]. In addition, GraphX includes a growing collection of graph algorithms (e.g., PageRank, Connected components, Label propagation and Triangle count) to simplify graph analytics tasks.

2.2.3.2 WordCount example with Spark

In Spark, every job is modeled as a graph. The nodes of the graph represent transformations and/or actions, whereas the edges represent data exchange between the nodes through RDD objects.

Through Fig. 2.4, we show the execution plan for a WordCount job. In the first step, the SparkContext object is used to read the input data from any sources (e.g., HDFS) and to create an RDD. In the second step, several operations can be applied to the RDD. In this example, we apply a flatMap operation that receives the lines of RDD, and applies a lambda function to each line of the RDD in order to generate a set of words. Then, a map function is applied in order to create a set of key–value pairs, in which the key is a word and the value is the number one. The next step consists of computing the sum of the values of each key using the reduceByKey function. The final results are written using the saveAsFile function.

2.2.4 Apache Storm

2.2.4.1 Storm system overview

Storm [Toshniwal 2014] is an open source framework for processing large structured and unstructured data in real-time. Storm is a fault tolerant framework that is suitable for real-time data analysis, machine learning, sequential and iterative computation. Following a comparative study of storm and Hadoop, we find that the first is geared for real-time applications while the second is effective for batch applications. As shown in Fig. 2.5, a storm program/topology is represented by a directed acyclic graph (DAG). The edges of the program DAG represent data transfer. The nodes of the DAG are divided into two types: spouts and bolts. The spouts (or entry points) of a storm program represent the data sources. The bolts

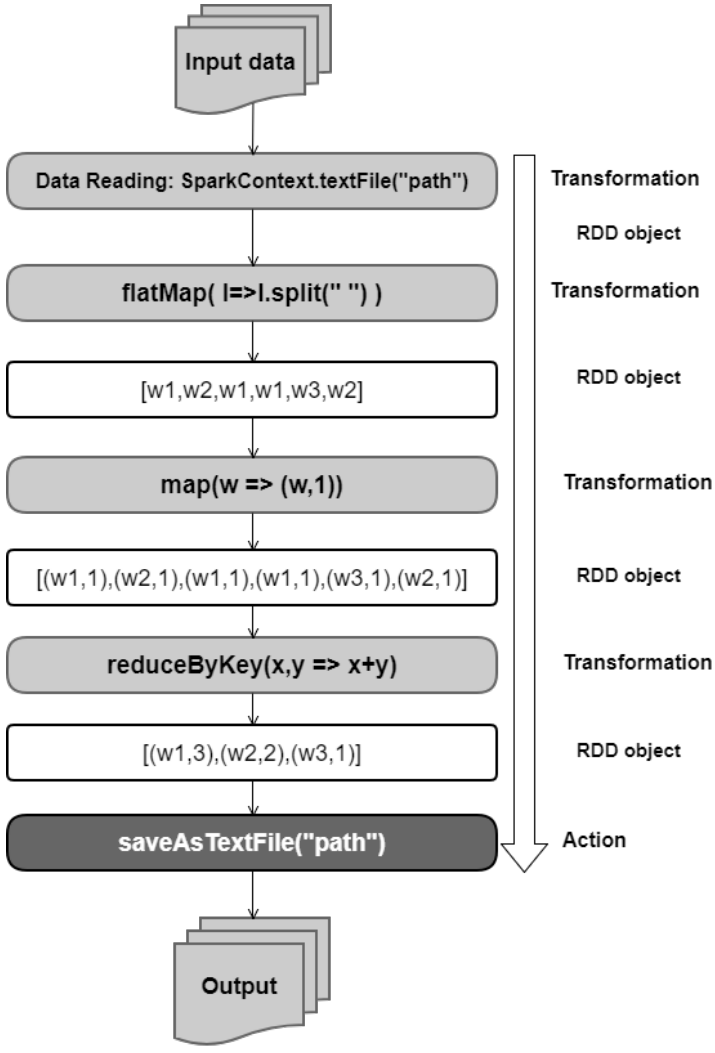


Figure 2.4: WordCount example with Spark

represent the functions to be performed on the data. Note that storm distributes bolts across multiple nodes to process the data in parallel. In Fig. 2.5, we show a storm cluster administrated by zookeeper, a service for coordinating processes of distributed applications [Hunt 2010]. Storm is based on two daemons called Nimbus (in master node) and supervisor (for each slave node). Nimbus supervises the slave nodes and assigns tasks to them. If it detects a node failure in the cluster, it re-assigns the task to another node. Each supervisor controls the execution of

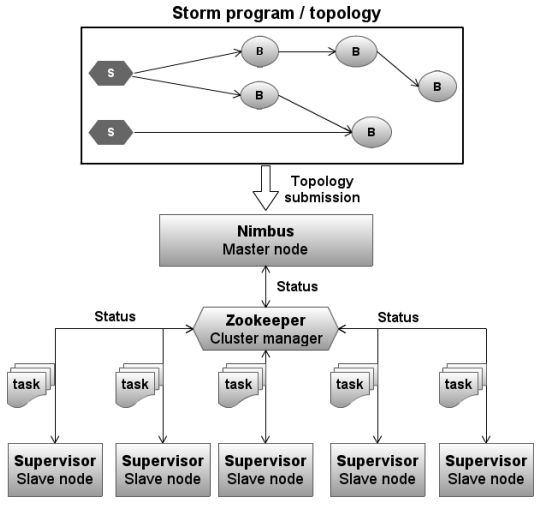


Figure 2.5: Topology of a Storm program and architecture

its tasks (affected by the nimbus). It can stop or start the spots following the instructions of Nimbus. Each topology submitted to Storm cluster is divided into several tasks.

2.2.4.2 WordCount example with Storm

Since Storm is a framework for stream processing, we run the WordCount example in stream mode. A Storm WordCount job consists of a topology that combines a set of spouts and bolts, where the spouts are used to get the data and the bolts are used to process the data. In Fig. 2.6, three processing layers are used to process the data. In the first layer, the spouts are used to read the input data from the sources and push the data (as lines of text) to the next layer. Then, in the next layer, a set of bolts are used to generate a set of words from each consumed line (from the previous layer). Finally, the last bolts are used to count for each word its number of occurrences.

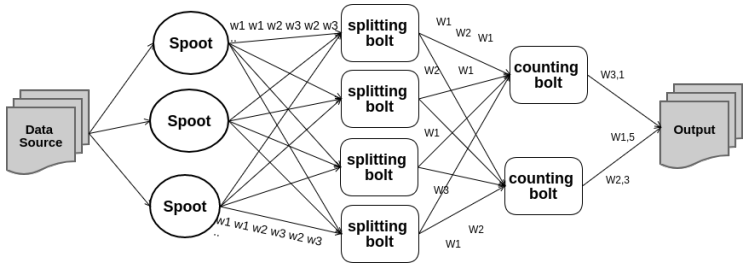


Figure 2.6: WordCount example with Storm

2.2.5 Apache Samza

2.2.5.1 Samza system overview

Apache Samza [Samza 2014] is a distributed processing framework created by LinkedIn to solve various kinds of stream processing requirements such as tracking data, service logging, and data ingestion pipelines for real-time services. Since then, it was adopted and deployed in several projects. Samza is designed to handle large messages and to provide file system persistence for them. It uses Apache Kafka as a distributed broker for messaging, and YARN for distributed resource allocation and scheduling. YARN resource manager is adopted by Samza to provide fault tolerance, processor isolation, security, and resource management in the used cluster. As illustrated in Fig. 2.7, Samza is based on three layers. The first layer is devoted to streaming data and uses Apache Kafka to manage the data flow. The second layer is based on YARN resource manager to handle the distributed execution of Samza jobs and to manage CPU and memory usage across a multi-tenantcluster of machines. The processing capabilities are available in the third layer which represents the Samza core and provides APIs for creating and running stream tasks in the cluster [Samza 2014]. In this layer, several abstract classes can be implemented by the user to perform specific processing tasks. These abstract classes could be implemented with MapReduce, in order to ensure a distributed processing.

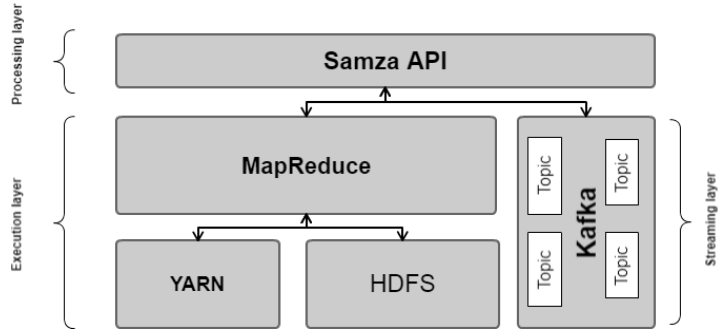


Figure 2.7: Samza architecture

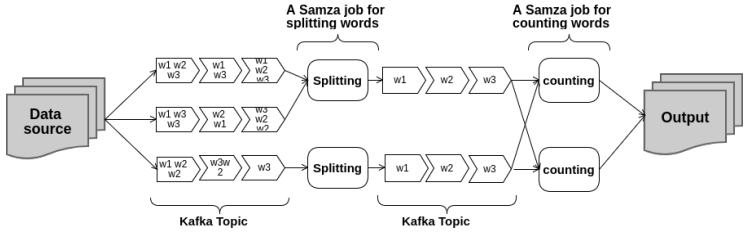


Figure 2.8: WordCount example with Samza

2.2.5.2 WordCount example with Samza

A Samza job is usually based on two parts. The first part is responsible for data processing and the second part is responsible for data flow transfer between the data processing units.

As shown in Fig 2.8. the execution steps of a wordCount job with Samza. In the first step, the data is read from the source and sent to the first Samza task, called a splitter, through a kafka topic. In this step, each message is split into a set of words. In the next step, another Samza task called counter consumes the set of words, and counts for each one the number of occurrences and generates the final result.

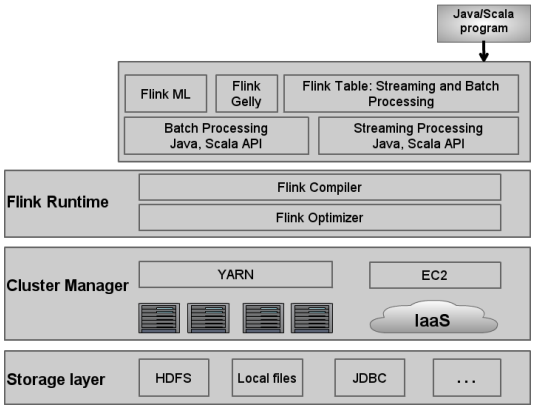


Figure 2.9: Flink architecture

2.2.6 Apache Flink

2.2.6.1 Flink system overview

Flink [Alexandrov 2014b] is an open source framework for processing data in both real time mode and batch mode. It provides several benefits such as fault-tolerant and large scale computation. The programming model of Flink is similar to MapReduce. By contrast to MapReduce, Flink offers additional high level functions such as join, filter and aggregation. Flink allows iterative processing and real time computation on stream data collected by different tools such as Flume [Chambers 2010] and Kafka [Garg 2013]. It offers several APIs on a more abstract level allowing the user to launch distributed computation in a transparent and easy way. Flink ML is a machine learning library that provides a wide range of learning algorithms to create fast and scalable Big Data applications. In Fig. 2.9, we illustrate the architecture and components of Flink. As shown in Fig. 2.9, the Flink system consists of several layers. In the highest layer, users can submit their programs written in Java or Scala. User programs are then converted by the Flink compiler to DAGs. Each submitted job is represented by a graph. Nodes of the graph represent operations (e.g., map, reduce, join or filter) that will be applied to process the data. Edges of the graph represent the flow of data between the operations. A DAG produced by the Flink compiler is received by the Flink optimizer in order to improve performance by optimizing the DAG (e.g., re-ordering of

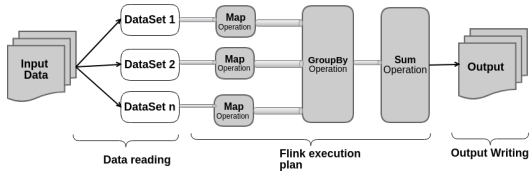


Figure 2.10: WordCount with Flink

the operations). The second layer of Flink is the cluster manager which is responsible for planning tasks, monitoring the status of jobs and resource management. The lowest layer is the storage layer that ensures storage of the data to multiple destinations such as HDFS and local files.

2.2.6.2 WordCount example with Flink

In order to implement the WordCount example with Flink, we can use the abstract functions provided by Flink such as map, flatMap and groupBy. First, the input data is read from the data source and stored in several dataset objects. Then, a map operation is applied to the dataset objects in order to generate key– value pairs, with the word as a key and one as value. Then, the groupBy function is applied to aggregate the list of key–value pairs generated in the previous step (Fig. 2.10). Finally, the number of occurrences of each word is calculated using the sum function and the final results are generated.

2.2.7 Categorization of Big Data frameworks

Before proceeding to the experimental survey, we start by presenting some popular Big Data frameworks and categorizing them according to their key features. These key features are (1) the programming model, (2) the supported programming languages, (3) the type of data sources and (4) the capability to allow for iterative data processing, (5) the compatibility of the framework with existing machine learning libraries, and (6) the fault tolerance strategy. We present in Table 2.1 a comparative study of the presented frameworks according to these features. As shown in Table 2.1, Hadoop, Flink and Storm use the key–value format to represent their data. This is motivated by the fact that the key–value format allows access to

	Hadoop	Spark	Storm	Flink	Samza
Data format	Key-value	Key-value, RDD	Key-value	Key-value	Events
Processing mode	Batch	Batch and Stream	Stream	Batch and Stream	Stream
Data sources	HDFS	HDFS, DBMS and Kafka	HDFS, HBase and Kafka	Kafka, Kinesis, message queus, socket streams and files	Kafka
Programming model	Map and Reduce	Transformation and Action	Topology	Transformation	Map and Reduce
Supported programming languages	Java	Java, Scala and Python	Java	Java	Java
Cluster manager	YARN	Standalone, YARN and Mesos	YARN or Zookeeper	Zookeeper	YARN
Comments	Stores large data in HDFS	Gives several APIs to develop interactive applications	Suitable for real-time applications	Flink is an extension of MapReduce with graph methods	Based on Hadoop and Kafka
Iterative computation	Yes (by running multiple MapReduce jobs)	Yes	Yes	Yes	Yes
Interactive Mode	No	Yes	No	No	No
Machine learning compatibility	Mahout	SparkMLlib	Compatible with SAMOA API	FlinkML	Compatible with SAMOA API
Fault tolerance	Duplication feature	Recovery technique on the RDD objects	Checkpoints	Checkpoints	Data partitioning

Table 2.1: Comparative study of popular Big Data frameworks

heterogeneous data. For Spark, both RDD and key–value models are used to allow fast data access. We have also classified the studied Big Data frameworks into two categories: (1) batch mode and (2) stream mode. We have shown in Table 2.1 that Hadoop processes the data in batch mode, whereas the other frameworks allow the stream processing mode. In terms of physical architecture, we notice that all the studied frameworks are deployed in a cluster architecture, and each framework uses a specified cluster manager. We note that most of the studied

frameworks use YARN as cluster manager. From a technical point of view, we mention that all the presented frameworks provide APIs for several programming languages like Java, Scala and Python.

Each framework provides a set of abstract functions that is used to define the desired computation. We also presented in Table 2.1 weather the studied framework provides a machine learning library or not. We notice that Spark and Flink provide their own machine learning libraries, while the other frameworks have some compatibility with other tools, such as SAMOA for Samza and Mahout for Hadoop. It is important to mention that Hadoop is currently one of the most widely used parallel processing solutions. Hadoop ecosystem consists of a set of tools such as Flume, HBase, Hive and Mahout. Hadoop is widely adopted in the management of large-size clusters. Its YARN daemon makes it a suitable choice to configure Big Data solutions on several nodes [Yao 2014]. For instance, Hadoop is used by Yahoo to manage 24 thousands of nodes. Moreover, Hadoop MapReduce was proven to be the best choice to deal with text processing tasks [35]. We notice that Hadoop can run multiple MapReduce jobs to support iterative computing but it does not perform well because it cannot cache intermediate data in memory for faster performance.

As shown in Table 2.1, Spark importance lies in its in-memory features and micro-batch processing capabilities, especially in iterative and incremental processing [Bajaber 2016]. In addition, Spark offers an interactive tool called SparkShell which allows exploiting the Spark cluster in real time. Once interactive applications were created, they may subsequently be executed interactively in the cluster. We notice that Spark is known to be very fast in some kinds of applications due to the concept of RDD and also to the DAG-based programming model.

Flink shares similarities and characteristics with Spark. It offers good processing performance when dealing with complex Big Data structures such as graphs. Although there exist other solutions for large-scale graph processing, Flink and Spark are enriched with specific APIs and tools for machine learning, predictive analysis and graph stream analysis [Alexandrov 2014b] [Zaharia 2010].

2.3 Comparative study of Big Data frameworks

We have performed an extensive set of experiments to highlight the strengths and weaknesses of popular Big Data frameworks. The performed analysis covers the scalability, the impact of several configuration parameters on the performance and the resource usage. For our tests, we evaluated Spark, Hadoop, Flink, Samza and Storm. For reproducibility reasons, we provide information about the implementation details and the used datasets online. In the following sub-section, we describe the experimental setup and we discuss the obtained results.

2.3.1 Experimental environment

All the experiments were performed in a cluster of 10 machines operating with Linux Ubuntu 16.04. Each machine is equipped with a 4 CPU, 8 GB of main memory and 500 GB of local storage. For our tests, we used Hadoop 2.9.0, Flink 1.3.2, Spark 1.6.0, Samza 0.10.3 and Storm 1.1.1. All the studied frameworks have been deployed with YARN as a cluster manager. We also varied these parameters in order to analyze the impact of some of them on the performance of the studied frameworks.

2.3.1.1 Experimental environment

We consider two scenarios according to the data processing mode (Batch and Stream) of the evaluated frameworks.

- In the Batch mode scenario, we evaluate Hadoop, Spark and Flink while running the WordCount, K-means and PageRank workloads with real and synthetic data sets.

  In the WordCount application, we used tweets that are collected by Apache Flume [Chambers 2010] and stored in HDFS. As shown in Fig2.11, the collected data may come from different sources including social networks, local files, log files and sensors. In our case, Twitter is the main source of our

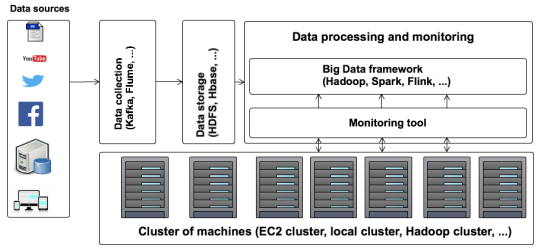


Figure 2.11: Batch Mode scenario

collected data. The motivation behind using Apache Flume to collect the processed tweets is its integration facility in the Hadoop ecosystem (especially the HDFS system). Moreover, Apache Flume allows data collection in a distributed way and offers high data availability and fault tolerance. We collected 10 billion tweets and we used them to form large tweet files with a size on disk varying from 250 MB to 100 GB of data.

For K-means, we generated a synthetic datasets containing between 10,000 and 100 million learning examples. As for the PageRank workload, we have used seven real graph datasets with different numbers of nodes and edges. Table 2.2 shows more details of the used datasets.

The above-presented datasets have been downloaded from the Stanford Large Network Dataset Collection (SNAP) [1] and formatted as plan files in which each line represents a link between two nodes. We implemented the PageRank workload with Hadoop using a three-jobs workflow. In the first job, we read data from the text file and we generated a set of links for each page. The second job is responsible for setting an initial score for each page. The last job iteratively computes and sorts the pages' scores. Regarding the PageRank implementation with Spark, we followed the same execution logic as in Hadoop. We implemented a Spark job that applies the *flatMap* function to generate key–value pairs for the corresponding links, and the *map* function to initialize an initial score for each page. Finally, the reduceByKey function is used to iteratively aggregate the page's scores. As

[1] https://snap.stanford.edu/data/

Dataset	Number of nodes	Number of edges	Description
G1	685 230	7 600 595	Web graph of Berkeley and Stanford
G2	875 713	5 105 039	Web graph from Google
G3	325 729	1 497 134	Web graph of Notre Dame
G4	281 903	2 312 497	Web graph of Stanford
G5	1,965,206	2,766,607	RoadNet-CA
G6	3,997,962	34,681,189	Com-LiveJournal
G7	4,847,571	8,993,773	Soc-LiveJournal

Table 2.2: Graph datasets.

for the implemented Flink job, it starts by generating the page-score pairs using the flatMap function. Then, it iteratively aggregates the scores for each page using the *groupBy* function. Finally, it computes the total score of each page by applying the sum function.

- In the Stream mode scenario, we evaluate real-time data processing capabilities of Storm, Flink, Samza and Spark. The Stream mode scenario is divided into three main steps. As shown in Fig. 2.12, the first step is devoted to data storage. To do this step, we collected 1 billion tweets from Twitter using Flume and we stored them in HDFS. The stored data is then transferred to Kafka, a messaging server that guarantees fault tolerance during the streaming and message persistence [Garg 2013]. The second step consists of sending the tweets as streams to the studied frameworks. To allow simultaneous streaming of the data collected from HDFS by Storm, Spark, Samza and Flink, we have implemented a script that accesses the HDFS and transfers the data to Kafka. The last step consists of executing our workloads in stream mode. To do this, we have implemented an Extract, Transform and Load (ETL) program in order to process the received messages from Kafka. The ETL routine consists of retrieving one tweet in its original format (JSON file), and selecting a subset of attributes from

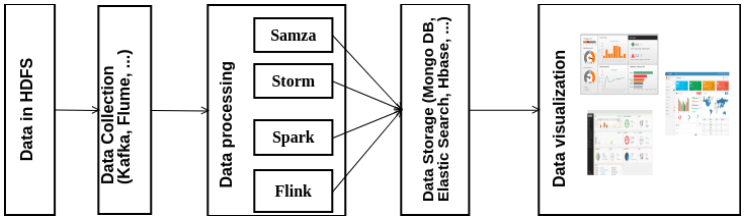


Figure 2.12: Stream mode scenario

the tweet such as hash-tag, text, geocoordinate, number of followers, name, surname and identifiers. All the received messages are processed by our implemented workload. Then, they are stored using ElasticSearch storage server, and possibly visualized with Kibana [Gupta 2015]. Regarding the hardware configuration adopted in the Stream mode, we used one machine for Kafka and one machine for Zookeeper that allows the coordination between Kafka and Storm. For the processing task, the remaining machines are devoted to access the data in HDFS and to send it to Kafka server.

To allow monitoring resources usage according to the executed jobs, we have implemented a personalized monitoring tool as shown in Fig. 2.13. Our monitoring solution is based on three core components : (1) data collection module, (2) data storage module, and (3) data visualization module. To detect the states of the machines, we have implemented a Python script and we deployed it in every machine of the cluster. This script is responsible for collecting CPU, RAM, Disk I/O, and Bandwidth history.

The collected data are stored in ElasticSearch, in order to be used in the evaluation step. The stored data are used by Kibana for monitoring and visualization purposes. For our monitoring tests, we used a dataset of 50 GB of data for the WordCount workload, 10 million examples for the K-means workload and the G5 dataset for the PageRankworkload. It is important to mention that existing monitoring tools like Ambari [Wadkar 2014] and Hue [Eadline 2015] are not suitable for our case, as they only offer real-time monitoring results.

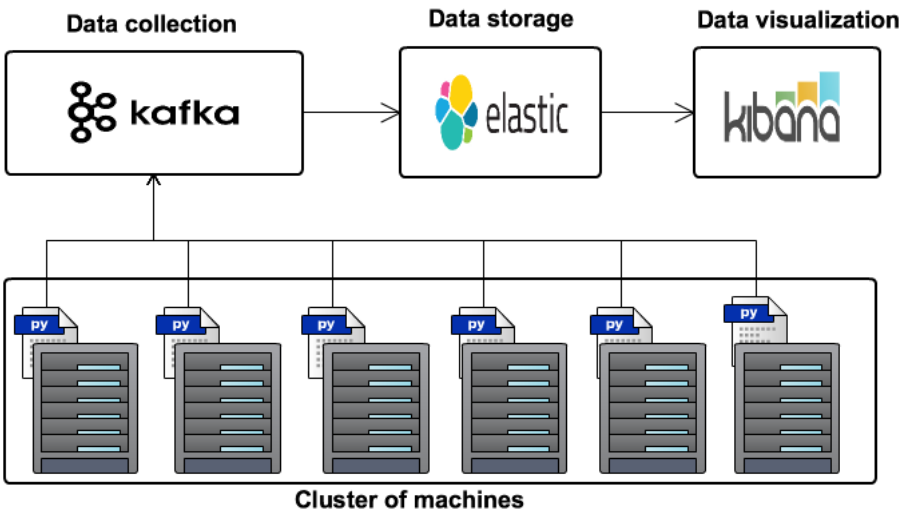


Figure 2.13: Architecture of our personalized monitoring tool

2.3.2 Experimental results

2.3.2.1 Batch mode

In this sub-section, we evaluate the scalability of the studied frameworks, and we measure their CPU, RAM, disk I/O usage, as well as bandwidth consumption while processing. We also study the impact of several parameters and settings on the performance of the evaluated frameworks. Evaluation of the horizontal scalability This experiment aims to evaluate the impact of the size of the data on the processing time. In this experiment, we used two simulations according to the size of data: (1) simulation with small datasets and (2) simulation with big datasets. Our experiments are conducted using the WordCount workload, with various datasets with a size on disk varying from 250 MB to 2 GB for the first simulation and from 1 GB to 100 GB for the second simulation. Fig. 2.14 and Fig. 2.15 show the average processing time for each framework and for every dataset. As shown in 2.14, Spark is the fastest framework for all the datasets, Flink is the next and Hadoop is the lowest. Fig. 2.15 shows that Spark has kept its order in the case of big datasets and Hadoop showed good results compared to Flink. We also notice that Flink is faster than Hadoop only in the case of very

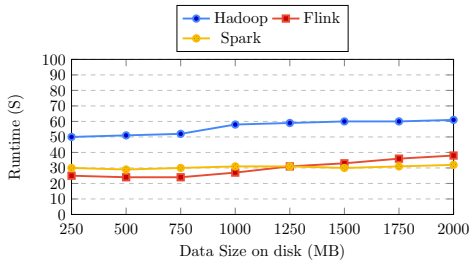


Figure 2.14: Impact of the size of the data on the average processing time: case of small datasets

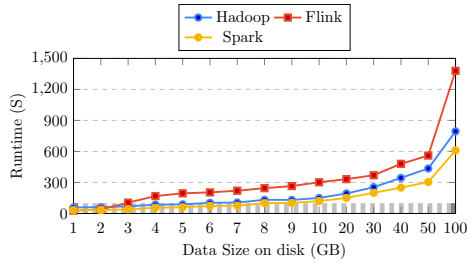


Figure 2.15: Impact of the size of the data on the average processing time: case of big datasets

small datasets. Compared to Spark, Hadoop achieves data transfer by accessing the HDFS. Hence, the processing time of Hadoop is considerably affected by the high amount of Input/Output (I/O) operations. By avoiding I/O operations, Spark has gradually reduced the processing time. It can also be observed that the computational time of Flink is longer than those of Spark and Hadoop in the case of big datasets. This is due to the fact that Flink sends its intermediate results directly to the network through channels between the workers, which makes the processing time very dependent on the cluster's local network. In the case of small datasets, the data is transmitted quickly between workers. As shown in Fig. 2.14, Flink is faster than Hadoop. We also notice that Spark defines optimal time by using the memory to store the intermediate results as RDD objects.

In the next experiment, we tried to evaluate the scalability and the processing time of the considered frameworks based on the size of used cluster (the number of machines in the cluster). Fig. 2.16 shows the impact of the number of the

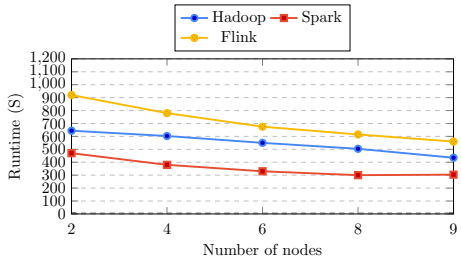


Figure 2.16: Impact of the number of machines on the average processing time (WordCount workload with 50 Gb of data)

used machines on the processing time. Both Hadoop and Flink take higher time regardless of the cluster size, compared to Spark. Fig. 2.16 shows instability in the slope of Flink due to the network traffic. In fact, Flink jobs are modeled as a graph that is distributed on the cluster, where nodes represent Map and Reduce functions, whereas edges denote data flow between Map and Reduce functions. In this case, Flink performance depends on the network state that may affect intermediate results which are transferred from Map to Reduce functions across the cluster. Regarding Hadoop, it is clear that the processing time is proportional to the cluster size. In contrast to the reduced number of machines, the gap between Spark and Hadoop is reduced when the size of the cluster is large. This means that Hadoop performs well and can have close processing time in the case of bigger cluster size. The time spent by Spark is approximately between 450 s and 300 s for 2–6 nodes cluster. Furthermore, as the number of participating nodes increases, the processing time, yet, remains approximately equal to 290 s. This is explained by the processing logic of Spark. Indeed, Spark depends on the main memory (RAM) and the available resources in the cluster. In case of insufficient resources to process the intermediate results, Spark requires more RAM to store its intermediate results. This is the case of 6 to 9 nodes which explains the inability to improve the processing time even with an increased number of participating machines.

In the case of Hadoop, intermediate results are stored on disk. This explains the reduced execution time that reached 400 s in the case of 9 nodes, compared to 600 s when exploiting only 4 nodes. To conclude, we mention that Flink

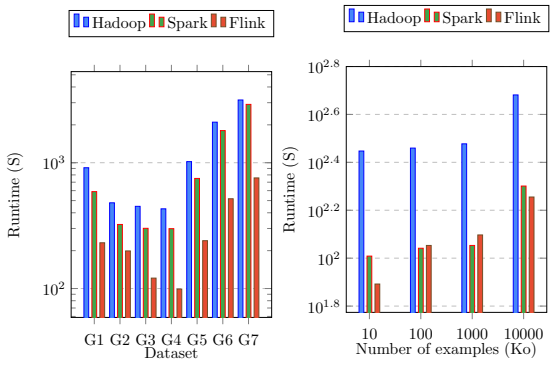


Figure 2.17: Impact of iterative processing on the average processing time

allows creating a set of channels between workers, to transfer intermediate results between them. Flink does not perform Read/Write operations on disk or RAM, which allows accelerating the processing times, especially when the number of workers in the cluster increases. As for the other frameworks, the execution of jobs is influenced by the number of processors and the amount of Read/Write operations, on disk (case of Hadoop) and on RAM (case of Spark).

Iterative processing

In the next scenario, we tried to evaluate the studied frameworks in the case of iterative processing with both K-means and PageRank workloads. In Fig.2.17, both use cases measure the impact of iterative computing on the studied frameworks. For K-means workload, we find that Spark and Flink are similar in response time and they are faster compared to Hadoop. This can be explained by the fact that Hadoop writes the output results, for each iteration in the hard disk which makes Hadoop very slow. The PageRank workload is an iterative processing but it consumes more memory resources, which degrade performances in the case of Spark. In this case, Spark consumed all the available memory to create a new RDD object. Then, Spark applies its own strategy to replace the useless RDD and, when it does not fit in the memory, it slow down the execution compared to both Flink and Hadoop. Through the next experiment, we try to show the impact of the number of iterations on the runtime. We tested our frameworks by running K-means on 10 million examples in the training set. We varied the number of

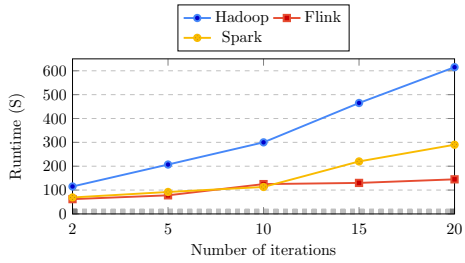


Figure 2.18: Impact of the number of iterations on the average processing time (Kmeans workload with 10 million examples)

iteration in each simulation. As shown in Fig. 2.18, with both Flink and Spark frameworks, the number of iterations has no significant influence on the execution time. Fig. 2.18. Impact of the number of iterations on the average processing time (K means workload with 10 million examples). One can conclude that the curve of Hadoop is characterized by an quadratic slope whereas in the case of Spark and Flink the curve are characterized by a linear slope. According to Fig. 2.18 and to the literature, we can interpret that Hadoop is not the best choice for this kind of processing (iterative computing).

Data partitioning

In the next experiment, we try to show the impact of data partitioning on the studied frameworks. In our experimental setup, we used HDFS for storage. We varied the block size in our HDFS system and we run K-means with 10 iterations with all the used frameworks. Fig. 2.19 presents the impact of the HDFS block size on the processing time. As shown in Fig. 2.19, the curves are inflated proportionally to the size of the HDFS block size for both Hadoop and Spark, while Flink does not imply any variation in the processing time. This can be explained by the degree of parallelism adopted by the studied frameworks. We mention that in Hadoop, the number of mappers is directly proportional to the input splits, which depends on HDFS block size. When we increase the number of splits, the degree of parallelism increases too. One possible solution to improve the processing time is to enhance resource usage, but this is not always possible according to the Hadoop curve's behavior presented in Fig. 2.19. Note also that when we set a block of HDFS whose size is less than 16 MB, the processing time

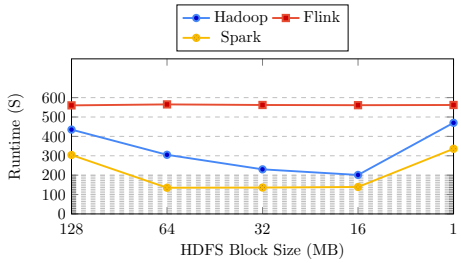


Figure 2.19: Impact of HDFS block size on the runtime (Kmeans workload with 10 million examples and 10 iterations)

decreases as the number of input splits exceeds the number of cores in the cluster. For Spark, we have almost the same results compared to Hadoop. Precisely, when Spark loads its data from HDFS, it converts or creates for each input split an RDD partition. In this case, the partition makes and provides the degree of parallelism, because Spark context program assigns for each worker an RDD partition.

As for Flink, each job is modeled as a directed graph, where nodes are reserved for data processing and edges represent data flow. In addition, each Flink job reserves a list of nodes in the graph to read the input data and to write the final results. In this case, the and send the data as stream flow to other processing nodes. This mechanism makes Flink independent on the HDFS block size, as shown in Fig. 2.19.

Impact of the cluster manager

In our work, we mainly used YARN as a cluster manager. We also tried to evaluate the impact of the cluster manager on the performance of the studied frameworks. To do this, we compared Mesos, YARN and the standalone clusters manager of the studied frameworks. For our tests, we run the WordCount workload with 50 GB of data, K-means with 10 million examples and Pagerank with the G5 dataset (see Table 2.1). As shown in Fig. 2.20, the standalone mode is faster than both YARN and MESOS. In fact, the standalone uses all the resources while executing a job, whereas both YARN and MESOS have a scheduler to run multiple jobs at once and share the cluster resources with all the submitted applications [Jha 2014].

Impact of bandwidth

In order to study the impact of bandwidth consumption on the performance of the

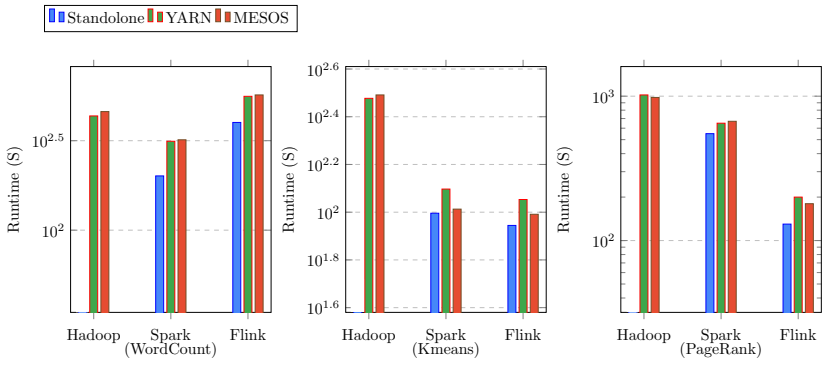


Figure 2.20: Impact of the cluster manager on the performance of the studied frameworks

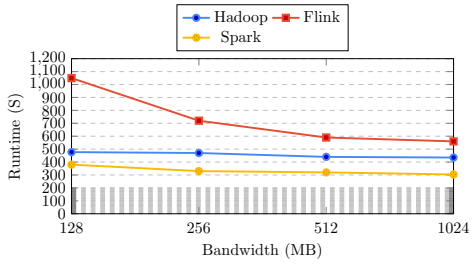


Figure 2.21: Impact of bandwidth on the performance of the studied frameworks

studied frameworks, we run the Word- Count workload with 50 GB of data and we varied the bandwidth from 128 MB to 1 GB. As shown in Fig. 2.21, Flink is bandwidth dependent.

In fact, when the bandwidth increases, the response time decreases. This can be explained by the fact that each Flink job sends the data directly from a source to a calculating unit across the network. We also notice that Spark uses the network to, sometimes, migrate the data to the processing unit. Hadoop allows data locality, which means that Hadoop moves the computation close to where the actual data resides on the node.

Impact of some configuration parameters

All the studied frameworks have a large list of configuration parameters, which can influence their behaviors. In order to understand the impact of these parameters on the performance and the quality of the results, we try in this

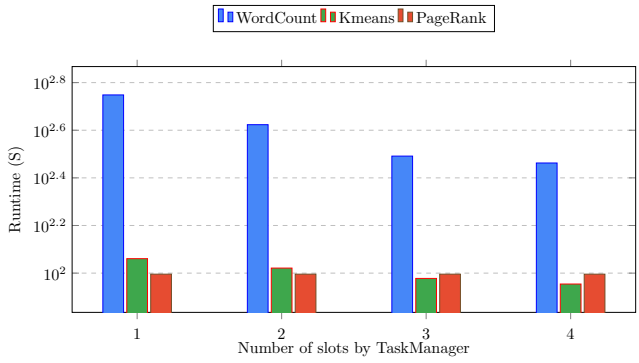


Figure 2.22: Impact of parallelism parameters on the performance of Flink

sub-section to study some of them mainly those which are related to the RAM and the number of threads in each framework. In Hadoop, the Application Manager daemon distributes the Map and Reduce functions on the available slots of the cluster. To configure this aspect, we set both parameters *mapred.tasktracker.map.tasks.maximum* and *mapred.tasktracker.reduce.tasks.maximum* in the *site-mapred.xml* configuration file. These parameters represent respectively the maximum number of Map and Reduce tasks that will run simultaneously on a node. Note that Spark uses executor-cores parameter and Flink uses slots parameter to configure the number of executed threads in parallel. In order to define the amount of memory buffer, Hadoop uses the *io.sort.mb* parameter in the *site-mapred.xml* configuration, Spark uses the *executor-memory* parameter and Flink uses the taskManagerMemory parameter.

- **Flink configuration:** In order to evaluate the impact of some configuration parameters on the performance of Flink, we first executed our workloads while varying the number of slots in each TaskManager. Then, we varied the amount of used memory by each TaskManager.

  Fig. 2.22 presents the impact of the number of slots on the processing time of the WordCount workload. In fact, the latter is characterized by a high CPU resource consumption which explains the reduction of the response time when the number of slots increases. Note that this is not the case with K-means and PageRank workloads. By analyzing Fig. 2.23, we notice that

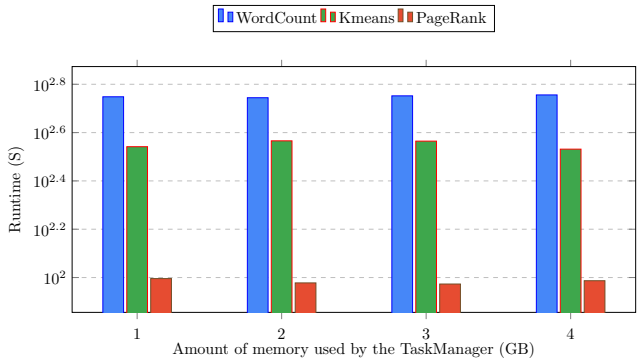


Figure 2.23: Impact of the memory size on the performance of Flink

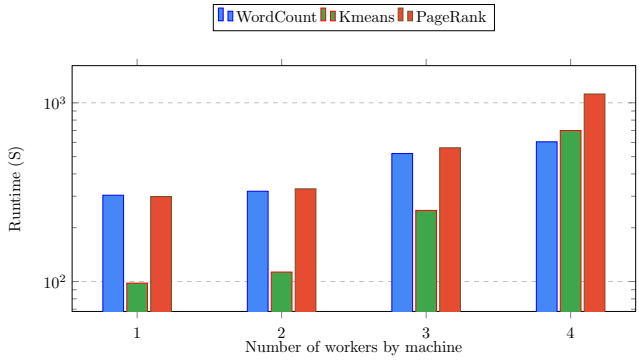


Figure 2.24: Impact of the number of workers on the performance of Spark

the memory resource does not have a large effect on the processing time in these workloads since Flink is based on sending the output results directly from one computing unit to another one without a high usage of disk or memory.

- **Spark Configuration:** The parallelism configuration in Spark requires the definition of the number of executor-cores by machine. In addition, memory management is primordial as we must configure the memory for each worker. These two parameters are respectively executor-cores and executor-memory.

  As shown in Fig. 2.24, when we increase the number of workers per machine the processing time increases too. This behavior may be related to the

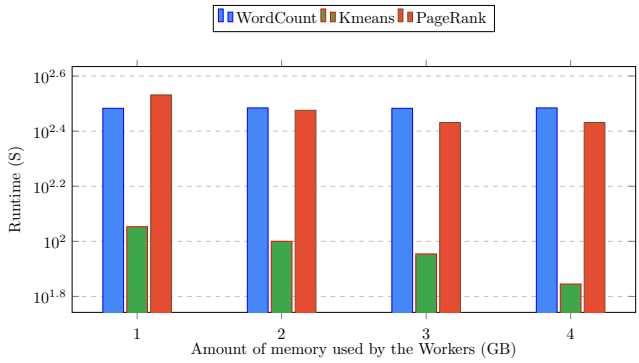


Figure 2.25: Impact of the memory size on the performance of Spark

memory management of Spark. In fact, when the memory is shared and distributed on several slots, the slot of each worker will be limited which slows down the computing performance. In this case, it is advisable to limit the number of workers, if the machine has a limited memory, and to maximize it proportionally to the capacity of the memory. In the same context, we notice the importance of the memory size through Fig. 2.25. When we increase the memory the response time decreases. This behavior is not always valid because it depends on some other constraints such as the availability of other resources traffic networks.

- **Hadoop Configuration:** To configure the number of slots on each node in a Hadoop cluster, we must set the two following parameters : (1) m*apreduce.tasktracker.map.tasks.maximum* and (2) *mapreduce. tasktracker.reduce.tasks.maximum*. These two parameters define the number of Map and Reduce functions that run simultaneously on each node of the cluster. These parameters maximize the CPU usage which can improve the processing time. Fig. 2.26 shows the impact of the number of slots on the performance of Hadoop jobs. We find that the best performance is guaranteed when using two slots for both Map and Reduce functions. However, this value depends on the number of cores in each node of the cluster.

  In our case, we have four cores in each machine and the best value is two slots for Map and Reduce, since the other cores are reserved for both

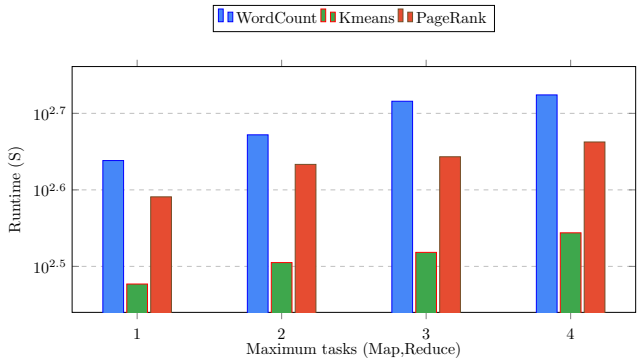


Figure 2.26: Impact of the number of slots on the performance of Hadoop

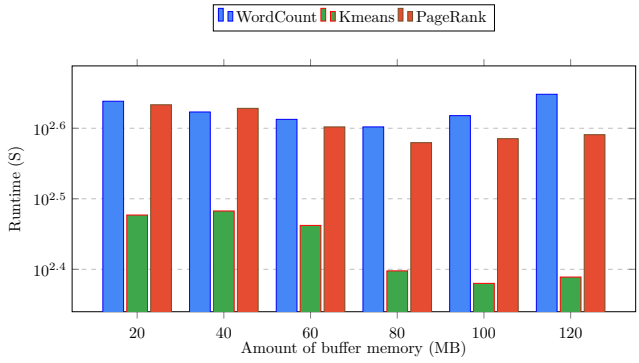


Figure 2.27: Impact of the memory size on the performance of Hadoop

daemons DataNode and NodeManager. The same behavior is observed with the WordCount workload because this latter is based on CPU resource compared to the other workloads. Among the characteristics of Hadoop, we note the use of the hard disk to write intermediate results between iterations or between Map and Reduce functions. Before writing data to the disk, Hadoop writes its intermediate data in a memory buffer. This memory can be configured through the *io.sort.mb* parameter. In order to evaluate the impact of this parameter, we varied its values from 20MB to 120MB as illustrated in Fig. 2.27. It is also clear that the processing time decreases subsequently and reaches 100 MB when we increase the value of *io.sort.mb* parameter. A level of stability is achieved when the satisfaction of the computing units by this resource is guaranteed.

2.3.2.2 Stream mode

In stream experiments, we measure CPU, RAM, disk I/O usage and bandwidth consumption of the studied frameworks while processing tweets, as described in Section 4.2. The goal here is to compare the performance of the studied frameworks according to the number of processed messages within a period of time. In the first experiment, we send a tweet of 100 KB (on average) per message. Fig. 33 shows that Flink, Samza and Storm have better processing rates compared to Spark. This can be explained by the fact that the studied frameworks use different values of window time. The values of window time of Flink, Samza and Storm are much smaller than that of Spark (milliseconds vs seconds).

In the next experiment, we changed the sizes of the processed messages. We used 5 tweets per message (around 500 KB per message). The results presented in Fig. 34 show that Samza and Flink are very efficient compared to Spark, especially for large messages.

2.3.2.3 Summary of the evaluation

From the above-presented experiments, it is clear that Spark can deal with large data sets better than Hadoop and Flink. Although Spark is known to be the fastest framework due to the concept of RDD, it is not a suitable choice in the case of intensive memory processing tasks. Indeed, intensive memory applications are characterized by the massive use of memory (creation of RDD objects at each transformation operation). This process degrades the performance of Spark since the SparkContext will be led to find the unused RDD and remove them in order to get more free memory space. The carried experiments in this work also indicate that Hadoop performs well on the whole. However, it has some limitations regarding the writing of intermediate results in the hard disk and requires a considerable processing time when the size of data increases, especially in the case of iterative applications. According to the resource consumption results in batch mode, we can conclude that Flink maximizes the use of CPU resources compared to both frameworks Spark and Hadoop. This good exploitation is relative to the pipeline technique of Flink which minimizes the period of idle resources. However,

it is characterized by high demands on the network resource compared to Hadoop. In fact, this resource consumption explains why Flink is faster than Hadoop. In the stream scenario, Flink, Samza and Storm are quite similar in terms of data processing. In fact, they are originally designed for stream processing. We also notice that Flink is characterized by its low latency since it is based on pipe-lined processing and on message passing processing technique, whereas Spark is based on Java Virtual Machine (JVM) and belongs to the category of batch mode frameworks. Each Samza job is divided into one or more partitions and each partition is processed in an independently container or executor, which shows best results with large stream messages. Another important aspect to be considered while tuning the used framework is the cluster manager. In the standalone mode, the resource allocation in Spark and Flink are specified by the user during the submission of its jobs whereas using a cluster manager such as Mesos or YARN, the allocation of the resources is done automatically.

2.4 Real-world applications and best practices

In this sub-section, we discuss the use of the studied frameworks in several real-world applications including health care applications, recommender systems, social network analysis and smart cites. More technically, we describe some examples where these frameworks can be applied in specific fields in the best practices part.

2.4.1 Real-world applications

2.4.1.1 Healthcare applications

Healthcare scientific applications, such as body area network provide monitoring capabilities to decide on the health status of a host. This requires deploying hundreds of interconnected sensors over the human body to collect various data including breath, cardiovascular, insulin, blood, glucose and body temperature [Zhang 2015a]. However, sending and processing iteratively such a stream of health data is not supported by the original MapReduce model. Hadoop was initially designed to process Big Data already available in the distributed file system.

In the literature, many extensions have been applied to the original Mapreduce model in order to allow iterative computing such as Haloop system [Bu 2012] and Twister [Ekanayake 2010]. Nevertheless, the two caching functionalities in Haloop that allow reusing processing data in the later iterations and make checking for a fix-point lack efficiency. Also, since processed data may partially remain unchanged through the different iterations, they have to be reloaded and reprocessed at each iteration. This may lead to resource wastage, especially network bandwidth and processor resources.

Unlike Haloop and existing MapReduce extensions, Spark provides support for interactive queries and iterative computing. RDD caching makes Spark efficient and performs well in iterative use cases that require multiple treatments on large in-memory datasets [Bajaber 2016].

2.4.1.2 Recommendation systems

Recommender systems are another field that began to attract more attention, especially with the continuous changes and the growing streams of users' ratings. Unlike traditional recommendation approaches that only deal with static item and user data, new emerging recommender systems must adapt to the high volume of item information and the big stream of user ratings and tastes. In this case, recommender systems must be able to process the big stream of data. For instance, news items are characterized by a high degree of change and user interests vary over time which requires a continuous adjustment of the recommender system. In this case, frameworks like Hadoop are not able to deal with the fast stream of data (e.g. user ratings and comments), which may affect the real evaluation of available items (e.g. product or news). In such a situation, the adoption of effective stream processing frameworks is encouraged in order to avoid overeating or incorporating user/item related data into the recommender system. Tools like Mahout, FlinkML and SparkMLlib include collaborative filtering algorithms, that may be used for e-commerce purposes and in some social network services to suggest suitable items to users [Domann 2016].

2.4.1.3 Social media

Social media is another representative data source for Big Data that requires real-time processing and results. Its is generated from a wide range of Internet applications and Web sites including social and business-oriented networks (e.g. LinkedIn, Facebook), online mobile photo and video sharing services (e.g. Instagram, Youtube, Flickr), etc. This huge volume of social data requires a set of methods and algorithms related to, text analysis, information diffusion, information fusion, community detection and network analytics, which maybe exploited to analyze and process information from social-based sources [Bello-Orgaz 2016a]. This also requires iterative processing and learning capabilities and necessitates the adoption of in-stream frameworks such as Storm and Flink along with their rich libraries.

2.4.1.4 Smart cities

Smart city is a broad concept that encompasses economy, governance, mobility, people, environment and living. It refers to the use of information technology to enhance the quality, the performance and the interactivity of urban services in a city. It also aims to connect several geographically distant cities [43]. Within a smart city, data is collected from sensors installed on utility poles, water lines, buses, trains and traffic lights. The networking of hardware equipment and sensors is referred to as the Internet of Things (IoT) and represents a significant source of Big Data. Big Data technologies are used for several purposes in a smart city including traffic statistics, smart agriculture, healthcare, transport and many others [Stimmel 2015a]. For example, transporters of the logistic company UPS are equipped with operating sensors and GPS devices reporting the states of their engines and their positions respectively. This data is used to predict failures and track the positions of the vehicles. Urban traffic also provides large quantities of data that come from various sensors (e.g., GPSs, public transportation smart cards, weather conditions devices and traffic cameras). To understand this traffic behavior, it is important to reveal hidden and valuable information from the big stream/storage of data. Finding the right programming model is still a challenge

because of the diversity and the growing number of services [44]. Indeed, some use cases are often slow such as urban planning and traffic control issues. Thus, the adoption of a batch-oriented framework like Hadoop is sufficient. Processing urban data in micro-batch fashion is possible to provide eGovernment and public administration services. Other use cases like healthcare services (e.g. remote assistance of patients) need decision making and results within few milliseconds. In this case, real-time processing frameworks like Storm are encouraged.

Combining the strengths of the above discussed frameworks may also be useful to deal with cross-domain smart ecosystems also called big services [Xu 2015].

2.4.2 Best practices

In Section 1.3, two major processing approaches (batch and stream) were studied and compared in terms of speed and resource usage. Choosing the right processing model is a challenging problem, given the growing number of frameworks with similar and various services [Sakr 2016]. This section aims to shed light on the strengths of the above discussed frameworks when exploited in specific fields including stream processing, batch processing, machine learning and graph processing.

2.4.2.1 Stream processing

As the world becomes more connected and influenced by mobile devices and sensors, stream computing emerged as a basic capability of real-time applications in several domains, including monitoring systems, smart cities, financial markets and manufacturing [Bajaber 2016]. However, this flood of data that comes from various sources at high speed always needs to be processed in a short time interval. In this case, Storm and Flink may be considered, as they allow pure stream processing. The design of in-stream applications needs to take into account the frequency and the size of incoming events data. In the case of stream processing, Apache Storm is well-known to be the best choice for the big/high stream oriented applications (billions of events per second/core). As shown in the conducted experiments, Storm performs well and allows resource saving, even if the stream of

events becomes important.

2.4.2.2 Micro-batch processing

In case of batch processing, Spark may be a suitable framework to deal with periodic processing tasks such as Web usage mining, fraud detection, etc. In some situations, there is a need for a programming model that combines both batch and stream behavior over the huge volume/frequency of data in a lambda architecture. In this architecture, periodic analysis tasks are performed in a larger window time. Such behavior is called micro-batch. For instance, data produced by healthcare and IoT applications often require combining batch and stream processing. In this case, frameworks like Flink and Spark may be good candidates [Landset 2015a]. Spark micro-batch behavior allows processing data sets in larger window times. Spark consists of a set of tools, such as SparkMLLIB and Spark Stream that provides rich analysis functionalities in micro-batch. Such behavior requires regrouping the processed data periodically, before performing the analysis task.

2.4.2.3 Machine learning algorithms

Machine learning algorithms are iterative in nature [Landset 2015a]. They are widely used to process huge amounts of data and to exploit the opportunities hidden in Big Data [Zhou 2017b]. Most of the above discussed frameworks support machine learning capabilities through a set of libraries and APIs. FlinkML library includes implementations of K-means clustering algorithm, logistic regression, and Alternating Least Squares (ALS) for recommendation [Chakrabarti 2008]. Spark has a more efficient set of machine learning algorithms such as SparkMLlib [Assefi 2017] and MLI [Sparks 2013]. Spark MLlib is a scalable and fast library that is suitable for general needs and most areas of machine learning. Regarding Hadoop framework, Apache Mahout aims to build scalable and performant machine learning applications on top of Hadoop.

2.4.2.4 Big graph processing

The field of large graph processing has attracted considerable attention because of its huge number of applications, such as the analysis of social networks [Giatsidis 2011], Web graphs [Alvarez-Hamelin 2008] and bioinformatics [Huttenhower 2009] [Dhifli 2017a]. It is important to mention that Hadoop is not the optimal programming model for graph processing [Elser 2013]. This can be explained by the fact that Hadoop uses coarse-grained tasks to do its work, which are too heavyweight for graph processing and iterative algorithms [Landset 2015a]. In addition, Hadoop cannot cache intermediate data in memory for faster performance. We also notice that most of Big Data frameworks provide graph-related libraries (e.g., Graphx [Xin 2013] with Spark and Flinkgelly [Carbone 2015] with Flink).

2.5 Conclusion

In this chapter, we surveyed popular frameworks for large-scale data processing. After a brief description of the main paradigms related to Big Data problems, we presented an overview of the Big Data frameworks Hadoop, Spark, Storm and Flink. We presented a categorization of these frameworks according to some main features such as the used programming model, the type of data sources, the supported programming languages and whether the framework allows iterative processing or not. We also conducted an extensive comparative study of the above presented frameworks on a cluster of machines and we highlighted best practices while using the studied Big Data frameworks.

Key points

- We presented basic notions and features of some Big Data frameworks that help understanding distributed computing.
- We presented an overview of most popular Big Data frameworks.
- We presented a categorization of the presented frameworks and techniques.
- We cared out an extensive set of experiments to evaluate the studied Big Data frameworks.
- We showed a description of best practices related to the use of popular Big Data frameworks in several application domains.

Publications

- Inoubli. W, Aridhi. S, Mezni. H, Maddouri. M et Mephu Nguifo. E. An experimental survey on Big Data frameworks Journal: Future Generation Computer Systems Elsevier, 86, pp. 546-564, 2018
- W. Inoubli , S. Aridhi, H. Mezni, M. Maddouri and E. Mephu Nguifo. A Comparative Study on Streaming Frameworks for Big Data . Proceedings of the Latin America Data Science Workshop co-located with 44th International Conference on Very Large Data Bases (VLDB 2018), Rio de Janeiro, Brazil, Aug 27, 2018.
- W. Inoubli, S. Aridhi, H. Mezni, M. Maddouri, E. M. Nguifo. An Experimental Survey on Big Data Frameworks. 34-èmes journées de la conférence « Gestion de Données – Principes, Technologies et Applications » (BDA 2018), Bucarest, Romania.
- W. Inoubli, S. Aridhi, H. Mezni, M. Maddouri and E. Mephu Nguifo. An Experimental Survey on Big Data Frameworks. Extremely Large Databases Conference (XLDB) 2017, Clermont Ferrand, France. (Lightning talk, poster)

Chapter 3

Graph mining

Contents

3.1 Graph: definitions 55

3.2 Related work on big graph: processing models and frameworks 56

3.2.1 Graph programming models 56

3.2.1.1 Vertex-centric model 56

3.2.1.2 Edge-centric model 57

3.2.1.3 Gather-Apply-Scatter model (GAS) 57

3.2.1.4 Block-centric model 59

3.2.2 Summary of graph processing models 60

3.3 Related work on big graph processing frameworks 61

3.3.1 Graph processing frameworks 61

3.3.2 Graph processing frameworks: an overview 62

3.3.3 Summary of graph processing framworks 66

3.4 Related work on graph clustering 67

3.4.1 Modularity based algorithm 68

3.4.2 Markov Clustering (MCL) 69

3.4.3 Label propagation algorithm 70

3.4.4 Spectral clustering 71

3.4.5 Structural graph clustering 71

3.4.6 Summary of graph clustering algorithms 72

3.4.7 Related work on structural graph clustering 74

3.5 Conclusion 77

Goals

In the previous chapter, an overview of distributed computing and some Big Data frameworks were presented.Adding to that, a comparative study has been carried out on these frameworks. Both data size and data velocity stand for the main challenges of Big Data analysis applications. In the same way, big graph analysis imposes several challenges such as graph size and velocity. This chapter introduces basic definitions and the related work on graph mining. The related work provides an overview of the graph processing fields, and we present some graph clustering algorithms.

Keywords: GraphX, Pregel, Graph processing models, Graph processing frameworks, Graph clustering.

3.1 Graph: definitions

Graph (sometimes called network) is a data structure similar to the relational structure. A graph is noted by $G = (V,E)$, where the V is a set of vertices, whereas $E = V \times V$ denote the set of edges. Each element $e \in E$ represents a link between a pair of vertices and is noted as follows $e(v_i, v_j)$, where v_i and $v_j \in V$. This graph structure can represent several real-world applications (e.g.;social network) where the vertices and edges represent respectively the social network members and the relationships between them. We can also find various extensions of this data structure as an attributed graph, a directed or an undirected graph, and Weighted or unweighted graph.

Definition 3.1.1 *Weighted and unweighted graph. A basic graph is a set of edges that link each pair of vertices. As described above, an edge $e = (v_1,v_2)$ in E. But in some cases, some edges are more important than others, so they are given higher weights. In this perspective, weighted graphs are introduced, making the ponderation of edges possible. For more details, each edge can be embedded with a value according to its weight. Given an edge e in E, $e= (v_1,v_2,w)$ is a triple value, where v_1 and v_2 correspond to the edge bounds, and w is the edge weight.*

Definition 3.1.2 *Directed and Undirected Graph. An undirected graph $G = (V,E)$ consists of a list of unidirectional edges. Otherwise, there is no associated directions of edges, unlike to directed graphs where a direction must be added to each edge.*

Definition 3.1.3 *Attributed graph. An attributed graph can be an undirected or a directed graph. It is defined as $G = (V, E, A_e, A_v)$. As detailed in Definition 3.1, V being the set of vertices, E is the set of edges. Two variables A_e and A_v are added to the basic definition to materialize the attributed graph. Added variables represent respectively the attributes of vertices and edges in G. For more details, $A_e = (A_e1, A_e2, ..A_{en})$ represent the attributes of each edge in G while $A_v = (Av_1, Av_2, ..)$ represent the attributes associated with all vertices in G.*

3.2 Related work on big graph: processing models and frameworks

3.2.1 Graph programming models

Implementing distributed graph applications is an extremely hard task, where the parallel computing, graph partitioning, and communication management are major challenges of distributed graph algorithms. To cope with the issues of distributed graph processing, several high-level programming models have been recently introduced. In the next section, we present the popular used programming models.

3.2.1.1 Vertex-centric model

The principle of this model [Kalavri 2017] is how to think like a vertex. One programmer summarizes the program to be applied as an input graph through the implementation of each vertex in the graph. The user-defined function in the vertex is iteratively executed until satisfying a defined goal or when reaching the defined number of iterations, knowing that the number of iterations is defined by the user. During the processing, each vertex can access to its neighboring vertices in order to get some pieces of information or data useful for its treatment. Otherwise, a program can be modeled as a sequence of exchanged messages between vertices. This sequence represents the program steps and during the latter the vertices can change their status. Since each vertex can run its internal program independently from the other vertices, this independence makes the implementation of a parallel or distributed program more effective. Fig. 3.1 provides an explanation of concepts of vertex-centric programming. In the initial graph state , a programmer defines the *vertex-defined* function and runs the program iteratively. During each iteration, a vertex changes its status from active to inactive, according to both internal value and received values from its neighbors. In iteration 3, all vertices are inactive, so the program will stop. As shown in Fig. 3.1 all vertices are inactive initially, and after some iterations they converge to be inactive vertices (iteration 3).

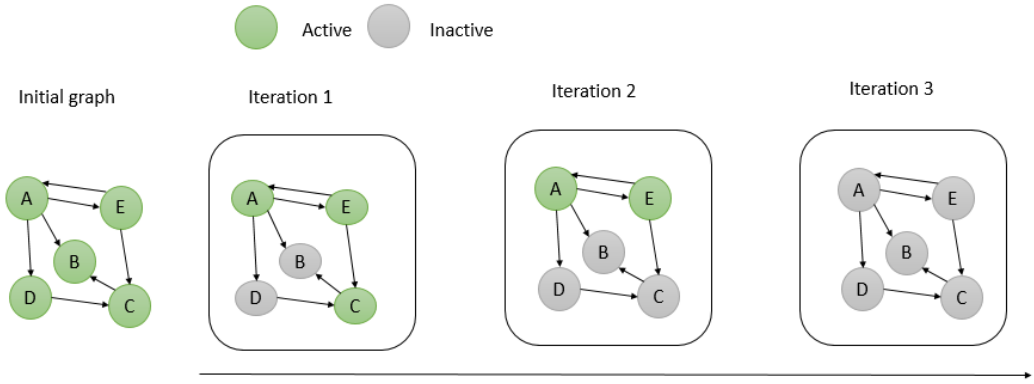


Figure 3.1: Vertex-centric programming model

3.2.1.2 Edge-centric model

This model [Zhou 2017a] is based on the streaming of edges. The processor takes one graph as an input set of edges. Each coming edge has two connected vertices and an edge value. Each edge is computed by the processor and is stored in the global graph. This model is also designed for iterative algorithms. Similarly to the vertex-centric model, this latter describes a graph program for each edge. In this model, a program is executed in several iterations. In each iteration, an edge is processed in three steps: (i) collect the information from the sources (vertices), (ii) update its internal values, and (iii) send the new value to its destination vertices.

3.2.1.3 Gather-Apply-Scatter model (GAS)

This model [Gonzalez 2014] is similar to the Vertex-centric model. Each graph must be viewed as a vertex and implement a program from a vertex point of view. Nevertheless, it divides the vertex program, which will be executed in each iteration) into three sub-functions: *gather*, apply and *scatter*. In the *gather* function, the current vertex gets information from its neighboring vertices or edges and optionally aggregates them in a single value σ. In the *apply* function, the state of the vertex is updated based on the σ value, and probably on the specific features of the vertex neighbors. Finally, in the *scatter* function, each vertex shares its new state to its neighboring vertices or edges. Then, every implemented program must

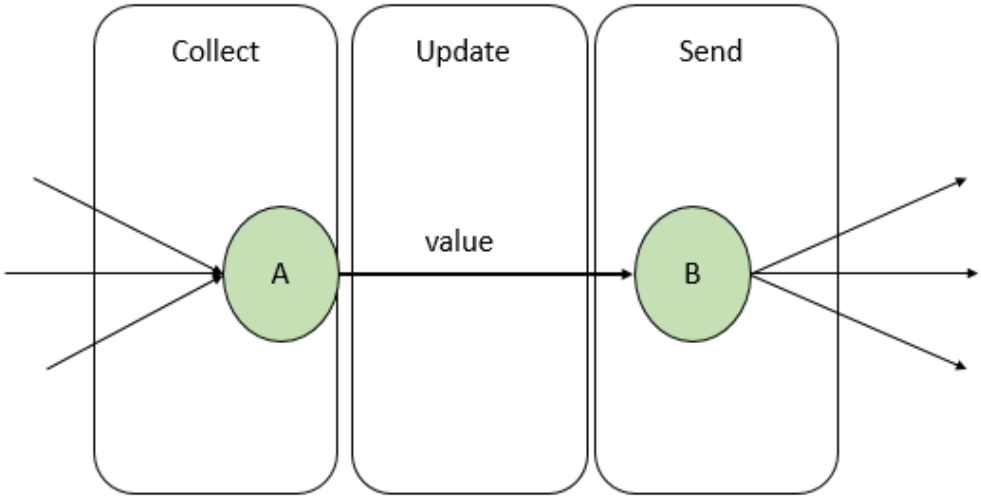


Figure 3.2: Edge-centric programming model

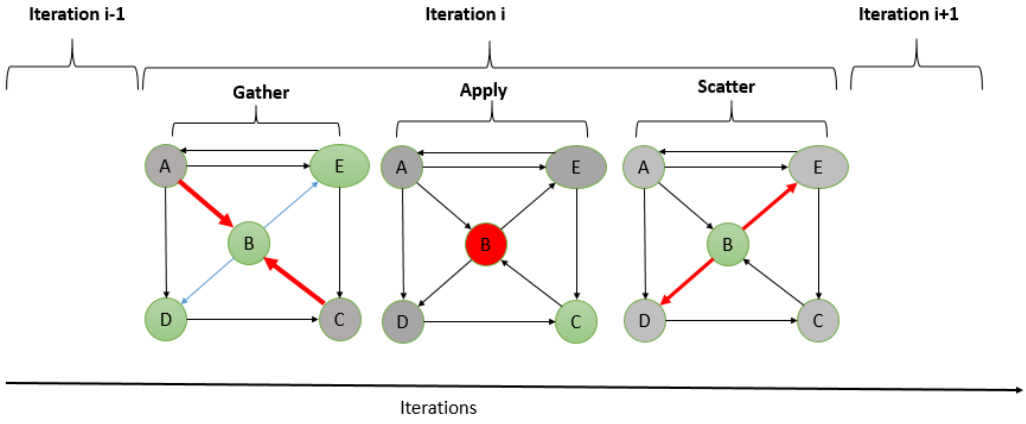


Figure 3.3: GAS programming model

be modeled as three functions that manipulate the σ values in order to change the vertex state or to stop the program. As shown in Fig . 3.3, a graph algorithm is a set of steps or iterations. In the first sub-iteration (*Gather*), each active and destination vertex gets information from its neighbors. Thereafter, the *Apply* function is executed in order to update the value σ and to change the vertex state.

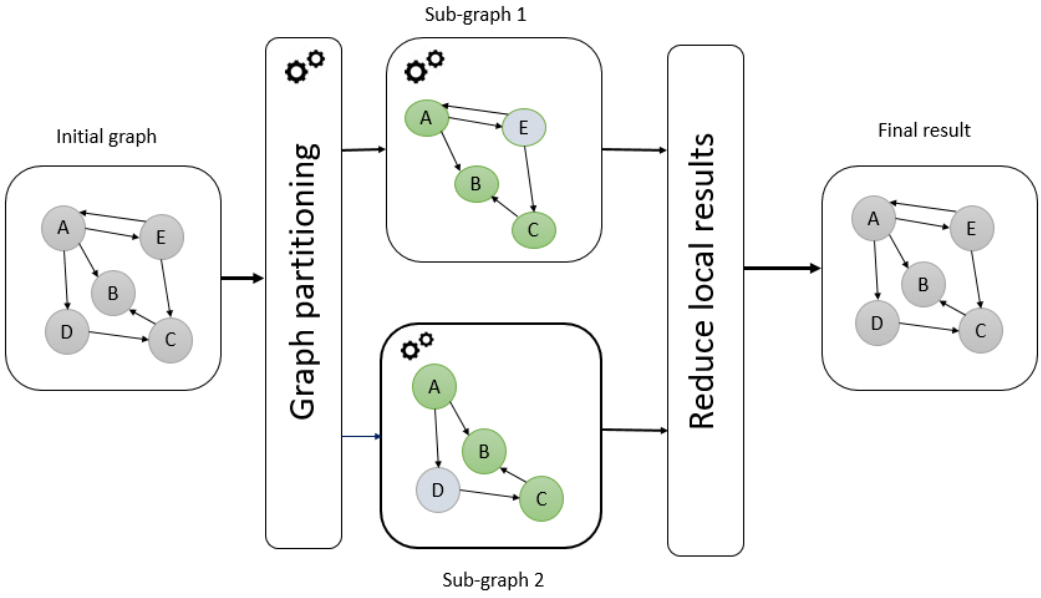


Figure 3.4: Block-centric programming model flow chart

Finally, the current vertex sends its value to all its neighbors. As illustrated in Fig . 3.3, when the vertex B is in run, vertices A and C send information to B. Vertex B updates its σ value, its state, and sends this value to the vertices D and E.

3.2.1.4 Block-centric model

The strategy of this model is to think like a sub-graph or a partition [Fan 2017]. In the above abstraction models, each vertex or edge in an input graph is executed dependently in a slot or machine with a parallel or a distributed setting. The execution is performed iteratively and the communication must be ensured between slots or machines. In the block-centric model, however,the input graph is divided into a set of sub-graphs. Then, these sub-graphs are assigned to slots (in a parallel setting) or to machines (in a distributed setting), in order to execute a user-defined function on the sub-graphs in each iteration. Fig. 3.4 illustrates a general three-step workflow of a graph program using the Block-centric model. Splitting an input graph into sub-graphs represents the first step of this model, while preserving the coherence of the graph. As Fig. 3.4 shows, sub-graph 1 and

sub-graph 2 share the vertices A, B and C as frontiers vertices. The second step is the sub-graph processing. A user-defined function is executed in this step on all sub-graphs iteratively or in a single iteration. In the last step, after generating local results, these latter should be aggregated into one global result.

3.2.2 Summary of graph processing models

In this section, we summarize some advantages and limits for each graph programming model. Through Tab. 3.1, we can see that all processing models use

Graph programming model	Advantages	Limits	Partitioning method
Vertex-centric model	Easily allows parallelism	Communication cost is high	Random hash-based partitioning
Edge-centric model	Easily allows parallelism and allows the streaming graph processing	It is designed for a stream graph processing	Random hash-based partitioning
GAS model	Easily allows parallelism	Communication cost is high	Random hash-based partitioning
Block-centric model	Minimizes the communication costs	An additional cost of partitioning step	Specific partitioning method

Table 3.1: Graph processing models comparative study

the graph partitioning, step since they are designed for distributed and parallel settings. Therefore, an input graph must be distributed on machines or slots, in order to perform a distributed processing. Tab 3.1, shows that GAS, Edge-centric, and Vertex-centric use a random hash-based partitioning method. This method is based on a hash function to split sets of vertices or edges into sets of machines or partitions. But, the Block-centric model uses a partitioning method which consists in spliting a graph into sub-graphs or partitions. This method is very expensive in

terms of running time, since it takes into consideration the graph density in order to lower the communication costs. Thereby, the partitioning step represents an advantage for some models such as GAS, Edge-centric, and Vertex-centric model which is less expensive. However, the graph partitioning step is considered as a major weakness in the Block-centric model. In fact, thinking like a vertex or an edge can facilitate the parallelism of processing, because these models are characterized by independence between their vertices and edges. But, they need high communications in each step. On the other hand, the communication cost of the Block-centric model is extremely low compared to that in other models. This is due to the aggregation of several vertices and edges into some partitions.

3.3 Related work on big graph processing frameworks

3.3.1 Graph processing frameworks

Due to the Big data revolution, several abstract programming models have been proposed like MapReduce [Dean 2008] or dryad [Isard 2007]. These models are implemented as research projects or in industrial projects such as Hadoop, Spark, etc. The graph processing topic has also attracted many researchers and industries. In fact, it is costly in terms of computing resources due to its complexity and to the large size of the real-world graphs. Therefore, a single thread or a simple machine could not meet the emerging needs, especially when dealing with new applications like social networks. As detailed in the previous section, several abstract programming models have been proposed in order to support the new graph processing challenges. These abstract models are implemented in both research and industrial projects. Moreover, several frameworks have been implemented in the context of a single graph programming model. Thus, the large number of frameworks in the literature makes the selection of the right framework for a use case or an application extremely difficult. In this section, we present the most popular frameworks for graph processing, and we enumerate their major features.

3.3.2 Graph processing frameworks: an overview

Pregel [Malewicz 2010b] is a scalable, fault-tolerant and distributed framework for large graph processing. It represents an evolution of the MapReduce framework [Dean 2008] to the graph processing systems. Pregel is available as a C++ API and based on Master/slaves architecture. The Master machine starts to divide an input graph into slave machines candidates. Slave machines are managed by a master machine during their treatment. Pregel is known as the first implementation of the vertex-centric model using the computation model Bulk Synchronous Parallel(BSP) [Goudreau 1996]. It divides a graph algorithm into a sequence of iterations called super-steps, which are limited by separators called barriers. In each super-step, each vertex exchanges messages with its neighbors. Note that communication between the vertices is based on the message-passing method. To write a Pregel program, a user must implement some abstract methods as *Compute()* and *getValue()* which represent the vertex user-defined function and the function to get information from its neighbors respectively.

Giraph [Han 2015] is an open source graph processing framework that was developed by Google. It represents a next generation of Pregel framework. It combines both Mapreduce and vertex-centric programming models. Giraph is based on the Hadoop framework to ensure a distributed system with a master/slaves architecture. The master machine divides the input graph into partitions, sends them a cross the slave machines, and coordinates all global barriers in each super-step. Also, It uses multi-threading processing in each slave machine in order to optimize the local computing. Each graph algorithm is considered as a MapReduce Job and it uses HDFS to store the input, final and intermediate results. Giraph uses global barriers between consecutive iterations. These barriers make the communication between the workers synchronous, using BSP model.

Graphx [Gonzalez 2014] is an API for graph processing, which is embedded within the Spark distributed dataflow system. GraphX takes advantage of the Resilient Distributed Dataset (RDD) to introduce the Resilient Distributed Graph (RDG) data storage abstraction. This abstraction allows Graphx to distribute the input data (graph) in a distributed memory, which allows a parallel processing using

several high-level functions. To implement a graph algorithm using Graphx, a thinking like a vertex is required. The graph processing model used by graphX is the GAS model, which provides three functions (gather, apply and scatter) in each iteration. Also, the GAS model allows both synchronous and asynchronous communication modes during the graph processing.
The data structure used by the GraphX framework is represented as a pair of vertex and edge. The first pair is a list of vertices objects. These objects are defined by the user and each one is keyed by unique identifier. The second pair is the edges list. This pair represents a set of edges objects also defined by the user and keyed by the source and destination vertex identifiers. This structure is called triplet representation, and allows GraphX to use the basic operations of Spark core system such as the map, joint, groupBy. In addition, other specific functions proposed by GraphX like mrTriplets (MapReduce triblets), sendMsg, Vprog. This abstraction also allows the use of a declarative programming language like SQL, in order to perform a descriptive analysis of the graph.
GraphIn To meet the dynamic graphs needs, GraphIn [Sengupta 2016] framework was proposed as an incremental and multicore graph processing engine. It is implemented on the top of another graph processing framewrok called GraphMat [Sundaram 2015]. The latter is also a Graph processing framework that is designed to support large graphs using a multicore processing. Thus, GraphIn is adopted for both dynamic and large graphs using High Performance Computing (HPC) machines. GraphIn is mainly designed for dynamic graphs processing. In this context, it introduces a novel programming model known as Incremental-Gather-Apply-Scatter (I-GAS). Through this model, the processing of a dynamic or stream graph can be performed in an incremental way. It considers a dynamic graph as a set of batches, and in each batch, it checks only the changed vertices. These vertices are called inconsistent vertices.
Blogel [Fan 2017] is one of the first distributed block-centric graph processing frameworks. It is an open source project implemented in C++. Using Libhdfs library, it uses the HDFS as a graph storage system. The main motivation behind the introduction of the Blogel framework is the high diameter of real-world graphs. In fact, using a vertex-centric programming model for these graphs needs high

number of super-steps. For example, a single-source shortest path algorithm in [Malewicz 2010b] takes 10,789 super-steps on a USA road network graph. These latter requires expensive communication costs. which decreased the performance of the graph algorithm. Thus, Blogel was proposed to overcome this limit using the bloc-centric model where an input graph must be divided into blocks or partitions. Blogel supports also several predefined partitioning method and provides many interfaces to users, , allowing them to implement their own methods.

GraphLab [Low 2012] is an open-source and distributed graph processing Framework. It is implemented in C++ and its programming philosophy is similar to the GAS model, which is among the thinking like vertex processing model. But, this program should be divided into three functions (Gather, Apply and Scatter). GraphLab provides two execution modes: synchronous and asynchronous. The synchronous mode is ensured by the BSP model which uses barriers between the super-steps. Differently, the asynchronous mode does not uses neither barriers or super-steps. Unlike the Pregel framework, the communication between the super-steps is ensured by the shared memory. The computation model used by graphLab is similar to Mapreduce. It is based on the implementation of two predefined functions: the first one is called update function for a local computing and the second one aggregates the local results and is called sync mechanism.

PowerGraph [Gonzalez 2012] using a distributed framework for graph processing depends on the graph partitioning. This latter can affect the performance of any graph processing framework, since with a bad partitioning scheme a cost of processing and communication must be added. Based on this property, PowerGraph has been introduced. Being a distributed and parallel framework for large graph processing, PowerGraph takes into consideration the real-world graphs property. Where the degree of vertices is varied. In other words, some vertices have very high degrees while others have very low ones. The PowerGraph framework has tackled this challenge using a partitioning technique that affects the high degree vertices to several workers in order to ensure the load balancing between all workers. Otherwise, by exploiting the GAS model as a programming model, PowerGraph was able to ensure the distributed computation of a single vertex on several workers. In addition, the communication between all workers have been ensured by the shared

memory mechanism, which allows an asynchronous processing.

BLADYG [Aridhi 2017] is a distributed and parallel graph processing framework that runs on a commodity of machines. This framework is based on block-centric graph processing model. The architecture of BLADYG is based on a master/slaves topology. BLADYG starts by reading the input graph from many different sources, which can be local or distributed files, such as HDFS and Amazon Simple Storage Service (Amazon S3). The communication model used by BLADYG is the message passing technique, which consists in sending messages explicitly from one component to another in order to get or send useful data during the graph processing. In the same way, BLADYG defines two types of messages: (1) worker-to-worker (W2W) messages, and (2) master-to-worker messages (M2W). BLADYG allows its users to implement their own partitioning techniques. The synchronous technique used in BLADYG is PSP model with the partitions or machines when barriers are used between iterations in an iterative graph algorithms. Also, BLADYG uses an asynchronous communication using a shared memory among vertices in the same partition.

Other systems The very large number of graph processing frameworks motivated us to discuss all these systems. We choose to present some other graph processing frameworks according to their scope in the literature and their uses in experimental studies and industrial projects. For example, some frameworks represent a continuity of an existing project, such as Hama [Siddique 2016] mizan [Khayyat 2013] and GPS [Salihoglu 2013] which enhance the communication and data partitioning of the basic Pregel system. Pegasus [Kang 2009] is a large-scale graph mining system that has been implemented on top of the Hadoop framework. It uses MapReduce to directly implement iterative graph algorithms and HDFS for the data sharing between the super-steps. Trinity [Shao 2013] is another graph processing framework that is based on a distributed memory processing. Using a share distributed memory, Trinity has optimized the communication cost. GraphMat [Sundaram 2015], Graph++ and GRACE[Prabhakaran 2012] are other graph processing frameworks which are qualified as centralized systems, but they use a partitioning technique in order to improve their performance.

3.3.3 Summary of graph processing framworks

Framework	Programming model	Distributed /Centralized	Communication	Dynamic/Static graph
Pregel	Vertex-centric	Distributed	Message passing	Static
GraphX	GAS	Distributed	Shared memory	Static
BLADYG	Block-Centric	Distributed	Hybrid	Dynamic
Giraph	Vertex-centric	Distributed	Message passing	Static
GraphIn	I-GAS	Centralized	Message passing	Dynamic
GraphMat	GAS	Centralized	Message passing	Static
PowerGraph	GAS	Distributed	Shared memory	Static
GraphLab	GAS	Distributed	Shared memory	Static
Blogel	Block-Centric	Distributed	Message passing	Static
Mezan	Vertex-centric	Distributed	Message passing	Static
GPS	Vertex-centric	Distributed	Message passing	Static
Trinity	GAS	Distributed	Shared memory	Static
Hama	Vertex-centric	Distributed	Message passing	Static
Pegasus	MapReduce	Distributed	no communication	Static
Giraph++	Vertex-centric	Centralized	Message passing	Static
GRACE	Vertex-centric	Centralized	message passing	Static

Table 3.2: Graph processing frameworks comparative study

Through the comparative Table. 3.2, we established a summary study of the studied graph processing frameworks. The used features in this table are determined according to the literature. These invoices have represented contributions to combat some problem related to the processing of large graphs. However,

there is no perfect framework for each application or use case. For example, in the case of modest graphs multi-core and centralized frameworks should be used in this kind of applications . Thereby, GraphMat or GraphIn are the most suitable frameworks for this application. Moreover, some applications require a lot of communication which will make the shared memory frameworks represent the right choice. Also, the shared memory can ensure synchronous communication which is required in some use cases especially in unbalanced graphs. In the same way, the study application in this project is the big and dynamic graphs clustering. Wherefore, the used framework should support both dynamic and big graphs. As it is shown in Table. 3.2 most frameworks are distributed, but only two frameworks (BLADYG and GraphIn) can support dynamic graphs. However GraphIn is a multi-core framework that cannot support both big graphs and distributed graphs.

3.4 Related work on graph clustering

Graphs are one of the most used data models in several applications like social networks [Said 2018], road maps [Cao 2009], bioinformatics [Xu 2002]. For instance, a recent 5 ranking shows that the popularity of graph databases model increased up to around 500% in the last years [Gandomi 2015a]. It has shown that graphs are optimum data models that are able to represent easily many relationships and to facilitate the exploration of data. Taking social networks as an example, the graph model organizes data elements into a set of vertices representing the members, and a list of edges to materialize the relationships between them (vertices). Also, the last years featured a Big data explosion especially in graph-based social networks. As an example, Facebook in 2013 had over 874 million monthly users [Gandomi 2015a]. This proliferation of a huge amount of data and the massiveness of graphs introduce several research topics such as graph storage, graph indexing, graph processing and graph mining. This latter also occurs in several use cases such as nodes classification, links prediction, graph clustering which is sometimes called *community detection*. In this thesis, we focus on the graph clustering.
Graph Clustering is a fundamental problem in graph mining area. It groups the

vertices of an input graph into a collection of dense and disjoint subsets called clusters [Žalik 2018, Günnemann 2012]. Different clusters have to be weakly connected between them. Graph clustering has many applications. For example, the protein annotation task is among the challenges in bioinformatics fields. This task consists of an understanding of the expression, function, and regulation of encoded proteins by an organism. These proteins can be represented by a graph of proteins often called Protein Interaction (PPI) network. In this network, proteins in each subset interact together in order to perform a specific biological function. In this context, graph clustering represents one of a set of methods used for detecting protein in the PPI. In some other applications, the used graph could be very large and could be partitioned into several sub-graphs, where the computation is performed in a parallel/distributed way. Generally, graph clustering algorithms are used to ensure the graph partitioning. Likewise, detecting and analyzing research communities is a real-world application, which uses graph clustering to aggregate the authors under the same area according to the co-authorship (collaboration) relationships.

The complexity of the graph data model and its benefits drive the search to propose several algorithms for this field. Proposed algorithms oscillate from modularity-based algorithm [LaSalle 2015] Markov clustering [Lei 2016] Label propagation algorithm [Xie 2013] Spectral based clustering [White 2005a] and structural clustering. These algorithms are detailed in the next subsections.

3.4.1 Modularity based algorithm

Basically, the modularity [Brandes 2007] is a used metric for measuring the strength of partitioning a graph into groups, in addition to the quality of clustering. The goal of this measure is to quantify the clustering schema using the number of edges between the different clusters. These edges are also called 'fraction of edges'. The formula of modularity [Aktunc 2015] can be written as in Eq. 3.1:

$$Q = \frac{1}{2*m}\sum_{uv}^{n}\left[A_{uv} - \frac{d_u d_v}{2*m}\right]\delta(C_u, C_v) \tag{3.1}$$

where m is the number of edges in the graph, v and u are two vertices, and d_u and d_v are the degrees of u and v, respectively. A_{vw} represents the connectivity between u and v, which is 1 when u and v are connected by an edge and is 0 otherwise. C_u and C_v are the clusters that contain u and v, respectively. The value of δ function is 1 when C_u and C_v is the same cluster; otherwise it is 0.

The modularity measure can be applied in order to perform the graph clustering. When generating several partitioning schema and evaluating the modularity of each partitioning scheme in order to select the best scheme. The partitioning generation step is an NP-complet problem. In order to optimize the modularity quality we need to evaluate all possible partitioning scheme which depend on the number of vertices in an input graph. In this context, the authors in [Zeng 2015] have proposed an heuristic method called the Louvain method. This latter is iterative and is based on multiple phases. The first step is to find a local maximum which consists to assign each vertex to a cluster, after that then clustering vertices to the neighbor cluster which increase the modularity measure. The last step must be repeated until no individual vertex move can improve the modularity score. The second phase is reserved to the reconstruction of the new network. During this phase, each community funded in the first phase is considered as a super vertex. The modularity measure between all founded communities is applied using the edge between the latter.

3.4.2 Markov Clustering (MCL)

Markov clustering (MCL) [Vlasblom 2009] is one of the most popular graph clustering algorithms. It is frequently used in the bioinformatic field, especially in protein-protein interaction networks (PPINs) and protein similarity networks and demonstrates its usefulness in these use cases. The frequent use of this algorithm in bioinformatics is explained by the the clusters' balanced feature. Because, for complex problems such as protein finding, it is required to not find big clusters. The ideal cluster size (complex proteins) does not exceed 15-30 vertices [Satuluri 2010]. It is also important to deal with a limited set of singleton clusters. MCL is based on the simulation of random walks using a matrix of probabilities of transition in

an input graph. This algorithm consists on two phases. The first one is to build an adjacency matrix A from an input graph, while the second one is to normalize the columns in order to produce the transition probabilities matrix noted M, and which is defined as follows:

$$M_{ij} = \frac{A_{ij}}{\sum_{k=1}^{n} A_{kj}} \tag{3.2}$$

For more details, the second phase is devoted to calculate the matrix probability. This matrix is initialized according to Eq. 3.2 and iterates the followed steps until the convergence. The iterative process can be divided into two steps: The first one is called the expand step, which represents a simple matrix multiplication $M{=}M{*}M$. As for the second one, it is called inflate step that represents a normalization of each column of M using the next equation:

$$M_{ij} = \frac{(M_{ij})^r}{\sum_{k=1}^{n} (M_{kj})^r}, r > 1. \tag{3.3}$$

Then, a pruning technique is applied on M to ensure that values below a given threshold (r) are set to 0. When MCL converges, the clustering step is applied to aggregate the vertices which have non-zero values in the same row of M are assigned to the same cluster. In the case where two nodes have multiple non-zero values in their columns, the row selection is done arbitrarily. Note that until the convergence, MCL must ensure that each column in M has at least one non-zero value. With the iterative multiplication of the matrix and especially with large graphs, MCL uses a lot of computing resources and takes a lot of time. In return, it ensures that vertices in dense regions will appear in a single cluster only, which ensure the non-overlapping clustering.

3.4.3 Label propagation algorithm

Label propagation algorithm (LPA) [Wang 2007] is an iterative and heuristic algorithm used by several works in several fields including community detection and bioinformatics. LPA uses both the graph structure and the label of each vertex in the graph in order to group vertices into clusters. Initially, LPA starts by distin-

guishing labels for all vertices and propagate these them until convergence. In the propagation step, each vertex sends its own label to its neighbors. Usually, each vertex changes its label by the largest value of its neighbors or using a defined function. After some iterations, the LPA converges. In this case, it groups the vertices that share the same label value into the same cluster.

3.4.4 Spectral clustering

The spectral clustering [White 2005b] is based on the graph density unlike other graph clustering algorithms. The density of a graph provides the optimal number of clusters. Then, the similarity measure can be applied between data points in order to aggregate similar vertices. The goal of this algorithm is to transform a graph as data points and apply any other basic clustering algorithm. At the first step, this algorithm starts by representing an input graph with an adjacency matrix A. This latter represents the links and edges between all vertices in the graph, and the degree matrix $D = n^*n$, where n the number of vertices in the graph. The D matrix has for each vertex, the number of its adjacency or its degree. At the second step, it computes the laplacian matrix L which is defined as L=D-A. After that, it computes the eigenvalues (lambda) and eigenvectors (x) of A with x as follows :

$$A * x = lambda * x \tag{3.4}$$

The third step, consists in selecting the largest n eigenvalues and redefining the input space as a n dimensional subspace. A basic clustering algorithm, like K-means, is finally applied on the new redefined space with n dimensions.

3.4.5 Structural graph clustering

The Structural Clustering Algorithm for Networks (SCAN) was proposed in [Xu 2007] aiming, not only to identify the clusters in a graph, but also to provide additional informations like hubs (vertices between one or more clusters) and outliers (vertices that do not belong to any cluster). These additional pieces of information can be used to detect vertices that can be considered as noise and

also vertices that can be considered as bridges between clusters. The functional principle of SCAN is based on graph topology. It consists of grouping vertices that share the maximum number of neighbors. Moreover, it computes the similarity between all the edges of the graph in order to perform the clustering. The similarity computation step in SCAN is quadratic according to the number of edges, which degrades SCAN performance especially in the case of large graphs. Structural graph clustering is one of the most effective clustering methods for differentiating the various types of vertices in a graph.

3.4.6 Summary of graph clustering algorithms

Tab. 3.3 shows an overview of the discussed algorithms. As it is shown, modularity-based algorithm, label propagation and Markov clustering algorithm are heuristic methods which are mainly based on the propagation step in order to give the final result. The results given by these algorithms are based on the density of the graph. In addition, Markov clustering algorithm ensures the non-overlapping clustering as required in some cases. Also, Label propagation algorithm does not need any parameter or number of clusters required while running. Moreover, both Spectral clustering and Markov clustering are based only on the matrix representation. This representation offers several advantages, such as ease of manipulation. However, it poses a scalability problem in the case of large graphs and also in the dynamic graphs, especially in the adding/removing of a vertex. Whereas in each each update, a new matrix must be builded. The Structural clustering, unlike modularity based label propagation and Markov clustering is an exact method for the graph clustering. It uses any graph representation, which ensure a flexibility with the dynamic graphs. In addition to that, the above methods provide, as output, a list of clusters which are not really sufficient to understand the graph behavior. To address this issue, the Structural Clustering Algorithm for Networks (SCAN) was proposed in [Xu 2007] aiming, not only to identify the clusters in a graph, but also to provide additional information like hubs and outliers. Motivated by this comparative study and based on the advantage of structural clustering, we choose this algorithm as a reference algorithm. In the next section, we discuss

Algorithm	Advantages	Disadvantages
Modularity based [Brandes 2007]	• Maximize the modularity of the clustering schema	• Convergence can take a lot of time • Heuristic method
Markov clustering [Vlasblom 2009]	• Non-overlapping clustering	• Only the matrix representation must be used • Matrix multiplication followed by inflation operator
Spectral clustering [White 2005b]	• Algorithm unsupervised • Using the basic clustering algorithm	• Only the matrix representation must be used
Label propagation [Wang 2007]	• The number of clusters do not required	• Heuristic method • Results produced by LPA are not stable • Convergence can take a lot of time
Structural clustering [Xu 2007]	• Differentiate the types of vertices • Provides outliers and bridges vertices	• Processing time is high in the large graph

Table 3.3: Comparative study on the graph clustering algorithms

some works related to the structural graph clustering in order to propose a new algorithm that supports both big and dynamic graphs.

3.4.7 Related work on structural graph clustering

The functional principle of SCAN is based on graph topology. It consists of grouping vertices that share the maximum number of neighbors. Moreover, it computes the similarity between all the edges of the graph in order to perform the clustering. The similarity computation step in SCAN is quadratic according to the number of edges, which degrades SCAN performance especially in case of large graphs. Structural graph clustering is one of the most effective clustering methods for differentiating the various types of vertices in a graph. In the literature, several works have been proposed for the structural graph clustering to overcome the drawback of SCAN. In [Shiokawa 2015], Shiokawa et al. proposed an extension of the basic SCAN algorithm, namely SCAN++. The proposed algorithm aims to introduce a new data structure of directly two-hop-away reachable node set (DTAR). This new data structure is the set of two-hop-away nodes from a given node that are likely to be in the same cluster as the given node. SCAN++ could save many structural similarity operations, since it avoids several computations of structural similarity by vertices that are shared between the neighbors of a vertex and its two-hop-away vertices.

In the same way, the authors in [Chang 2017a] suppose that the identification of core vertices represents an essential and expensive task in SCAN. Based on this assumption, they proposed a pruning method for identifying the core vertices after a pruning step, which aims to avoid a high number of structural similarity computations. To improve the performance and robustness of the basic SCAN, an algorithm named LinkSCAN* has been proposed in [Lim 2014]. LinkSCAN* is based on a sampling method, which is applied on the edges of a given graph. This sampling aims to reduce the number of structural similarity operations that should be executed. However, LinkSCAN* provides approximate results.

Other works have proposed parallel implementations of SCAN algorithm [Takahashi 2017] [Mai 2017] [Chang 2017b] [Stovall 2015] [Wen 2017]. In

[Takahashi 2017], the authors proposed an approach based on openMP library [Bull 1999]. The authors' aims were to ensure a parallel computation of the structural similarity and to show the impact of parallelism on the response time. Their method was proven to be faster than the basic SCAN algorithm. Other works have focused on the problem of dynamic graph clustering. In [Mai 2017] and [Chang 2017b], the authors have extended SCAN algorithm to deal with the addition or removal of nodes and edges. Authors in [Stovall 2015], used the graphical processing unit (GPU) whose purpose is to parallelize the processing and to benefit from the high number of processing slots in the GPU which increase the degree of parallelism.

Most of the above presented works face a two major problems: (1) they do not deal with big graphs and (2) they do not consider already distributed/partitioned graphs.

algorithm	Parallel	Distributed	Processing model	Graph partitioning	Main contributions
SCAN [Xu 2007]	No	No	Sequential	No	Basic implementation of structural graph clustering
SCAN++ [Shiokawa 2015]	No	No	Sequential	No	Reducing the number of similarity computations
AnySCAN [Zhao 2017]	Yes	No	Parallel	No	Parallelizing SCAN
pSCAN [Chang 2017a]	Yes	No	Parallel	No	Reducing the number of similarity computations
Index-based SCAN [Wen 2017]	No	No	Sequential	No	Interactive SCAN
ppSCAN [Che 2018]	Yes	No	Parallel	No	Parallel version of pSCAN
SCAN based on GPU [Stovall 2015]	Yes	No	Parallel	No	A GPU-based version of SCAN
pm-SCAN [Seo 2017]	Yes	No	Parallel	Yes	Graph partitioning to reduce the I/O costs

Table 3.4: Comparative study on existing structural graph clustering approaches

Through Table 3.4, we have summarized the discussed approaches according to some features.
As shown in Table. 3.4, most proposed algorithms allow parallel processing but could not deal with very large graphs. It is also clear that none of the studied approaches allow distributed computing. In addition, these approaches are unable to process large and dynamic graphs. Also in some applications, like social networks, graphs are distributed by nature. Thereby, using the discussed algorithms. All the partitions of a distributed graph should be aggregated in one machine, in order to run the graph clustering. Based on these limits, in this work, we tackle the problem of large and dynamic graph clustering in a distributed setting.

3.5 Conclusion

In this chapter, we presented a background on graphs as a widely used data structure. Secondly, we discussed the popular graph programming model, which are implemented by several graph processing frameworks. After a quick summary of these programming models, we have shown the main advantages and limits of each one. Then, in order to chose the right graph processing, we surveyed the popular frameworks and, we have compared them according to several features such as programming model, the graph processing model (dynamic or static). Thirdly, state-of-the-art on graph clustering algorithms is provided, after that the structural graph clustering is selected to study its related work. Finally, we have presented the recent work related to the structural graph clustering.

Key points

- We presented some definitions and features of graphs.
- We presented a related work on the graph processing model.
- We discussed the popular graph processing framework, and a theoretical study have been cared out on these frameworks.
- We presented a related work on the graph clustering, and a related work on structural graph clustering has been presented.

Publications

- W. Inoubli, L. Almada, T.L. Coelho da Silva, G. Coutinho, L. Peres, R.P. Magalhaes, J.F. de Macedo, S. Aridhi, E. Mephu Nguifo. A Distributed Framework for Large-Scale Time-Dependent Graph Analysis. Joint Workshop on Large-Scale Evolving Networks and Graphs in conjunction with ECML-PKDD 2017, Skopje, Macedonia.

Part II

Contributions

Chapter 4

Distributed Structural Graph Clustering Algorithm

Contents

4.1 Strctural graph clustering 83

4.1.1 Definitions and basic concepts 83

4.1.2 Running example 85

4.2 DSCAN: A Distributed Algorithm for Large-Scale Graph Clustering 87

4.2.1 Distributed graph partitioning 87

4.2.2 Distributed clustering 88

4.2.2.1 Illustrative example 93

4.2.2.2 Time complexity analysis of DSCAN 94

4.2.2.3 Communication complexity 95

4.3 Experiments 95

4.3.1 Experimental protocol 95

4.3.2 Experimental data 96

4.3.3 Experimental results related to DSCAN 96

4.4 Conclusion 102

Goals The graph model is recently used in several applications such as bioinformatics, chemoinformatics and social networks applications. Otherwise, some use cases have suffers from the size of graphs in the real-world applications

as social network. As mentioned in previous chapter, several frameworks are designed for big graph processing. Due to this fact, we propose a distributed graph clustering in order to overcome the issues discussed in the previous chapter. The proposed algorithm is implemented on BLADYG framework, it can support both centralized and distributed graph. It supports both centralized and distributed graphs. In a centralized graph setting, the proposed algorithm starts by splitting the graph into subgraphs using our proposed graph partitioning. In the distributed graph setting, the proposed algorithm takes the already distributed graph as input and lunch directly the graph clustering task. We also present an experimental study, and we show that our proposed algorithm can support big graphs compared to other algorithms. In addition to that the experimental results show the scalability of our proposed algorithm.

Keywords: Big graph, graph clustering, community detection, graph analysis, distributed graph processing.

4.1 Strctural graph clustering

4.1.1 Definitions and basic concepts

Graph clustering consists in dividing a graph G into several partitions or subgraphs. Like other clustering techniques, we should use one or more metrics to measure the similarity between two vertices or partitions in G. In the structural clustering technique, the graph structure or topology is used to divide G into a set of subgraphs. These subgraphs are relatively distant, and vertices in some subgraphs are strongly connected.

As a well-known algorithm for structural graph clustering, SCAN algorithm uses the structural similarity between vertices to perform the clustering task. In addition, it provides other pieces of information like outliers and bridges. In the following, we present an overview of graphs model and the structural graph clustering algorithm named SCAN [Xu 2007].

Consider a graph $G = \{V,E\}$, where V is a set of vertices, and E is a set of edges between vertices. Each of those elements can represent a real property in real-word applications. Let u and v be two vertices in V. We denote by (u,v) an edge between u and v; u (resp. v) is said to be a neighbor of v (resp. u).

In the following, we extend some basic definitions of structural graph clustering, which will be used in our proposed algorithm.

Definition 4.1.1 *(Structural neighborhood) The structural neighborhood of a vertex v,is denoted by $N(v)$, and represents all the neighbors of a given vertex $v \in V$, including the vertex v:*

$$N(v) = \{u \in V | (v,u) \in E\} \cup \{v\} \tag{4.1}$$

Definition 4.1.2 *(Structural similarity) The structural similarity between each pair of vertices (u,v) in E represents a number of shared structural neighbors between u and v. It is defined by $\sigma(u,v)$.*

$$\sigma(u,v) = \frac{|N(u)| \cap |N(v)|}{\sqrt{|N(u)| \,.\, |N(v)|}} \tag{4.2}$$

After calculating the structural similarity with Eq. (4.2), SCAN uses two parameters to detect the core vertices in a given graph G.

Definition 4.1.3 *(ε-neighborhood) Each vertex has a set of structural neighbors, like it is mentioned in Definition 4.1.1. To group one vertex and its neighbors in the same cluster, they must have a strong connection (denoted by ε-neighborhood) between them. SCAN uses a ε threshold and Eq. (4.3) to filter, for each vertex, its strongest connections. The ε-neighborhood is defined as follows.*

$$N_\varepsilon(u) = \{N(u) | \sigma(u,v) \geq \varepsilon\} \tag{4.3}$$

The ε threshold $0 < \varepsilon \leqslant 1$ shows to what extent two vertices u and v are connected based on the shared structural neighbors. In addition, it represents a metric with which the most important vertices, also called *core* vertices, are detected.

Definition 4.1.4 *(Core) Core vertices detection is a fundamental step in SCAN algorithm. It consists of finding the dominant vertices in a given graph G. This step allows to build the set of clusters or a clustering mapping. A core vertex v is a vertex which has a sufficient number of neighbors strongly connected with it $N_\varepsilon(v)$. We use μ as a minimum number of strong connected neighbors (see Definition 4.1.3). A core vertex is modeled as follows:*

$$V_c = \{v \mid |N_\varepsilon(v)| \geq \mu\} \tag{4.4}$$

Definition 4.1.5 *(Border) Let v_c be a core vertex. v_c has two lists of structural neighbors: (1) weak connected neighbors to v_c, also called noise vertices ($N(V_c) \setminus N_\varepsilon(V_c)$), and (2) strong connected neighbors called reachable structured neighbors. In our work, reachable structured neighbors are called border vertices. $N_\varepsilon(V_c)$ represents the border vertices of a core vertex V_c.*

Once the nodes and their borders are determined, it is straightforward to start a clustering step. To do so, we use the following definition:

Definition 4.1.6 *(Cluster) A cluster C ($|C| \geq 1$) is a nonempty subset of vertices, where its construction is based on the set of core vertices and their border vertices.*

The main steps of clusters' building algorithm are the following: first, randomly chose a core from the cores' list and create a cluster C, then push the core and its borders into the same cluster. At the same time, the algorithm checks if the list of borders has a core vertex. Then, it inserts their borders into the same cluster and it applies this step recursively until adding all the borders of the connected cores. Finally, the algorithm chooses other core vertices and applies the same previous steps until checking all core vertices.

Among the fundamental information returned by SCAN, compared to other graph clustering algorithms, we mention bridge and outlier information, as defined as below:

Definition 4.1.7 *(Bridge and Outlier) The clustering step aggregates the core vertices and their borders into a set of clusters. However, some vertices do not have strong connections with a core vertex, which does not give the possibility to join any cluster. In this context, SCAN algorithm classifies these vertices into two families: bridges and outliers.*
A vertex v, that is not part of any cluster and has at least two neighbors in different clusters, is called bridge. Otherwise it is considered as an outlier.

4.1.2 Running example

In this section, we explain through a running example, how SCAN algorithm works. Consider a graph G presented in Figure 4.1 and the parameters $\varepsilon = 0.7$ and $\mu = 3$. In the first step, lines 1-3 of Algorithm 12 use Eq. (4.2) to compute the structural similarity for each edge $e \in \mathbb{E}$. Then, Eq. (4.3) (lines 5-8) is used to define the core vertices (see gray vertices in Fig. 4.1). After that, we proceed to the clustering step, then we apply Definition 4.1.6 (lines 6-7) to build the clustering schema. In our example we have four core vertices: 0,2,9 and 10. Each core has a list of border vertices (the neighbors that have strong connections with a core). In our example, vertex number 2 is a core. This later has the vertices number 1, 4, 5, 3 and 0 as the list of borders since they have strong connections with the core. The core and its borders build a cluster, and if a border is a core we join all its

Algorithm 1 Basic SCAN algorithm

Input : A Graph G and parameters (μ ,ε)
Output: $\mathbb{C},\mathbb{B},\mathbb{O}$
2 **foreach** $(u,v) \in E$ **do**
3 | Compute $\sigma(u,v)$
4 **end**
5 $Core \leftarrow \emptyset$
foreach $u \in V$ **do**
6 | **if** $(|N_\varepsilon(u)| \geq \mu)$ **then** $Core = Core \cup \{u\}$;
7 **end**
8 $Cluster \leftarrow \emptyset$
foreach *unprocessed core vertex* $u \in Core$ **do**
9 | $Cluster= \leftarrow \{u\}$
| $Mark u as processed$
| **foreach** *unprocessed border of vertex* $v \in N_\varepsilon(u)$ **do**
10 | | $Cluster= \leftarrow$Cluster $\cup \{v\}$ Mark v as processed
| |
11 | **end**
12 **end**

borders into the cluster. Like in our example, the vertex 2 is a core. Hence, we join all its borders (1, 3 ,5 ,4 including 0 as being a core vertex). In this case, if the vertex 3 is not a border of vertex 2 and it is a border of vertex 0, then vertex 3 should belong to the same cluster of vertex 2 (which is a core vertex), since it is a reachable border of vertex 2. The last step of SCAN algorithm consists in defining the bridges and the outliers. As shown in Fig. 4.1, we have two clusters. The first one is composed of vertices 1,2,3,4, and 5, whereas the second cluster is composed of nodes 8, 9, 10, and 11. The remaining vertices (6 and 7) must be categorized as outliers and bridges according to the Definition 4.1.7. In our example, vertex 7 has two connections with two different clusters, and vertex 6 has only one connection with one cluster which makes vertex 7 a bridge and vertex 6 an outlier.

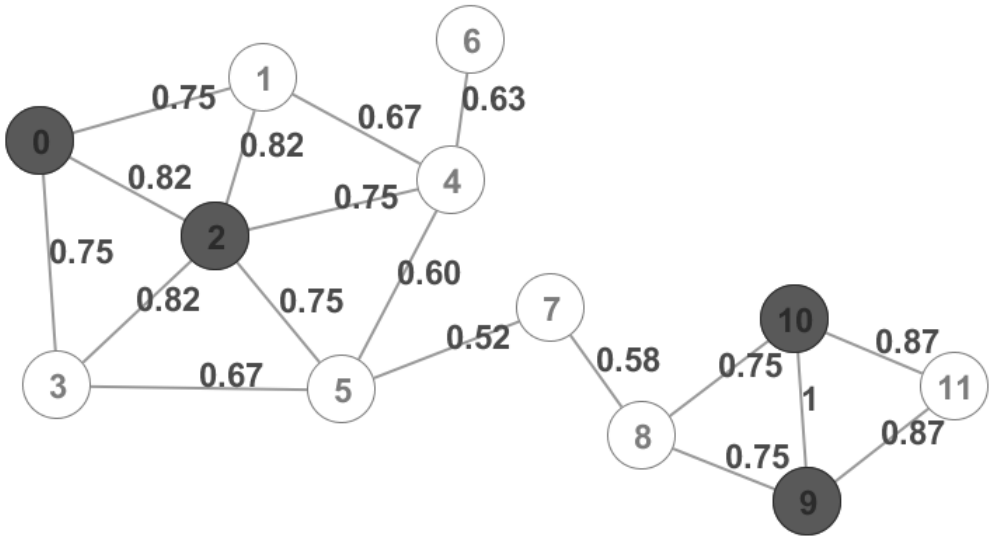


Figure 4.1: Running example ($\varepsilon = 0.7$ and $\mu = 3$).

4.2 DSCAN: A Distributed Algorithm for Large-Scale Graph Clustering

In this section, we introduce DSCAN: a new distributed algorithm for structural graph clustering. Our proposed algorithm is based on a master/slave architecture, and it is implemented on top of BLADYG framework [Aridhi 2017]. This latter is a distributed graph processing framework in which the slaves machines are responsible for the execution of a specific computation and the master machine coordinates between all the slaves. The input data must be divided into sets of chunks (subgraphs in our case). Each chunck/partition is assigned to a worker or slave machine, which performs a specific computation. The master machine orchestrates the workers' execution. In the following, we present the main two steps of DSCAN: (1) graph partitioning and (2) graph clustering.

4.2.1 Distributed graph partitioning

In this step, DSCAN splits the input graph G into several small partitions $P_1, P_2, ..P_n$, while keeping data consistency (graph structure). To ensure the con-

sistency property while dividing the input graph, we must identify a list of cuts edges in order to have a global view of G. Usually, the graph partitioning problem is categorized under the family of NP-hard problems, that need to parse all the combinations in order to get the best partitioning result part. For this reason, we proposed an approximation and a distributed partitioning method as a preliminary step for our distributed clustering algorithm. Algorithm 2 shows that, at the beginning, the master machine divides equitably an input graph file into sub-files according to the number of edges, and sends the sub-files to all the worker machines. Secondly, each worker machine gets a list of edges and vertices from its sub-file. Thereafter, it sends its list of vertices to all workers, in order to determine the frontier vertices so that to get the cuts edges. Afterwards, when each worker receives a list of vertices from its neighbor workers, it determines the vertices that belong to the current worker (partition). Consequently, these vertices are considered as frontier vertices in their partitions. In the last step, when each worker could determine the frontier vertices, it starts to fix the cuts edges, i.e. edges that have a frontier vertex.

4.2.2 Distributed clustering

The distributed clustering step of DSCAN has also two main steps: (1) local clustering step and (2) merging step.

Step 1: Local clustering. As presented in Algorithm 3, the input graph G is splitted into multiple subgraphs/partitions ($\mathbb{P}$), each one is assigned to a worker machine. The partitioning step, as mentioned in Section 4.2.1, is performed according to the α parameter, which refers to the number of worker machines (line 2). To overcome the loss of information during the partitioning step (edges connecting nodes in different partitions), frontier vertices are duplicated into neighboring partitions (line 3-8). Subsequently, for any partition P_i, a local clustering is performed (lines 9-11) on a worker machine. Fig. 4.2 shows a demonstrative example of the duplication step. The demonstrative example describes how the graph consistency will be ensured during the partitioning step.

Assume that a graph G is partitioned into two partitions $P1$ (vertices in blue)

Algorithm 2 Distributed partitioning

Input : Graph file GF as a text file, parameter (NP number of partitions)
Output: $\mathbb{P}$ set of partitions

BLADYG initialization according to the NP parameter
Master machine: split GF into a set of sub-files $\mathbb{GF}$
foreach $GF_i \in \mathbb{GF}$ **do**
    Assign GF_i to worker W_i
end
`/* In parallel */`
foreach *Worker* $W_i \in \mathbb{W}$ **do**
    Get list of vertices and edges from GF_i
    Find the list of frontier vertices from the neighbor workers
end
`/* In parallel */`
foreach *Worker* $W_i \in \mathbb{W}$ **do**
    foreach $Edge \in \mathbb{E}$ **do**
        (a,b)=Edge
        Let P_i^f the frontier vertices
        if *($a \in P_i^f \cup b \in P_i^f$)* **then** ;
        Set the edge E as a cut edge
    end
end

Algorithm 3 DSCAN

Input : Graph G , parameters $(\mu, \varepsilon, \alpha)$
Output: $\mathbb{C}$lusters, $\mathbb{B}$ridges, $\mathbb{O}$utliers

/* Divide G into subgraphs $\mathbb{G}$ = $\{G_1 G_2 .. G_\alpha\}$ according to parameter α */
$\mathbb{P} \longleftarrow$ ***Partition*** (G, α)
BLADYG initialization
/* In parallel */
foreach $P_i \in \mathbb{P}$ **do**
  Assign P_i to W_i
end
/* In parallel */
/* Step 1: Local clustering */
foreach *Worker* $W_i \in \mathbb{W}$ **do**
  Let P_i the current partition
  Find the frontier vertices in P_i and duplicate them into neighbor partitions
end
/* In parallel */
foreach *Worker* $\in \mathbb{W}$ **do**
  Compute the structural similarity of a partition P_i using V^f list
  Retrieve local Cores and Borders in P_i
  Build local clusters in P_i
  Find local Bridges and Outliers in P_i
end
/* Step 2: Merging */
All workers exchange theirs local clusters between them using **Worker2Worker** message
foreach *Worker* $W_i \in \mathbb{W}$ **do**
  if *($C_1 \cap C_2 \cap .. \cap C_\alpha = \mathbb{V}$; and $\exists V_i \in \mathbb{C}ore$)* **then**
    $C \leftarrow$ ***Merge***$(C_1, C_2, .. C_\alpha)$
    Send C to the master
  end
  else
    Send local clusters to the master
  end
  for V_i ***IN*** $\mathbb{O}utliers$ **do**
    if *($V_i \in \mathbb{C}ore \cup \mathbb{B}order \cup \mathbb{B}ridges$)* **then**
      Remove V_i *from the list of* $\mathbb{O}utliers$
    end
  end
  for V_i ***IN*** $\mathbb{O}utliers$ **do**
    $Nb_{Connections} \leftarrow 0$
    for C_i ***IN*** $\mathbb{C}lusters$ **do**
      if *($N(V_i) \cap C \neq \varnothing$)* **then**
        $Nb_{Connections}++$
      end
    end
    if *($Nb_{Connections} \geq 2$)* **then**
      Add V_i *to* $\mathbb{B}ridges$
      Remove V_i *from* $\mathbb{O}utliers$
    end
  end
end
Send $\mathbb{C}$lusters,$\mathbb{B}$ridges,$\mathbb{O}$utliers to the master using Worker2Master message

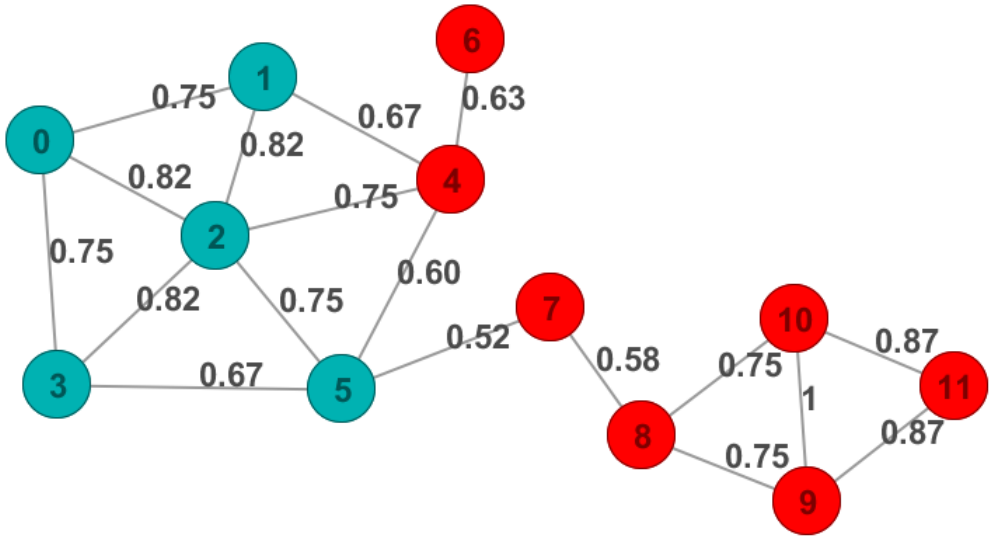


Figure 4.2: Illustrative example

and $P2$ (vertices in red), like it is depicted in Fig. 4.2, and each partition has a set of vertices connected with other partitions. We call them frontier vertices of a given partition P, and they are denoted by V_P^f. For example, V_{P1}^f={1,2,5} and V_{P2}^f={4,7}. Each $v \in V_P^f$ has a set of internal neighbors and external neighbors. For example, vertex 4 is a vertex of the partition $P2$. It has the vertex 6 as internal and the vertices 1,2 and 5 as external neighbors. Computing the structural similarity of a frontier vertex u considers that all the neighbors of u belong to the same partition. Thereby, we duplicate all frontier vertices in partition P to all neighbor partitions and we call them external vertices. For example in our running example, $P1$ has frontier vertices V_{P1}^f={1,2,5} and $P2$ is the neighboring partition. Thus, we must duplicate V_{P1}^f into $P2$ to ensure the accuracy of structural similarity of V_{P2}^f (see Fig. 4.3).

After that, when we apply a local clustering on $P1$ and $P2$, we will find several conflicts such as the vertex $v \in V_{P1}^f$ is a core vertex in $P1$, and an outlier in $P2$. These conflicts should be avoided in the merging step.

Step 2: Merging. The distribution of similarity computation and the local clustering step can improve the response time of our proposed DSCAN, compared to the basic sequential algorithm. However, we should take into consideration

the exactness of the returned results compared to those of the basic SCAN. To ensure the same result of basic SCAN, we defined a set of scenarios regarding the merging step. These scenarios will repetitively be applied to every two partitions of G, until combining all the partitions (see Algorithm 3, merging step). For each pair of partitions P_i and P_j, a merging function is executed to combine the local results from P_i and P_j and store them in global variables like clusters, borders, bridges and outliers. Algorithm 3 also achieves several scenarios (Lemmas 4.2.1, 4.2.2 and 4.2.3) to solve the encountered conflicts, mentioned below:

Lemma 4.2.1 ***(Merging local clusters)*** *Let $\mathbb{C}_1$ and $\mathbb{C}_2$ two sets of local clusters in different partitions P_1 and P_2, respectively. $\exists c_1 \in \mathbb{C}_1$ and $\exists c_2 \in \mathbb{C}_2$, $Core(c_1) \cap Core(c_2) \neq \emptyset$.*

Proof 1 *Let C_i be a cluster that groups a set of border and core vertices. If C_i shares at least one core vertex c with another cluster C_j, then c has a set of borders in C_i and C_j, and all the vertices in C_i and C_j are reachable from c. Hence, C_i and C_j should be merged into the same cluster.*

Lemma 4.2.2 ***(Outlier to Bridge)*** *Let $\mathbb{C}_1$ and $\mathbb{C}_2$ two sets of local clusters in partitions P_1 and P_2, respectively. $\exists$ C_1 and C_2 two clusters that belong to the two different sets of clusters $\mathbb{C}_1$ and $\mathbb{C}_2$. In addition, $\exists$ o an outlier in both partitions P_1 and P_2, with $N(o) \cap c_1 \neq \emptyset$ and $N(o) \cap c_2 \neq \emptyset$*

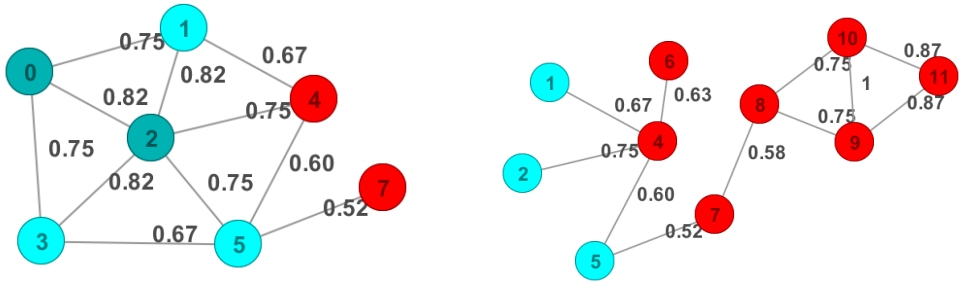
Proof 2 *If C_i and C_j $(i \neq j)$ share an outlier o, this means that o is weakly connected with the two clusters C_i and C_j. Hence, according to Definition 4.1.7, o should be changed to a bridge vertex.*

Lemma 4.2.3 ***(Bridge to Outlier)*** *Let c_1 and c_2 two local clusters in the partitions P_1 and P_2, respectively.*
$\exists$ a bridge b according to only two clusters c_1 and c_2, when c_1 and c_2 two clusters that will be merged into one cluster, b should be changed into an outlier.

Proof 3 *Let C_i and C_j two clusters that have a set of vertices (borders or cores), and a set of bridges with other local clusters, and $\exists$ b a bridge vertex according to*

the two clusters C_i and C_j only, where $i \neq j$. In the merging step and according to Lemma 4.2.1, if one or several clusters share at least a core vertex, then they will be merged into a single cluster. In this case, $(C_i,C_j) \Rightarrow C$ which makes b be weakly connected with only one cluster C, then according to Definition 4.1.7, b should change its status from bridge to outilier.

For instance looking at lines 26 to 28, we have focused on the shared cores between two clusters and the case when they share at least one core. According to Lemma 4.2.1, we should merge them into one single cluster. Subsequently, in lines 32-36, we verify for each outlier if it does not belong to some sets of cores, borders or bridges. In this case, we must remove it from the list of outliers. Otherwise, a vertex v should be changed as a bridge if it is classified as an outlier in the two clusters C_i and C_j that are not in the same partition, and if it has two connections with different clusters in the merging step.



(a) Partition 1

(b) Partition 2

Figure 4.3: Partitioned graph G used in the running example in Section 4.1.2

4.2.2.1 Illustrative example

Fig. 4.3 shows a demonstrative example of a graph clustering using DSCAN. In this example, we use the same parameters (ε and μ) of the running example in Section 4.1.2, and two partitions ($P1$ and $P2$). In the first step of DSCAN, the input graph G is divided into two partitions $P1$ and $P1$ as it presented in Fig. 4.3, where the blue vertices represent the first partition and the red vertices represent the second partition. In the second step, DSCAN duplicates the frontier vertices

in each pair of adjacent partitions. For example, the blue vertices 1,2 and 5 are duplicated in partition $P2$ since they represent frontier vertices in their partition. In the same way, the vertices 4 and 7 are duplicated in the partition $P2$. In the third step, all the workers perform the similarity computation, check the status for each vertex and build the local clusters, as it is shown in Fig 4.3. The last step of DSCAN consists of combining all local results returned by each worker. As shown in Fig. 4.3, there are some conflicts in terms of node status. For example, vertex 2 is a core vertex in $P1$ whereas it is a noise (outlier) vertex in $P2$. In the same way, vertex 4 is a border in $P1$ and a noise in $P2$. Then, after building the local clusters, $P1$ has one cluster (1,2,3,4, and 5) in which vertex 7 is a noise vertex in $P2$. As for $P2$, it groups the vertices 8,9, 10 and 11 as a cluster and the remaining vertices (4,5,1,2,7 and 6) are considered as noise vertices including vertex 7. This latter is a bridge according to basic SCAN (see running example in Section 4.1.2)). In the merging step, DSCAN considers that the vertex 7 has two weak connections with two different clusters. Thus vertex 7 is marked as a bridge.

4.2.2.2 Time complexity analysis of DSCAN

Let $\mathbb{E}$ be the set of all edges (internal and cut edges), $\mathbb{V}$ be the set of all vertices(internal and external) and the set of core vertices is $\mathbb{V}_c$. The time complexity of DSCAN is like basic SCAN, and can be divided into three step, (i) the structural neighborhood step, when DSCAN according to Definition 4.1.1 aggregates for each vertex its list of neighbors. Thus, the complexity is of the order of $O(|\mathbb{E}|)$. (ii) the time complexity of the computation of structural similarity which is defined by Definition 4.1.2. In this step, DSCAN enumerates for each edge (u,v) the set of common neighbors. Therefore, the time complexity is $O(|\mathbb{E}|)$. (iii) DSCAN performs the clustering step using the μ in order filter the core vertices, and groups those which have strong connections into a same cluster. Like this, the complexity is $O = (|V_c| - |(v_i,v_j)|_{i,j=1}^{|V|}, v_i, v_j \in V_c$ and $\sigma(v_i,v_j) \geq \varepsilon)$

4.2.2.3 Communication complexity

Let $\mathbb{S}$ be the set of slave machines, the communication complexity of DSCAN is the number of messages sent in the network during the merging step of DSCAN. This complexity is depended on the message type of BLADYG [Aridhi 2017] (Worker2worker, worker2Master). In the worker2Worker message the communication complexity is $O= |\mathbb{S}|^2$, and in the worker2Master message the complexity is $O= |\mathbb{S}|$.

4.3 Experiments

In this section, we present an experimental study and we evaluate the effectiveness and efficiency of our proposed algorithm for structural clustering of large distributed graphs.

4.3.1 Experimental protocol

We implemented DSCAN on top of BLADYG framework [Aridhi 2017], which was designed to build graph processing algorithms for large graphs. In the first experiment part, we compared DSCAN with four existing structural graph clustering algorithms, and then we evaluate some parameters of features of DSCAN :

1. Basic SCAN[1].
2. pSCAN: a pruning SCAN algorithm[2].
3. AnyScan: a parallel implementation of basic SCAN using OpenMP library[3].
4. ppSCAN: a pruning and parallel SCAN implementation[4].

[1]https://github.com/eXascaleInfolab/pSCANdeploymet
[2]https://github.com/RapidsAtHKUST/ppSCAN/tree/master/SCANVariants/anySCAN
[3]https://github.com/RapidsAtHKUST/ppSCAN/tree/master/SCANVariants/anySCAN
[4]https://github.com/RapidsAtHKUST/ppSCAN/tree/master/ppSCAN-release

Dataset name	Number of vertices	Number of edges	Diameter	Avg. CC
G1: California road network	1 965 206	2 766 607	849	0.04
G2: Youtube	1 134 890	2 987 624	20	0.08
G3: Orkut	3 072 441	117 185 083	9	0.16
G4: LiveJournal	3 997 962	34 681 189	17	0.28
G5: Friendster	65 608 366	1 806 067 135	32	0.16

Table 4.1: Graph datasets

The above mentioned algorithms are implemented with C language. Thus, we used the GCC/GNU compiler to build their binary versions. The compared algorithms are divided into two categories: (1) centralized algorithms such as SCAN and pSCAN, and (2) parallel algorithms such as AnyScan and ppSCAN. To run both centralized and parallel algorithms, we used a *T3.2xlarge* virtual machine on Amazon EC2. This machine is equipped with an 8 vCPU Intel Skylake CPUs at 2.5 GHz and 32 GB of main memory on a Ubuntu 16.04 server distribution. In order to evaluate DSCAN, we used a cluster of 10 machines, each of them is equipped with a 4Ghz CPU, 8 GB of main memory and operating with Linux Ubuntu 16.04.

4.3.2 Experimental data

For all test we used real-world graph datasets (see Table 4.1), these graphs are obtained from the Stanford Network Analysis Project (SNAP) snap.

4.3.3 Experimental results related to DSCAN

Speedup. We evaluated the speedup of DSCAN compared to the basic SCAN and its variants presented in Section 4.3.1. The compared algorithms use different graph representations. In fact, AnyScan and SCAN implementations use the adjacency list representation [Doerr 2007], whereas both pSCAN and ppSCAN use the Compressed Sparse Row (CSR) format [Dâ^{TM}Azevedo 2005]. In our proposed

algorithm, we used an edge list format, in which each line represents one edge of the graph. The incompatibility of the graph representations poses an additional transformation cost while evaluating the studied methods. For example, the transformation of the live journal graph from edge list to adjacency list takes around 100 seconds using one machine equiped with an 8 vCPU and 32 GB of main memory.

Fig. 4.4 shows the runtime of the studied approaches with different datasets.

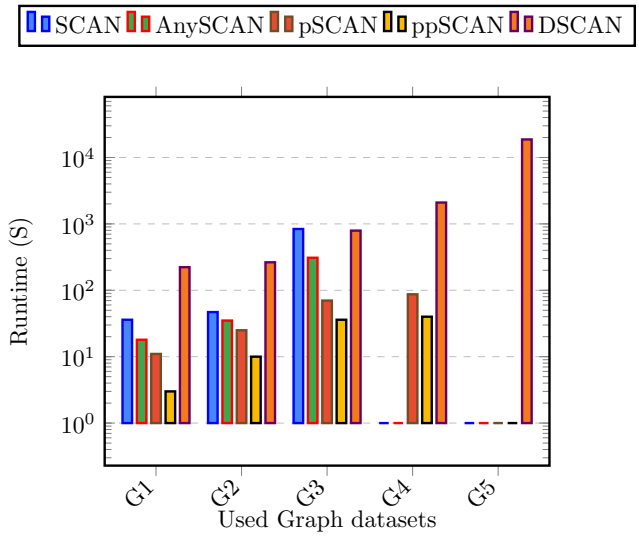


Figure 4.4: Impact of the graph size on the processing time of both SCAN and DSCAN

As shown in Fig. 4.4, our approach is slower than the other algorithms in the case of small graphs (G1,G2,G3) and there was a very large gap between DSCAN and the other algorithms especially with pSCAN and ppSCAN. On the other hand, this gap become reduced when the graph size increases (case of G4 dataset). The plots bars in Fig. 4.4 shows a gap of 12x between DSCAN and basic SCAN with the G1 dataset and 2x only with the G4 dataset. We notice that the gap between DSCAN and ppSCAN depends mainly on the size of the used dataset. For example, with the G1 dataset, the gap between DSCAN and ppScan is about 20x, while this gap is reduced to 11x for the G4 dataset. This can be explained by reaching the pruning step of pSCAN, which exempts several similarity computings

during the clustering step. It is important to mention that DSCAN is a distributed implementation of SCAN and the other studied algorithms are centralized. This leads to additional costs related to data distribution, synchronization and communication. Fig 4.4 also shows that with the modest hardware configurations, only DSCAN can scale with large graphs like the G5 dataset.

Scalability. The main goal of this experiment is to evaluate the horizontal scalability of our algorithm. We used two graphs (LiveJournal and California road network). We set the values of μ and ε to 3 and 0.5 respectively, and we varied the number of machines, with the goal to measure the response time for each size of the cluster. It can be clearly seen, from Fig. 4.5, that our algorithm is horizontally scalable, which was not guaranteed by the other algorithms, as discussed in the previous chapter. Fig. 4.5 also shows that the running time will decrease depending on the number of machines in the cluster. When we add a new machine to the cluster, the running time becomes smaller. As depicted in Fig. 4.5, the red curve (LiveJournal graph) shows a significant improvement of about 82% in the response time, when the number of machines reaches 10. We also notice a weak improvement, according to the number of machines in the cluster, when we have a small graph. This is the case of California road network which is smaller than LiveJournal graph. The red curve's behavior can be explained by the splitting of

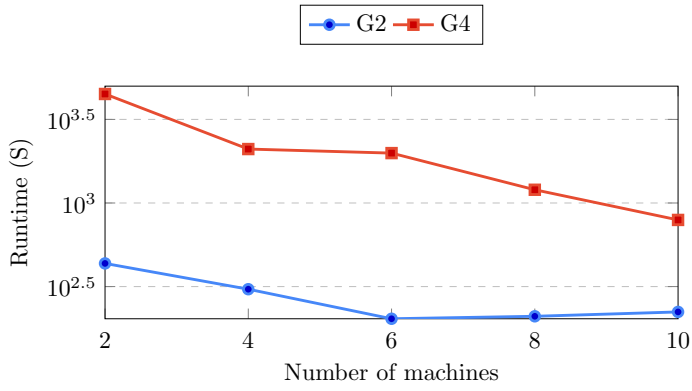


Figure 4.5: Impact of the number of workers on the running time (with $\varepsilon = 0.5$, $\mu = 3$).

the input graph into small sub-graphs and by performing a local clustering on each

sub-graph, which reduces the global response time even with the additional cost of aggregating the intermediate results returned by each machine in the cluster. That was not the case with the blue curve, where we have an improvement of about 50% using a cluster of 10 machines, compared to the results using a 6-machine cluster. We noted that the curve starts to increase when the cluster size exceeds 8 machines. This is due to the communication in the shuffling step.

Impact of ε value on DSCAN. The numbers of clusters, bridges and noise vertices depend on the values of ε and μ. In ppSCAN, when we decrease the value of μ, the running time increases, as the non-pruned edges are increased. Similarly in this experiment, we evaluated the impact of ε value on the running time of DSCAN. For this, we varied the value of ε from 0.2 to 0.8 for different graph data sets. As shown in Fig. 4.6, the response time is slightly dependent

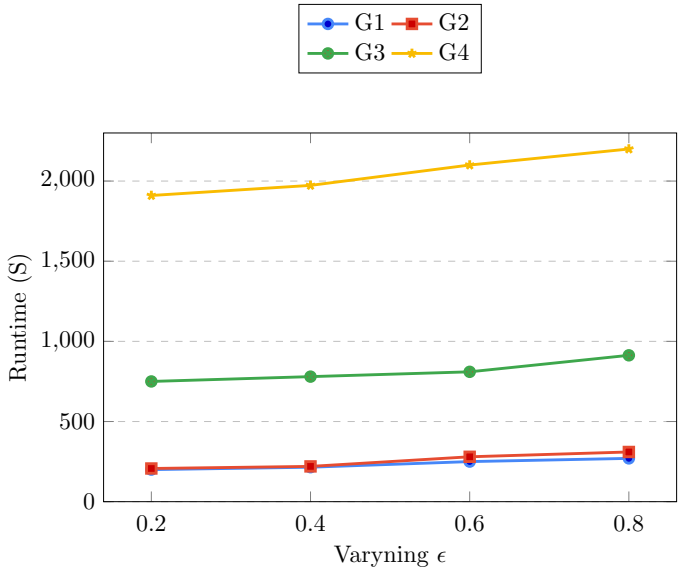


Figure 4.6: Impact of ε size on the running time of DSCAN

on the value of ε, especially with large graphs (G3 and G4). When we increase the value of ε the response time increases. Overall, the observed behavior can be explained by the merging step in DSCAN algorithm, since we did not see the impact of ε on the previous steps (graph loading and local clustering). In the graph loading step, we do not use this value and in local clustering we use the basic

SCAN clustering which is not dependent to ε. That is why the impact of ε on the running time can be explaind by the merging step. In fact, when the value of ε increases, the number of outliers becomes larger. Also, in the merging function (see Algorithm 3, lines 17-37), DSCAN combines the local results by starting to check the clusters that share almost one core, in order to merge them. Then, it verifies for all outliers if they are bridges or cores. Hence, outliers' checking requires more communication between the workers, which increases the respense time.

Evaluation of DSCAN steps: DSCAN algorithm is based on four main steps. In each step, DSCAN performs a specific treatment with different costs in terms of running time. To study the running cost of each step in DSCAN, we used four graph datasets and we fixed the values of ε and μ to 0.5 and 3, respectively. Fig.

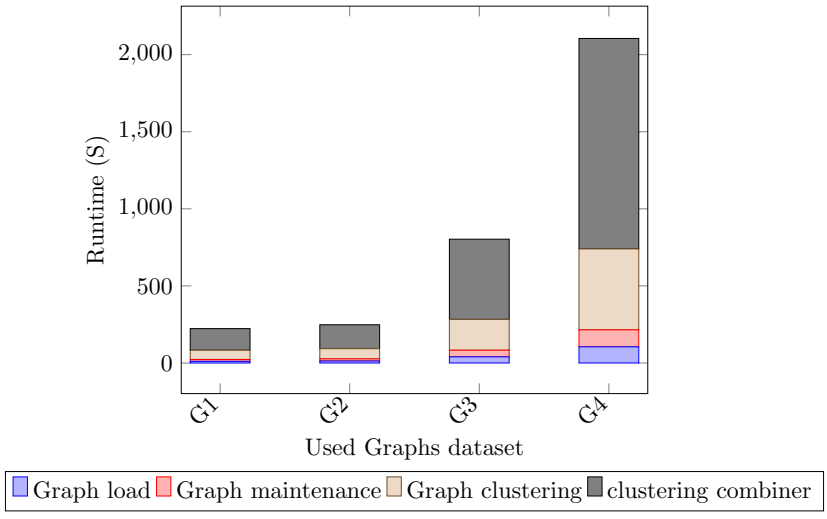


Figure 4.7: Performance of DSCAN phases

4.7 shows the response time of each step in DSCAN. The merging function is the most expensive step that makes DSCAN slow, compared to the other algorithms. It takes more than 50% of the global running time with all the used graph datasets, while the clustering step takes about 30% only. The rest of computation time is devoted to the graph loading step and the duplication of vertices in frontiers to ensure the consistency property, as we discussed in Section 4.2.1. This disparity

can be explained by the communication between machines during the aggregation of local results returned by each machine. Consequently, communication in DSCAN must be improved because the clustering step takes a little time which can make DSCAN faster.

Imapct of the graph partitioning on DSCAN. In our vision, the partitioning step has a direct impact on the response time of DSCAN. For this reason, we randomly generated four partitioning schemas, and for each one, we got the number of cut-edges as follows: 17.6M, 19.2M, 22.3M and 24.6M for the partitions P1, P2, P3 and P4, respectively. Then, we run DSCAN on all partitions with the same configuration (10 machines, ε=0.5 and μ=3). As shown in Fig. 4.8, there is a

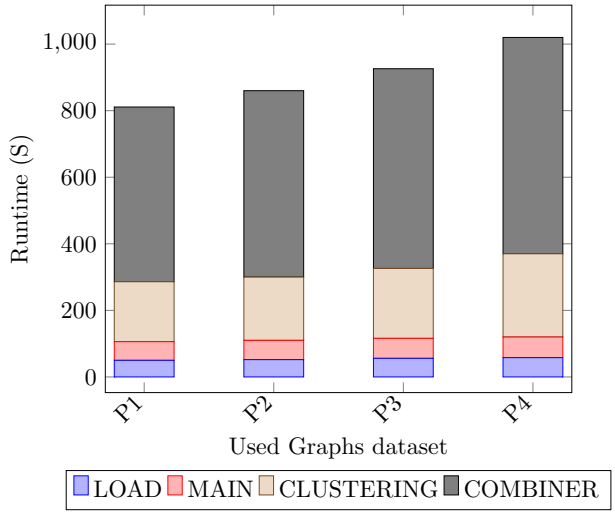


Figure 4.8: Impact of the partitionning step on DSCAN response time (ε=0.5, μ=3)

very clear impact of the graph partitioning on the response time of DSCAN, as this latter rises from nearly 800 to 1000 seconds with P1 and P2. Furthermore, the elapsed time of each DSCAN step varies from one partitioning to another. This disparity is noticed mostly during the merging step and slightly in the clustering step. This is probably explained by the number of vertices duplicated due to the number of cuts-edges produced by the graph partitioning. This number would affect the amount of similarity computing operations in the clustering step, and

increases the communication cost during the merging step.

4.4 Conclusion

In this chapter, the second contribution is presented. This proposed algorithm (DSCAN) is a distributed algorithm for big graph clustering based on the structural similarity. DSCAN is based on a distributed and master/slaves architecture which makes it scalable and works on the community of modest machines. In this chapter, Firstly, an overview on the structural graph clustering is presented. Secondly, we have presented the main steps of DSCAN, starting from the partitioning to the combining of intermediate results for each worker. Finally, we have performed an extensive experimentation about our proposed algorithm compared to other ones. The experiments have shown that DSCAN ensures the horizontal scalability, which is not guaranteed with other similar algorithms.

Key points

- We presented a new distributed graph partitioning based on a big graph processing framework.
- We proposed a novel distributed algorithm for the detection of clusters, bridges and outliers in these of large-scale networks.
- An experimental study to evaluate the distributed algorithm for the detection of clusters, bridges and outliers on large-scale networks.

Publications

- W. Inoubli, S. Aridhi, H. Mezni, M. Maddouri, E. M. Nguifo. Un algorithme distribué pour le clustering de grands graphes. Extraction et Gestion des Connaissances (EGC 2020), Bruxelles, Belgique.
- W. Inoubli, S. Aridhi, H. Mezni, M. Maddouri, E. Mephu Nguifo. A Novel Scalable Clustering Method for Distributed Networks. **submitted to Information Systems**.

Chapter 5

Distributed and Incremental Structural Graph Clustering Algorithm

Contents

5.1 Introduction to dynamic graphs 107

5.2 Dynamic graphs: definitions 107

5.3 DISCAN: Proposed incremental graph clustering 108

5.3.1 Distributed graph update 109

5.3.1.1 Adding a new vertex 109

5.3.1.2 Adding a new edge 110

5.3.1.3 Delete a vertex 110

5.3.1.4 Delete an edge 112

5.3.2 Distributed graph clustering 114

5.3.3 Time complexity analysis of DISCAN 119

5.3.3.1 Communication complexity 119

5.4 Experimental result 119

5.5 Conclusion 125

Goals The real-world applications that use a graph as data model, and perform a graph clustering should support dynamic graphs. The existing methods consists to re-executed a clustering algorithm in each new update. However, this approach

can suffer from the graphs size, which spends a lot of time to get a clustering result in each updating. Accordingly, in this chapter an incremental algorithm is proposed and named DISCAN. Thus, this chapter introduces the concepts of dynamic graphs through some definitions, after that a distributed graph maintenance and an incremental graph clustering are presented. Finally, in order to present the efficient of DISCAN, an experimental study is presented at the end of this chapter.

Keywords: Big and dynamic graph, Dynamic graph clustering, Dynamic community detection, Graph analysis, incremental graph processing.

5.1 Introduction to dynamic graphs

Due to the complexity of processing of big generated data in several applications, the graph data model has been proposed for several applications, and it has proven to be an effective model in some use cases like research networks (e.g., DBLP), social networks (e.g., Facebook and Twitter), sensor networks (e.g., Internet of Things). Furthermore, in real-world applications, the data naturally is not static. Therefore, graphs are models for encoding the data including the dynamic data, thus dynamic graphs have been introduced.
This new data model has been the main goal of several research projects, and it introduces new challenges that are mainly related to data velocity and data volume (graph). Additionally, several definitions defined the dynamic graphs, and these definitions are related to applications. In the next section, we refer to a set of definitions that are related to some applications and use cases to understand the concept of dynamic graphs.

5.2 Dynamic graphs: definitions

In the literature, authors propose several definitions and formulations. In this context, a survey in [Rossetti 2018] presented some definitions and representations on the graph relatively to the community detection use case.. Through Fig.5.1, four graph representations are illustrated, and in the next, we provides some definitions related to each representation.

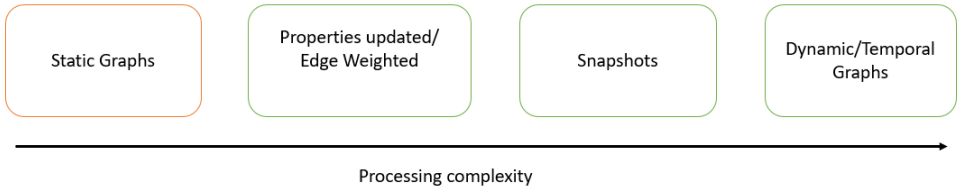


Figure 5.1: Different graph representations from static to dynamic graph

Definition 5.2.1 *(Updated proprieties/Edge Weighted) Properties updated is a dynamic graph variant, where the dynamism aspect is observed only on the graph*

properties. For example, in the case of attributed graph, attributes's values of edges/vertices can be changed at each time while maintaining the initial graph structure. Another example, is the case of simple graph the dynamism that can be observed on the edges Weight updating over time.

Definition 5.2.2 *(Snapshots) Graph snapshots type is represented as an history of graph snapshots over time or at each period. Thus, each snapshot is a graph at a time t or at a period p. In the snapshot graph case, from on snapshot to another, several updates can be observed. Then, a snapshot graph G can be defined by an ordered set $<G_1, G_2 \ . \ . \ .Gt>$ where each snapshot $G_i = (V_i , E_i)$ is univocally identified by the sets of nodes V_i and edges E_i.*

Definition 5.2.3 *(Dynamic/Temporal Graphs) A dynamic or temporal graph is a graph which undergoes several updates over time. Unlike properties updated graph, the updates over time can be structural or semantic. Where, the structure update is represented by the vertices and/or edges my be appear or disappear in each update, and the semantic updates are the properties or edge weight updates.*

5.3 DISCAN: Proposed incremental graph clustering

Consider a large and dynamic graph G, which will undergo several changes over time such as adding or deleting edges or/and vertices. When a clustering is performed, the classic approaches re-run the clustering from scratch, which is obviously very expensive especially in large graphs. In this work, we propose an incremental and distributed algorithm for large and dynamic graph clustering. The main idea of our approach is to determine in each update the vertices and the edges that are concerned by this update, and according to that, we apply the new updates on the old clustering results.

5.3.1 Distributed graph update

5.3.1.1 Adding a new vertex

In this section, we describe the scenario at the adding of a new vertex. In this context, several situations may be encountered, such as a separated vertex, vertex connected to a single existing vertex and vertex connected to several existing vertices. To minimize the task complexity, we have standardized this action to a single operation that will take into consideration all the discussed situations. In the case of several edges must be added to the graph and in the same time, we split them into a set of a simple edge as follows (V_{new}, V_{old}), where V_{new} is a new vertex to be added and V_{old} an existing Vertex.

As we discussed in the partition step, we have two types of vertex in each partition: internal vertices and external vertices. Then it is required to take into consideration the vertex type in the adding vertex operation. For more details, the algorithm 10 illustrates the fundamentals steps of an adding of a new vertex.

This operation starts by sending the tuple (V_{new}, V_{old}) by the master machine to

Algorithm 4 Adding a new vertex

Input : V_{new} V_{old}
Output: $G \cup V_{new}$

2 *Master machine: send (V_{new} V_{old}) to all workers*
  `/* In parallel */`
3 **foreach** *Worker* $W_i \in \mathbb{W}$ **do**
4 **if** *($V_{old} \neq null \cap V_{old} \in internalVertices \cap V_{old} \neq \in frantierVertices$)* **then**
5 *Add a new vertex V_{new}*
  Add V_{new} as a neighbor of V_{old} vertex
  Add a new edge (V_{new}, V_{old})
  Update the similarity of all affected edges
6 **end**
7 **else if** *($V_{old} \neq null \cap V_{old} \in internalVertices \cap V_{old} \in frantierVertices$)* **then**
8 Add V_{new} as a neighbor of V_{old}
  Update the similarity values of all affected edges
9 **end**
10 **end**

all worker machines (line 2). Thereafter, in each partition, the workers check if the V_{old} is an internal or frontier vertex (if $V_{old} \in$ the current partition). According to vertex type, each worker applies a specific treatment like as illustrated in the algorithm 10. In first case (line 4 to 6), when the V_{old} is an internal vertex, we add the v_{new} into the partition, add v_{new} as a neighbor of V_{old}, add a new edge and update the structural similarity of affected edges. In the other case, when the V_{old} is a frontier vertex, the worker must only add the v_{new} a neighbor of V_{old} and update the structural similarity of affected edges.

5.3.1.2 Adding a new edge

When we will add a new edge E_{new}, it is necessary to check if this is an internal or a cut edge. In the case of an internal edge, E_{new} is an edge which links two internal vertices, and the cut edge is an edge that links two vertices in different types (internal/frontier). Consequently, these two type of edges need a special treatment for everyone. To show all these details, algorithm 5 explains these main steps. After that the master machine sends the E_{new} to all worker machines. The latter check the type of E_{new}, if it is an internal or a cut edge. Thus, in the case of an internal edge, local changes will be performed in the partition that contains the E_{new}, as described in the algorithm 5 in line 3 to 6. Whereas in the second case, E_{new}=(v_1, v_2) i.e. v_1 and v_1 are in different partitions P_1 and P_1 respectively. Therefore, the algorithm adds v_1 as an external vertex in P_2 and v_2 as an external vertex in P_1 (see algorithm 5 lines 7-9).

5.3.1.3 Delete a vertex

In a local graph, a deleting of a vertex can be affected by several vertices and several structural similarities. In our case, when we have a distributed graph, several partitions can be affected by the vertex deleting, since one vertex can exist on several partitions but in different types: internal vertex, frontier vertex, and external vertex. Each vertex type requires a special treatment. The algorithm 6 presents all possible cases when deleting an existing vertex. In the first step, when the master receives a request to delete a vertex, it sends the vertex to all workers

Algorithm 5 Adding a new edge

Input : E_{new}=(V_1, V_2)
Output: $G \cup E_{new}$

2 *Master machine: sent the E_{new} to all worker machines*
  /* In parallel */
3 **foreach** *Worker* $W_i \in \mathbb{W}$ **do**
4 **if** *($V_1 \in \mathbb{I}nternal$ and $V_2 \in \mathbb{I}nternal$)* **then**
5 *Add V_1 as a neighbor in V_2*
  Add V_2 as a neighbor in V_1
  Add E_{new} into current partition
  Update the similarities of all affected edges
6 **end**
7 **else if** *($V1 \in \mathbb{I}nternal$)* **then**
8 *Add V_2 as a neighbor in V_1*
  Add E_{new} into current partition
  Update the similarities of all affected edges
  Sent V_1 to all Workers: with AddExternalVertex message
9 **end**
10 **end**
/* if any worker receives *AddExternalVertex* message with V_{from} parameter, it saves V_{from} as an external vertex */

to delete it. Then in the second step, each worker starts by checking if the

Algorithm 6 Delete a vertex

Input : V
Output: $G \setminus V$

```
2  Master machine: send the vertex V (that will be deleted) to all worker machines
       /* In parallel                                                    */
3  foreach Worker W_i ∈ 𝕎 do
4      if ((V ∈ ℙ_internal) and (V ∈ ℙ_frantier)) then
5          Remove all external vertices associated with V
6      end
7      else if ((V ∈ ℙ_external)) then
8          Remove all external vertices associated with V
           Remove all vertices associated with V from P_frontier
9      end
10     Remove V
       Remove all edges associated with V
11 end
```

requested vertex is belonged to the partition and performs a specific treatment for each case like detailed in algorithm 6 (lines 3-9). When V is a frontier vertex to be deleted, if V a frontier vertex, we delete all external edges that are associated only with it. Else, if V is an external vertex we delete it from all list of neighbors of each vertex contains V as a neighbor.

5.3.1.4 Delete an edge

Like deleting of a vertex V, when we will delete an edge E, two cases occures according to the type of the latter (internal or cut). Each type should be performed with a special treatment as explained in Algorithm 7. Therefore, the master sends to all workers the edge $E = (v_1, v_2)$ and asked them to delete it. Thus, each worker asks them to check if E is an internal or cut edge. Like in the algorithm 7 lines (3-6), when both vertices are in the same partition, then $V1$ must be removed from a list of neighbors of $V2$ and the same thing with $v2$. In the other case, where E is a cut edge and v_1 has belonged to the current partition (and v_2 in other partition) lines(7-9). Thus remove v_2 from a list of neighbors of v_1, and

remove v_1 from list of frontiers vertex if do not have any other connection with an external vertex. Then remove v_2 from the list of external vertices if and only if v_2 did not have any connections with an internal vertex.

Algorithm 7 Delete an edge

Input : (V_1, V_2)
Output: $G \setminus (V_1, V_2)$
2 Master machine: send the edge $E = (V_1, V_2)$ will be deleted to all worker machines
/* In parallel */
3 **foreach** *Worker* $W_i \in \mathbb{W}$ **do**
4 **if** $(E \in Edges_{internal})$ **then**
5 *Remove* v_1 *from* $N[v_1]$
Remove v_2 *from* $N[v_2]$
6 **end**
7 **else if** $((E) \in CutEdges and v_1 \in PInternal)$ **then**
8 Remove v_2 from $N[v1]$
Remove $v1$ from frontier vertices: (if do not have any other connection with an external vertex)
Remove $v2$ from external vertices: (if do not have any other connection with an internal vertex)
Remove (v_1, v_2) from Cut edges list
9 **end**
10 **else if** $((v_1, v_2) \in CutEdges and V_2 \in PInternal)$ **then**
11 Remove V_1 from $N[v2]$
Remove $V2$ from frontiers vertices :(if do not have any other connection with an external vertex)
Remove $V1$ from external vertices :(if do not have any other connection with an internal vertex)
Remove (V_1, V_2) from Cut edges list
12 **end**
13 Remove (V_1, V_2) from all edges
end

In the case when v_2 is belongs to the current partition, we perform the same steps like it is presented between line 10 and 13 in the algorithm 7. In the last step, for all cases (internal or cut-edge) we remove E, update the concerned edges

according to the deleting of E and get all affected vertices in order to sent them to master.

5.3.2 Distributed graph clustering

Assume that $G = (V,E)$ is going to undergo several updates U over time. U can be adding/deleting vertices or edges from/to G. These changes can affect the similarity values and some vertex status (borders, core, etc). Consequently, the clustering schema may be affected in several situations.

Definition 5.3.1 *Each vertex $u_i \in U$ generates modifications of several structural similarities, which is defined as the affected edges E_A. These vertices also change the status of several vertices and are noted as affected vertices V_A.*

Definition 5.3.2 *(Affected edges) When adding a vertex V_{new} associated with an existing vertex V_{old}, new edge (V_{new}, V_{old}) will be created and a set of edges denoted E_A will be marked as affected edges and should be updated. Note that $e \in E_A$ and $e = (u,v)$, where $u \in V_{old}$ and $v = v_{new}$.*

In the case of removing a vertex V_{TBD}, a set of edges noted as E_{TBD} should be deleted from G. Therefore, a V_c of vertices is concerned by this update. Here $V_c = (u,v) \in E_{TBD} \setminus V_{TBD}$, and according to V_c, the affected edges are $E_A = E_i \subset G$ and $E_i = (u,v)$ / {u,v} $\in V_c$.

Also when adding or deleting an edge (v_i, v_j), the list of neighbors of both vertices v_i and v_j will be changed which can produce several other edges. The affected edges update their structural similarities. E_A is a set of edges $(u,v) \in E$ and $v_i = u$ or $v_j = v$

Definition 5.3.3 *(Affected Vertices) For each update on G, some structural similarities can be changed. Thus, some vertices change their status (border, core vertices). These vertices are defined as affected vertices V_A. V_A can be divided into two sub-sets: (i) directly affected V_{AD}, which update their status depending directly on E_A, and are defined as V_{AD}, (ii) indirectly affected V_{ID} which depend on the status of V_{AD}. For example, when a core vertex changes his status to outlier*

or border, then all these border vertices will become affected indirectly according to this update. Let E_A=(v_1, v_2) be following an update V_{AD} =$V_{AD} \cup v_1 \cup v_2$, while V_{AI} are all neighbors of V_{AD}. Other vertices can be affected indirectly with several updates, and are dependent on the changed clusters. These vertices are defined the affected bridges and noted as V_{AB}. V_{AB} is the list of outliers or bridges which are connected with affected clusters described in the Definition 5.3.4.

Definition 5.3.4 *(Affected clusters) The affected vertices in several situations can affect the clustering schema. The Affected (concerned) clusters represent the sub set of clusters which contain one or more affected vertices, denoted by $C_{affected}$. $C_{affected}$=$c \in \mathbb{C}$, $\exists v \in (v_{AD} \cup v_{ID})$ and $v \in c$.*

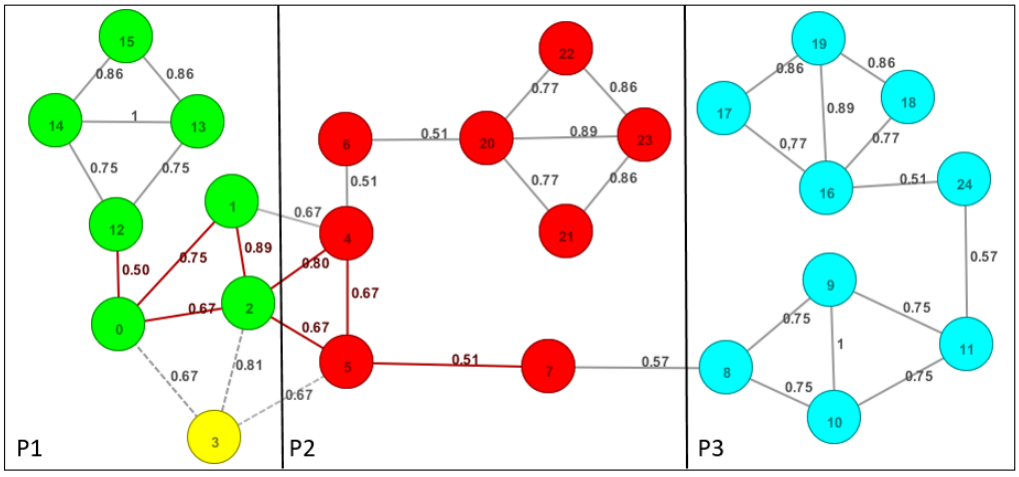


Figure 5.2: Partitioned graph G (P1, P2, P3), affected vertices, edges when deleting vertex 3

Lemma 5.3.1 ***Change of structural similarity of affected edges*** E_A *are mainly depend on the Definition 4.1.1. The similarity value is based on the neighbors of both vertices of an edge. Therefore, each update (adding/deleting an edge/vertex) can change the list of neighbors of some vertices. As a result, some edges must change their structural similarity.*

Lemma 5.3.2 ***Change in the status of affected vertex*** *According to the Definition 4.1.4 and the Definition 4.1.3, for some updates in the structural similarity, when the similarity value exceeds or decreases the threshold ε the status of this edge maybe changed to strong or weakly connection. Thereby, this change can affect the status of each vertex $v \in V_{AD}$. Consequently, according to 4.1.5 the V_DA can change other vertices status indirectly which are defined as indirectly affected vertices V_{IA}. Rationally, those changes can affect the mapping of the clusters, and several clusters may be changed. Thereby, according to Definition 4.1.7, $\exists$ a vertex $v \in$ borders list where v only depends on two clusters c_1 and c_2. When c_1 and c_2 will be merged into one cluster c_{new}, v will be weakly connected with only c_{new}. In this case and according to Definition 4.1.7 v will be an outlier vertex. In the same way, when we have a cluster c and v is an outlier to c with two weak connections related to v_1 and $v2$, after some updates c becomes splited to two clusters c_1 and c_2 when v_1 and v_2 are respectively in c_1 and c_2. In this case, v become a bridge between the new clusters c_1 and c_2 according to the connection of v with v_1 and v_2.*

Figure 5.2 shows an example with a graph G with 3 partitions for which we deleted the vertex 3 (yellow vertex). Then, all affected edges E_A (red edges) change their structural similarities since the vertices 0,2 and 5 (old neighbors of the deleted vertex 3) lost the vertex 3 as a neighbor, and all the edges associated with the vertex (3) are removed from G. This new update affects only a subset of vertices in G. The example shows that the vertex 2 ($v_2 \in v_A$) lost its core status, whereas vertex 9 keeps its core status ($v_9 \notin v_A$). Thus, E_A can change several vertices ' status directly (v_{AD}) or indirectly (v_{AI}). Like it is shown in our example, vertex 2 $\in v_{AD}$, it lost on strong connection with the removed vertex which changed to an outleir in this case. In the same way, v_{AD} can change the status of v_{AI}. Vertices 0,1,4 and 5 were considered as border vertices according to vertex 2 (in the initial graph), and in our case, when the vertex 2 lost the core status, as a result v_{AI} are changed to outliers. According to Definition 5.3.4, v_{AD} and v_{AI} can change the old clustering schema and our example confirm that. Partition 1 of G groups two clusters. The first one is $C_1 = \{$ 12,13,14$\}$ and the

second one is C_2 ={0,1,2,3,4,5} (4 and 5 are external vertices). Thus, $C1$ and $C2$ are affected because $C1$ contains the vertex 12 as an affected vertex, and in the $C2$, all its vertices are affected because of the removed vertex. These clusters can affect in the last step other vertices (bridge or outlier vertices). Like it is depicted in 5.2 C_2 should be removed since it does not have any core vertex, and the vertices 6 and 7 were considered as bridges with $C2$. Thereby, these bridges are susceptible to change their status.

Algorithm 8 describes the main steps of the proposed DISCAN. DISCAN provides real-time graph maintenance and a micro-batch clustering. Moreover, it performs the new clustering after several changes, in order to optimize the incremental clustering. We present below the main steps of DISCAN:

Step 1: Graph maintenance. In this step, DISCAN runs the graph maintenance in real time. In each update U, it (i) updates the graph G structure, (ii) checks the affected edges E_A, and (iii) recomputes the similarities of E_A (lines4-12 of the pseud-code 8). Then, it gets V_A according to E_A in order to memorize them in a global variable. Like discussed in Definition 5.3.4, V_A can affect several clusters in the current partition or in other partitions. Thereby, each worker shares its V_A with all workers so that each one determines the affected clusters and saves them. Once the affected clusters are fixed, they affect some bridges or outliers vertices. So the affected bridges should be added to the list of affected vertices to be checked in the next step.

Step 2: Incremental local clustering. This step is performed in micro-batch processing mode, after a number of updates or using a window time. In both methods, DISCAN runs the some scenarios. These latter consist of two choices for a user who selects one of them according to the number of changes per seconds or according to other needs such as the graph size, graph evolution. Therefore, in each batch, DISCAN first checks the core vertices. The checking is performed only on the affected vertices V_A, where in general the $|V_A| \ll |V|$.

Secondly, DISCAN checks the border vertices like in the first clustering but using only E_A, since the border vertices are dependent on the strong connections with core vertices. For this reason, DISCAN checks the core vertices according to E_A only.

After that, the updated cores and borders vertices will change the clustering schema depending of the affected vertices (cores, borders) like it is described in Definition 5.3.4. The affected clusters feat are four possible cases owing to each update of a graph: (1) split one cluster to small clusters, (2) remove an existing cluster, (3) build new clusters, (4) update the existing clusters and (5) merge two or more clusters. Consider set of cores and theirs borders, and the core vertices have strong connections beteen them. After the removal of a strong connection between two cores (c_1,c_2) and (c_1,c_2), then the clusters should be splited into sub-clusters depending on c_1 and c_2. For the case (2), each cluster is built on a list of cores and theirs borders. If one cluster does have any core vertex it should be removed. Sometimes, a new update makes an outlier vertex to a core vertex c_{new}. If this vertex has a strong connection with any other old core c, it joins the cluster of c, like in the case (4). If c_{new} does have any strong connections with any other core it builds a new cluster, like the case (3). In the last case (5), if there exist a new strong connection between two cores in two different clusters, then we merge these latter. In order to deal with all these cases, we remove all the affected clusters and re-build a new clustering in a same way of the first clusetring presented in Definition 4.1.6 and using only the affected vertices (cores and borders vertices). The new clustering is performed only on the changed vertices which will be very faster compared to the first clustering.
Finally, DISCAN uses the new clusters and the remaining affected vertices to check if they represent new border vertices.

Step 3: Merge the new updates. After each batch, each worker starts to merge the new updates (see Algorithm 8 lines 19-23). The master machine gets all the affected vertices from all workers in order to facilitate the combination of the new changes. Here some clusters should be verified. In each batch, the master keeps only the past unchanged clusters and requests all workers to get the changed clusters. Thereby, all workers combine the updated clusters eventually including new clusters. In the merging step, DISCAN uses the same scenarios as in the first clustering (see Lemma 4.2.1). If at most one cluster shares at least one core vertex, they can be merged into one cluster. In the next step, after getting the new clusters, the master machine requests all workers to get theirs borders

vertices. Then, each worker filters only the affected vertices which will belong to its local bridges, and sends them to the master. The rest of affected vertices will be added to the global list of outliers. Finally, DISCAN initializes the global affected vertices and global affected clusters to empty lists which will to be used in the next batch.

5.3.3 Time complexity analysis of DISCAN

The time complexity of DISCAN mainly depends on the affected edges and the affected vertices in each new update. Moreover, this complexity is very low compared to the initial complexity of DSCAN. Since the number of affected edges $|E_A| \ll |E|$ and the number of affected vertices $|V_A| \ll |V|$. In DISCAN, also the time complexity is based on three steps. (i) The graph maintenance, the complexity in this step is about $O = |E|/n$ in worst case and $O = 1$ in best case, where n is the number of partitions. (ii) In the similarity computation the complexity is $O = (min(|N(u)|, |N(v)|).|\mathbb{E}_A|)$, and (iii) also the clustering step manly depends on V_A. Thereby the incremental clustering complexity is about $O = (|V_c \in V_A| - |(v_i, v_j)|_{i,j=1}^{|V_A|}, v_i, v_j \in V_c$ and $\sigma(v_i, v_j) \geq \varepsilon)$.

5.3.3.1 Communication complexity

Let $\mathbb{S}$ be the set of slave machines, and $\mathbb{S}_\mathbb{A}$ the set of partitions (slaves) affected by graph updates. Moreover, this complexity is depend on the number of affected slaves. Thus, this complexity in the case of worker2worker message is $O= |\mathbb{S}|^2$, and in the worker2Master message the complexity is $O= |\mathbb{S}_\mathbb{A}|$.

5.4 Experimental result

SpeedUp. The main goal of this experiment is to compare DISCAN with other algorithms, Since there is no incremental algorithm for the structural graph clustering, we evaluated the speedup of DISCAN compared to the fastest existing algorithm (pSCAN and ppSCAN) which we presented in Section 4.3.1. Fig. 5.3 shows the running time of the tested algorithms with different graphs. In the first

Algorithm 8 Incremental and Distributed Clustering

Input : Initial graph G as a text file, parameter (NP number of partitions), a new update U

Output: $\mathbb{C},\mathbb{B},\mathbb{O}$

/* Global affected vertices and clusters in each worker */

$\mathbb{G}lobalAffectedVertices=\emptyset$

$\mathbb{G}lobalAffectedClusters=\emptyset$

/* Step 1: Graph maintenance */

/* Update the inital graph in each new update and get the affected vertices and the affected clusters */

Master machine: send a new update u to all workers

/* In parallel */

foreach $Worker_i$ $w_i \in \mathbb{W}$ **do**

  Update the current partition according to U

  Get E_A

  Recompute the similarities of E_A

  Get the immediately affected vertices V_I

  Share the immediately affected vertices V_I with all workers

  Get the indirectly affected vertices V_{Ind}, and the affected clusters C_A

end

/* Step 2: Local clustering schema maintenance */

/* Update the old local clustering schema according to the global affected vertices */

foreach $Worker_i$ $w_i \in \mathbb{W}$ **do**

  Check Core vertices from GlobalAffectedVertices

  Check Border vertices from GlobalAffectedVertices

  Check affected clusters

  Check Affected bridges

end

/* Step 3: Merge the new updates */

foreach $Worker_i$ $w_i \in \mathbb{W}$ **do**

  Merge all affected clusters from all workers

  Check new Bridge vertices according to the new clusters

  Check the remaining outlier vertices

end

/* Reset the global affected vertices and clusters in each worker */

$\mathbb{G}lobalAffectedVertices=\emptyset$

$\mathbb{G}lobalAffectedClusters=\emptyset$

Send $\mathbb{C}$,$\mathbb{B}$,$\mathbb{O}$ to master using Worker2Master message

time, we start by running all the algorithms with the used graphs, and in each moment, we added a new batch of updates in different sizes. In almost used graphs, DISCAN is slow compared to other algorithms, except for pSCAN which does not support the G5 graph. Despite that pSCAN and ppSCAN re-execute the clustering from scratch, they are faster than DISCAN in case of G2 and have the same response time in the case of G3. In the case of the G4, DISCAN is initially better than pSCAN and ppSCAN. Initially, ppSCAN and pSCAN are faster than DISCAN, but the gap in running time begins to narrow between G2 and G4. Then, when we add a new batch to the initial graphs, DISCAN becomes faster than pSCAN and ppSCAN. In the case of G3, Fig. 5.3 shows a gap of 2X between DISCAN and pSCAN and almost the same running time of DISCAN and ppSCAN. Furthermore, the gap between DISCAN and pSCAN starts to increase, to finally reach 3X, and 10% between DISCAN and ppSCAN.

DISCAN scalabilily. Fig. 5.4 shows the scalability of DISCAN w.r.t. the number of workers, with G2 and G3 datasets and for different batches of updates using the default parameters ε=0.5, μ=3.

Overall, the number of workers affects the running time in terms of size of the used dataset and the number of updates. In Fig. 5.4 with G2 dataset, all curves have almost the same look. They decrease for 2 to 4 workers, but with varying the degrees of improvement depending on the number of updates. These varying improvements are between 10% and 50%, respectively for 2000 and 10000 updates. Afterwards, the curves grow until 8 workers, then comes down. In the G3 dataset, the scalability becomes very high. The response time decreases depending on the number of workers. This improvement is about 30% and 40% when related to the number of updates. This behavior can be attributed to the number of affected vertices and clusters. When this number becomes small, the needed resources (e.g.,workers) should be small. Nevertheless, using many workers, DISCAN uses a lot of communication which increases the processing time.

Impact of the batch size. The main goal of this experiment is to evaluate the impact of batch size on the response time. Initially, we run DISCAN with different datasets (G2, G3, G4 and G5) and different batch sizes (between 2000 en 1000) on a cluster of 10 machines and using the default parameters ε=0.5,

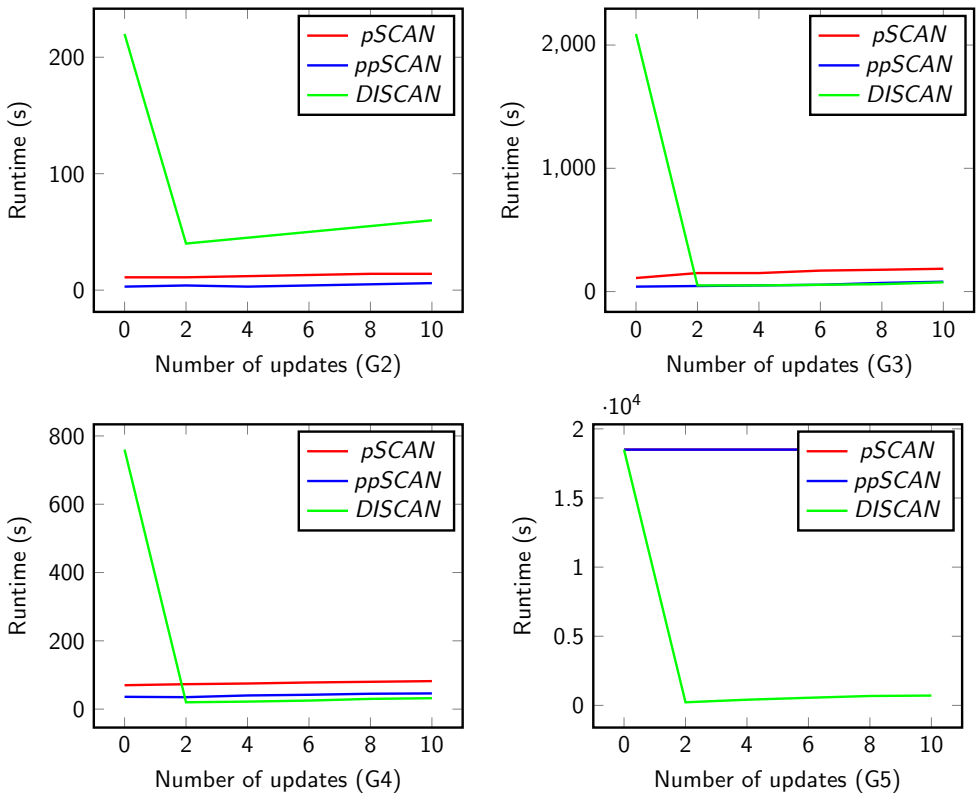


Figure 5.3: Impact of the number of updates on the processing time of the pSCAN, ppSCAN and DISCAN (ε=0.5, μ=3)

μ=3. Fig. 5.5 shows that the running time increases for all curves. This is explained by the initial graph size, as we noticed in Fig. 5.3, but also in function of the batch size. We can also notice that the increase depends on the size of the initial graph and the batch size. The increase rate is about 5%, 50% and 150%, respectively for G2, G4 and G5. This can mainly be explained by the graph size, since for each update DISCAN in the graph maintenance step check all affected vertices and edges in order to update the affected similarities. Therefore, this checking depends on the graph size. On the other hand, in the reclustering step, DISCAN first performs the clustering on the affected vertices. Then, it merges all the affected clusters in the master machine. Thus, the running time of the merging step depends on the number of affected edges because we

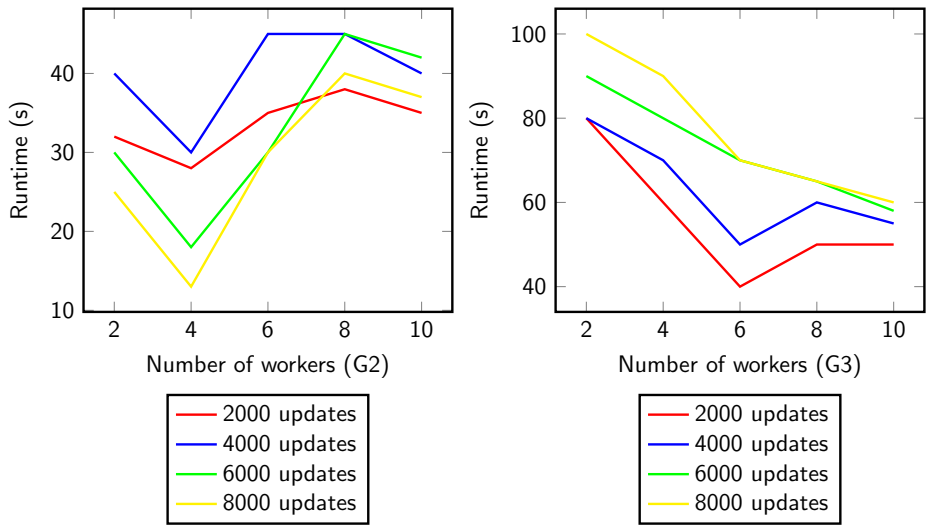


Figure 5.4: Impact of the n workers' number on the processing time of DISCAN (ε=0.5, μ=3)

must check all affected clusters. The affected clusters' checking requires using the communications between the master and other workers. In this way, when the batch is small, i.e. few affected vertices, a little communication should be used.

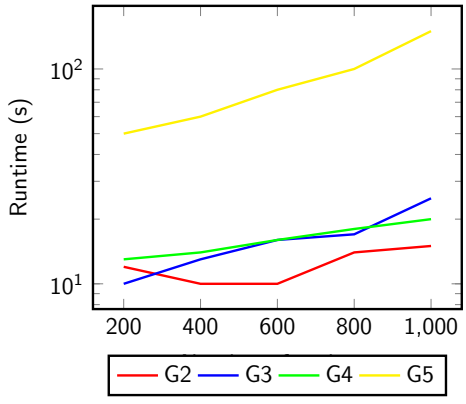


Figure 5.5: Impact of the batch size on the running time (ε=0.5, μ=3)

Impact of the vertex type (internal vs external). Each update on an existing graph maybe in internal or frontier (external) vertices. With, this exper-

iment we show the difference in terms of the running time between the updates in internal vertices and external vertices. We generated for G2, G3, G4 and G5 several internal and external batches of adding edges. After that, we ran DISCAN using them in order to compare their response times.

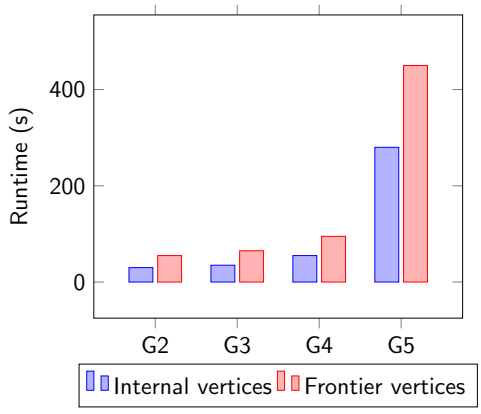


Figure 5.6: Impact of the vertices' type (internal or external) on the processing time (ε=0.5, μ=3)

As depicted in the Fig. 5.6 the updates on the frontier vertices are very expensive with regard to running time. This latters' gap trends to around 50% between internal and external vertices updates. This can be explained by the graph' maintenance step performed by DISCAN, that consumes a lot of processing time in the case of external vertices, compared to internal vertices. In fact, in the internal vertices all maintenance operations are done locally (in the same partition), whereas the graph maintenance is done in several partitions, which requires additional costs. These latter's are mainly related to the duplication of the frontier partitions and obviously the communication between all the affected workers.

DISCAN Steps. We notice, from the previous experiment, that the response time depends on the type of vertices that can be internal or external. Especially, for the external vertices the response time is slow because graph ' maintenance is expensive, compared to internal vertices.

Fig. 5.7 clearly shows a span of the difference between graph maintenance and

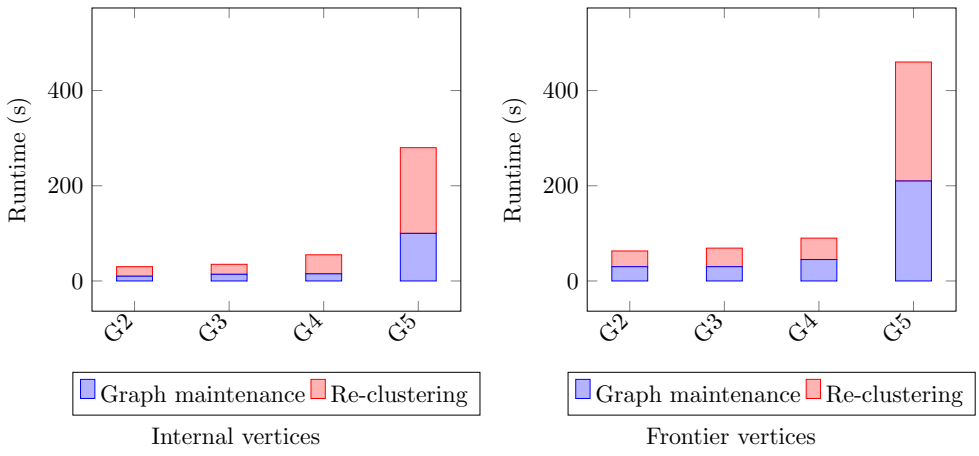


Figure 5.7: Performance of DISCAN steps (ε=0.5, μ=3)

re-clustering steps for both the internal and external vertices. The re-clustering step takes about 30% of the global running time in case of internal vertices, while this rate is about 50% in case of frontiers updates. On the other hand, the difference between internal and external updates in terms of running time during the re-clustering is about 3X. This is understandable because the external updates need additional computing time like the duplication of frontier vertices and the graph maintenance according to the neighboring partitions. The additional treatment sometimes requires several communications with other workers.

5.5 Conclusion

In this chapter, the presence of the third contribution accentuates that an incremental and distributed structural graph clustering algorithm is proposed. In fact, it represents an extension of DSCAN which is named DISCAN. The main goal of DISCAN is to perform a dynamic graph clustering incrementally. In this respect , we presented a full description of the proposed algorithm from graph maintenance to the incremental clustering. In addition, it is worth noting that an experimental study is performed at the end of this chapter which is mainly based on experimental results. Eventually, We can mention that DISCAN outperforms the rest of

algorithms within this work ,and this efficientis is basically related to the running time and scalability.

Key points

- We presented some definitions of dynamic graphs.
- We presented a distributed graph maintenance with dynamic graphs.
- We proposed a distributed and incremental graph clustering algorithm for dynamic and large-scale networks.
- An experimental study to evaluate the proposed incremental graph clustering algorithm for dynamic graphs.

Publications

- W. Inoubli, S. Aridhi, H. Mezni, M. Maddouri, E. Mephu Nguifo. A Distributed and Incremental Algorithm for Large-Scale Graph Clustering. **submitted to Data Mining and Knowledge Discovery**.

Chapter 6

Conclusion and prospects

Contents

6.1 Summary of contributions 130

6.1.1 Comparative study on big data frameworks 130

6.1.2 Distributed structural graph clustering 130

6.1.3 Dynamic graph clustering 131

6.2 Future works and prospects 131

6.2.1 First axis: improvement of the partitioning method proposed by DSCAN 131

6.2.2 Second axis: improvement of the communication 132

6.2.3 Third axis: evolving graphs 132

Goals

In this chapter, we conclude the thesis by summarizing our contributions. Then, we highlight the ongoing works we are conducting.

6.1 Summary of contributions

This thesis deals with distributed and dynamic graph clustering from huge graphs. Firstly, it surveys the big data concepts and popular frameworks proposed in the literature. Secondly, it emphasizes the big graph clustering task with a proposal for a distributed graph clustering algorithm. Thirdly, to solve the problem of dynamic graph processing, it presents an incremental graph clustering algorithm for dynamic graphs. In this section, we first present some details about the comparative study on popular big data frameworks. Then, we summarize the basic keys of the proposed graph clustering algorithm, and finally, we present an extension of the proposed algorithm to deal with dynamic graphs.

6.1.1 Comparative study on big data frameworks

In this axis, we surveyed popular frameworks for large-scale data processing. After a brief description of the main paradigms related to Big Data problems, we presented an overview of popular Big Data frameworks including Hadoop, Spark, Storm and Flink. We presented a categorization of these frameworks according to some main features such as the used programming model, the type of data sources, the supported programming languages and whether the framework allows iterative processing or not. We also conducted an extensive comparative study of the above presented frameworks on a cluster of machines and we highlighted best practices while using the studied Big Data frameworks.

6.1.2 Distributed structural graph clustering

In this axis, we proposed a new distributed graph clustering for big graphs, termed DSCAN. This latter is a distributed algorithm and supports both centralized and distributed graphs. For the centralized graphs, DSCAN provides its own partitioning method. For the distributed graphs, DSCAN takes the already partionned graph as input. The clustering method used by DSCAN is based on structural similarity and gives some other knowledge like bridges and outliers. The experimental results performed in this contribution demonstrates the horizontal scalability in the

case of big graphs. The efficiency of DSCAN is also shown in terms of running time in the case of big graphs. The outcomes of our comparative experiments confirm the efficiency of DSCAN to big graphs.

6.1.3 Dynamic graph clustering

In this contribution, we have extended DSCAN for big and dynamic graph clustering. The new extension is named Distributed and Incremental Structural Clustering Algorithm for networks DISCAN. This latter, unlike other dynamic graphs systems, provides dynamic graph maintenance. Also, DISCAN has covered all possible graph update operations, e.g., adding and deleting of vertices and/or edges. The efficiency related to running time of DISCAN has been demonstrated across experimental results. These results are performed in dynamic graphs in a distributed setting.

6.2 Future works and prospects

This thesis has covered two mains axes. In the first axis, we provided a large-scale graph clustering in a distributed setting. In the second axis, we touched the clustering of large and dynamic graphs. For each axis, an algorithm has been proposed to overcome the cited issues. The proposed algorithms have shown significant improvement compared to the existing algorithms. However, our algorithms have some challenges related to graph partitioning and data communication. Therefore, these challenges represent one of our future works. In this section, we propose some details about these ongoing work.

6.2.1 First axis: improvement of the partitioning method proposed by DSCAN

The proposed distributed graph clustering algorithm consists in dividing an input graph and performing a distributed graph clustering. However, the experimental results have shown that a significant impact of the partitioning scheme on the

running time. This is explained by the number of frontier vertices. When the number is high, a cost of computation resources is added. Starting from this interpretation, the partitioning step improvement can automatically decrease the running time of the proposed algorithm. As a future work in this axis, we will study the existing distributed graph partitioning methods and compare them with our proposed algorithm. We also aim to propose a dedicated partitioning method to DISCAN.

6.2.2 Second axis: improvement of the communication

The communication is a primordial aspect in distributed computing. Both DSCAN and DISCAN use the bandwidth exhaustively which increase the running time in the case of several dense graphs. This issue will represent a future work in the short term. To overcome this issue, we will propose a shared memory instead of a message passing communication technique between workers in the cluster. The communication method proposed can decrease the bandwidth resource consumption and then decrease the running time.

6.2.3 Third axis: evolving graphs

This thesis deals with both static and dynamic graph clustering, but recently in some real-word applications, another kind of graphs called evolving graph has emerged. In fact, evolving graph consists on a series of snapshots of an initial graph, which evolves over time. In this axis, we aim to extend and adapt our proposed algorithm to deal with evolving graph clustering. We will study the proposed approaches that take into consideration evolving graphs during the graph clustering. Then, and according to the study carried out, we will try to adapt DISCAN in order to track the graph evolution and to maintain the graph clustering over time.

Publications

Journal papers

- Inoubli. W, Aridhi. S, Mezni. H, Maddouri. M et Mephu Nguifo. E An experimental survey on big data frameworks Journal: Future Generation Computer Systems Elsevier, 86, pp. 546-564, 2018

Conference papers

- Inoubli. W, Aridhi. S, Mezni. H, Maddouri. M et Mephu Nguifo. E Un algorithme distribué pour le clustering de grands graphesExtraction et Gestion des Connaissances (EGC) 2020, brussels, Belgium.vol. RNTI-E-36, pp.349-356
- Inoubli. W, Aridhi. S, Mezni. H, Maddouri. M et Mephu Nguifo. E An experimental survey on big data frameworksBDA 2018 - 34ème Conférence sur la Gestion de Données – Principes, Technologies et Applications, Oct 2018, Bucarest, Romania (Highlight paper).
- Inoubli. W, Aridhi. S, Mezni. H, Maddouri. M et Mephu Nguifo. E A Comparative Study on Streaming Frameworks for Big Datathe Latin America Data Science Workshop co-located with 44th International Conference on Very Large Data Bases, Rio de Janeiro, Brazil, Aug 27, 2018
- Inoubli. W, Aridhi. S, Mezni. H, Maddouri. M et Mephu Nguifo. E An Experimental Survey on Big Data FrameworksExtremely Large Databases Conference (XLDB) 2017, Clermont Ferrand, France. (Lightning talk, poster)
- Inoubli. W, Almada. L, Coelho da Silva. T.L, Coutinho. G, Peres. L, Magalhaes. R.P, de Macedo. J.F, Aridhi. S, Mephu Nguifo. E A Distributed Framework for Large-Scale Time-Dependent Graph AnalysisLarge-Scale Time Dependent Graphs (TD-LSG) in conjunction with ECMLPKDD, Skopje, Macedonia, 2017

Submitted papers

- W. Inoubli, S. Aridhi, H. Mezni, M. Maddouri, E. Mephu Nguifo. A Novel Scalable Clustering Method for Distributed Networks. submitted to Information Systems.
- W. Inoubli, S. Aridhi, H. Mezni, M. Maddouri, E. Mephu Nguifo. A Distributed and Incremental Algorithm for Large-Scale Graph Clustering. Submitted to Data Mining and Knowledge Discovery.

Bibliography

[Abbas 2018] Zainab Abbas, Vasiliki Kalavri, Paris Carbone and Vladimir Vlassov. *Streaming graph partitioning: an experimental study*. Proceedings of the VLDB Endowment, vol. 11, no. 11, pages 1590–1603, 2018.

[Aktunc 2015] Riza Aktunc, Ismail Hakki Toroslu, Mert Ozer and Hasan Davulcu. *A dynamic modularity based community detection algorithm for large-scale networks: DSLM*. In Proceedings of the 2015 IEEE/ACM international conference on advances in social networks analysis and mining 2015, pages 1177–1183, 2015.

[Alexandrov 2014a] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl*et al. The Stratosphere platform for big data analytics*. The VLDB JournalThe International Journal on Very Large Data Bases, vol. 23, no. 6, pages 939–964, 2014.

[Alexandrov 2014b] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas and Daniel Warneke. *The Stratosphere Platform for Big Data Analytics*. The VLDB Journal, vol. 23, no. 6, pages 939–964, 2014.

[Alvarez-Hamelin 2008] José Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat and Alessandro Vespignani. *K-core decomposition of Internet graphs: hierarchies, self-similarity and measurement biases*. NHM, vol. 3, no. 2, pages 371–393, 2008.

[Amjad 2018] Tehmina Amjad, Ali Daud, Sadia Khan, Rabeeh Ayaz Abbasi and Faisal Imran. *Prediction of Rising Stars from Pakistani Research Communities*. In 2018 14th International Conference on Emerging Technologies (ICET), pages 1–6. IEEE, 2018.

[Andlinger 2015] PAUL Andlinger. *Graph DBMS increased their popularity by 500% within the last 2 years*, 2015.

[Aridhi 2015] Sabeur Aridhi, Laurent d'Orazio, Mondher Maddouri and Engelbert Mephu Nguifo. *Density-based data partitioning strategy to approximate large-scale subgraph mining*. Information Systems, vol. 48, pages 213–223, 2015.

[Aridhi 2016] Sabeur Aridhi and Engelbert Mephu Nguifo. *Big Graph Mining: Frameworks and Techniques*. Big Data Research, vol. 6, pages 1 – 10, 2016.

[Aridhi 2017] Sabeur Aridhi, Alberto Montresor and Yannis Velegrakis. *BLADYG: A Graph Processing Framework for Large Dynamic Graphs*. Big Data Research, vol. 9, no. C, pages 9–17, 2017.

[Armbrust 2015] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi and Matei Zaharia. *Spark SQL: Relational Data Processing in Spark*. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[Armstrong 2013] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur and Mark Callaghan. *LinkBench: a database benchmark based on the Facebook social graph*. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pages 1185–1196, 2013.

[Assefi 2017] M. Assefi, E. Behravesh, G. Liu and A. P. Tafti. *Big data machine learning using apache spark MLlib*. In 2017 IEEE International Conference on Big Data (Big Data), pages 3492–3498, Dec 2017.

[Aynaud 2010] Thomas Aynaud and Jean-Loup Guillaume. *Static community detection algorithms for evolving networks*. In 8th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks, pages 513–519. IEEE, 2010.

[Baborska-Narozny 2016] Magdalena Baborska-Narozny, Eve Stirling and Fionn Stevenson. *Exploring the Relationship Between a'Facebook Group'and Face-to-Face Interactions in'Weak-Tie'Residential Communities*. In Proceedings of the 7th 2016 International Conference on Social Media & Society, page 17. ACM, 2016.

[Bajaber 2016] Fuad Bajaber, Radwa Elshawi, Omar Batarfi, Abdulrahman Altalhi, Ahmed Barnawi and Sherif Sakr. *Big Data 2.0 Processing Systems: Taxonomy and Open Challenges*. Journal of Grid Computing, vol. 14, no. 3, pages 379–405, 2016.

[Bello-Orgaz 2016a] Gema Bello-Orgaz, Jason J. Jung and David Camacho. *Social big data: Recent achievements and new challenges*. Information Fusion, vol. 28, pages 45 – 59, 2016.

[Bello-Orgaz 2016b] Gema Bello-Orgaz, Jason J Jung and David Camacho. *Social big data: Recent achievements and new challenges*. Information Fusion, vol. 28, pages 45–59, 2016.

[Bonchi 2011] Francesco Bonchi, Carlos Castillo, Aristides Gionis and Alejandro Jaimes. *Social network analysis and mining for business applications*. ACM Transactions on Intelligent Systems and Technology (TIST), vol. 2, no. 3, pages 1–37, 2011.

[Brandes 2003] Ulrik Brandes, Marco Gaertler and Dorothea Wagner. *Experiments on graph clustering algorithms*. In European Symposium on Algorithms, pages 568–579. Springer, 2003.

[Brandes 2007] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Gorke, Martin Hoefer, Zoran Nikoloski and Dorothea Wagner. *On modularity clustering*. IEEE transactions on knowledge and data engineering, vol. 20, no. 2, pages 172–188, 2007.

[Bu 2012] Yingyi Bu, Bill Howe, Magdalena Balazinska and Michael D. Ernst. *The HaLoop Approach to Large-scale Iterative Data Analysis*. The VLDB Journal, vol. 21, no. 2, pages 169–190, April 2012.

[Bull 1999] J Mark Bull. *Measuring synchronisation and scheduling overheads in OpenMP*. In Proceedings of First European Workshop on OpenMP, volume 8, page 49. Citeseer, 1999.

[Cao 2009] Lili Cao and John Krumm. *From GPS traces to a routable road map*. In Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems, pages 3–12. ACM, 2009.

[Carbone 2015] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi and Kostas Tzoumas. *Apache Flink^{TM}: Stream and Batch Processing in a Single Engine*. IEEE Data Eng. Bull., vol. 38, no. 4, pages 28–38, 2015.

[Chakrabarti 2008] Soumen Chakrabarti, Earl Cox, Eibe Frank, Ralf Hartmut Gting, Jiawei Han, Xia Jiang, Micheline Kamber, Sam S. Lightstone, Thomas P. Nadeau, Richard E Neapolitan, Dorian Pyle, Mamdouh Refaat, Markus Schneider, Toby J. Teorey and Ian H. Witten. Data mining: Know it all. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[Chambers 2010] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw and Nathan Weizenbaum. *FlumeJava: Easy, Efficient Data-parallel Pipelines*. In Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10, pages 363–375, New York, NY, USA, 2010. ACM.

[Chang 2017a] Lijun Chang, Wei Li, Lu Qin, Wenjie Zhang and Shiyu Yang. *pSCAN: Fast and Exact Structural Graph Clustering*. IEEE Transactions on Knowledge and Data Engineering, vol. 29, no. 2, pages 387–401, 2017.

[Chang 2017b] Lijun Chang, Wei Li, Lu Qin, Wenjie Zhang and Shiyu Yang. *pSCAN: Fast and Exact Structural Graph Clustering*. IEEE Transactions on Knowledge and Data Engineering, vol. 29, no. 2, pages 387–401, 2017.

[Che 2018] Yulin Che, Shixuan Sun and Qiong Luo. *Parallelizing Pruning-based Graph Structural Clustering*. In Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, Eugene, OR, USA, August 13-16, 2018, pages 77:1–77:10, 2018.

[Chen 2014a] CL Philip Chen and Chun-Yang Zhang. *Data-intensive applications, challenges, techniques and technologies: A survey on Big Data*. Information Sciences, vol. 275, pages 314–347, 2014.

[Chen 2014b] Min Chen, Shiwen Mao and Yunhao Liu. *Big data: A survey*. Mobile Networks and Applications, vol. 19, no. 2, pages 171–209, 2014.

[Chen 2014c] Min Chen, Shiwen Mao and Yunhao Liu. *Big data: A survey*. Mobile Networks and Applications, vol. 19, no. 2, pages 171–209, 2014.

[Chen 2014d] Min Chen, Shiwen Mao, Yin Zhang and Victor CM Leung. Big data: related technologies, challenges and future prospects. Springer, 2014.

[Cuzzocrea 2014] Alfredo Cuzzocrea and Il-Yeol Song. *Big graph analytics: The state of the art and future research agenda*. In Proceedings of the 17th International Workshop on Data Warehousing and OLAP, pages 99–101, 2014.

[Dean 2008] Jeffrey Dean and Sanjay Ghemawat. *MapReduce: simplified data processing on large clusters*. Commun. ACM, vol. 51, no. 1, pages 107–113, 2008.

[Dede 2014] Elif Dede, Zacharia Fadika, Madhusudhan Govindaraju and Lavanya Ramakrishnan. *Benchmarking MapReduce implementations under different application scenarios*. Future Generation Computer Systems, vol. 36, pages 389–399, 2014.

[Dhifli 2017a] Wajdi Dhifli, Sabeur Aridhi and Engelbert Mephu Nguifo. *MR-SimLab: Scalable subgraph selection with label similarity for big data*. Information Systems, vol. 69, pages 155 – 163, 2017.

[Dhifli 2017b] Wajdi Dhifli, Sabeur Aridhi and Engelbert Mephu Nguifo. *MR-SimLab: Scalable subgraph selection with label similarity for big data*. Information Systems, vol. 69, pages 155–163, 2017.

[Dhillon 2001] Inderjit S Dhillon. *Co-clustering documents and words using bipartite spectral graph partitioning*. In Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining, pages 269–274. ACM, 2001.

[Ding 2001] Chris HQ Ding, Xiaofeng He, Hongyuan Zha, Ming Gu and Horst D Simon. *A min-max cut algorithm for graph partitioning and data clustering*. In Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on, pages 107–114. IEEE, 2001.

[Ding 2013a] Linlin Ding, Guoren Wang, Junchang Xin, Xiaoyang Wang, Shan Huang and Rui Zhang. *ComMapReduce: an improvement of mapreduce with lightweight communication mechanisms*. Data & Knowledge Engineering, vol. 88, pages 224–247, 2013.

[Ding 2013b] Linlin Ding, Guoren Wang, Junchang Xin, Xiaoyang Wang, Shan Huang and Rui Zhang. *ComMapReduce: An improvement of MapReduce with lightweight communication mechanisms*. Data & Knowledge Engineering, vol. 88, pages 224–247, 2013.

[Doerr 2007] Benjamin Doerr and Daniel Johannsen. *Adjacency list matchings: an ideal genotype for cycle covers*. In Proceedings of the 9th annual conference on Genetic and evolutionary computation, pages 1203–1210. ACM, 2007.

[Domann 2016] Jaschar Domann, Jens Meiners, Lea Helmers and Andreas Lommatzsch. *Real-time News Recommendations using Apache Spark*. In Working Notes of CLEF 2016 - Conference and Labs of the Evaluation forum, Évora, Portugal, 5-8 September, 2016., pages 628–641, 2016.

[Doulkeridis 2014] Christos Doulkeridis and Kjetil Nørvåg. *A survey of large-scale analytical query processing in MapReduce*. The VLDB Journal—The

International Journal on Very Large Data Bases, vol. 23, no. 3, pages 355–380, 2014.

[DâTMAzevedo 2005] Eduardo F DâTMAzevedo, Mark R Fahey and Richard T Mills. *Vectorized sparse matrix multiply for compressed row storage format*. In International Conference on Computational Science, pages 99–106. Springer, 2005.

[Eadline 2015] Douglas Eadline. Hadoop 2 quick-start guide: Learn the essentials of big data computing in the apache hadoop 2 ecosystem. Addison-Wesley Professional, 1st édition, 2015.

[Ekanayake 2010] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu and Geoffrey Fox. *Twister: A Runtime for Iterative MapReduce*. In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[Elsayed 2014] Abdelrahman Elsayed, Osama Ismail and Mohamed E El-Sharkawi. *MapReduce: state-of-the-art and research directions"*. International Journal of Computer and Electrical Engineering, vol. 6, no. 1, pages 34–39, 2014.

[Elser 2013] B. Elser and A. Montresor. *An evaluation study of BigData frameworks for graph processing*. In IEEE International Conference on Big Data, pages 60–67, 2013.

[Fadika 2010] Zacharia Fadika and Madhusudhan Govindaraju. *Lemo-mr: Low overhead and elastic mapreduce implementation optimized for memory and cpu-intensive applications*. In Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on, pages 1–8. IEEE, 2010.

[Faloutsos 1999] Michalis Faloutsos, Petros Faloutsos and Christos Faloutsos. *On power-law relationships of the internet topology*. ACM SIGCOMM computer communication review, vol. 29, no. 4, pages 251–262, 1999.

[Faloutsos 2004] Christos Faloutsos, Kevin S Mccurley and Andrew Tomkins. *Connection subgraphs in social networks*. In SIAM International Conference on Data Mining, Workshop on Link Analysis, Counterterrorism and Security, 2004.

[Fan 2017] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu and Jiaxin Jiang. *GRAPE: Parallelizing sequential graph computations*. Proceedings of the VLDB Endowment, vol. 10, no. 12, pages 1889–1892, 2017.

[Fernández 2014] Alberto Fernández, Sara del Río, Victoria López, Abdullah Bawakid, María J del Jesus, José M Benítez and Francisco Herrera. *Big Data with Cloud Computing: an insight on the computing environment, MapReduce, and programming frameworks*. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, vol. 4, no. 5, pages 380–409, 2014.

[Fournier-Viger] Philippe Fournier-Viger, Ganghuan He, Chao Cheng, Jiaxuan Li, Min Zhou, Jerry Chun-Wei Lin and Unil Yun. *A survey of pattern mining in dynamic graphs*. WIREs Data Mining and Knowledge Discovery, vol. n/a, no. n/a, page e1372.

[Gandomi 2015a] Amir Gandomi and Murtaza Haider. *Beyond the hype: Big data concepts, methods, and analytics*. International journal of information management, vol. 35, no. 2, pages 137–144, 2015.

[Gandomi 2015b] Amir Gandomi and Murtaza Haider. *Beyond the hype: Big data concepts, methods, and analytics*. International Journal of Information Management, vol. 35, no. 2, pages 137 – 144, 2015.

[García-Gil 2017] Diego García-Gil, Sergio Ramírez-Gallego, Salvador García and Francisco Herrera. *A comparison on scalability for batch big data processing on Apache Spark and Apache Flink*. Big Data Analytics, vol. 2, no. 1, page 1, 2017.

[Garg 2013] Nishant Garg. Apache kafka. Packt Publishing, 2013.

[Ghemawat 2003] Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung. *The Google File System*. SIGOPS Oper. Syst. Rev., vol. 37, no. 5, pages 29–43, October 2003.

[Giatsidis 2011] Christos Giatsidis, Dimitrios M. Thilikos and Michalis Vazirgiannis. *Evaluating Cooperation in Communities with the k-Core Structure*. In Proceedings of the 2011 International Conference on Advances in Social Networks Analysis and Mining, ASONAM '11, pages 87–93, Washington, DC, USA, 2011. IEEE Computer Society.

[Gonzalez 2012] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson and Carlos Guestrin. *Powergraph: Distributed graph-parallel computation on natural graphs*. In Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12), pages 17–30, 2012.

[Gonzalez 2014] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin and Ion Stoica. *Graphx: Graph processing in a distributed dataflow framework*. In 11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14), pages 599–613, 2014.

[Goudreau 1996] Mark Goudreau, Kevin Lang, Satish Rao, Torsten Suel and Thanasis Tsantilas. *Towards efficiency and portability: Programming with the BSP model*. In Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures, pages 1–12, 1996.

[Goyal 2018] Palash Goyal and Emilio Ferrara. *Graph embedding techniques, applications, and performance: A survey*. Knowledge-Based Systems, vol. 151, pages 78–94, 2018.

[Gu 2009a] Yunhong Gu and Robert L Grossman. *Sector and Sphere: the design and implementation of a high-performance data cloud*. Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, vol. 367, no. 1897, pages 2429–2445, 2009.

[Gu 2009b] Yunhong Gu and Robert L Grossman. *Sector and Sphere: the design and implementation of a high-performance data cloud*. Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, vol. 367, no. 1897, pages 2429–2445, 2009.

[Gui 2019] Chuang-Yi Gui, Long Zheng, Bingsheng He, Cheng Liu, Xin-Yu Chen, Xiao-Fei Liao and Hai Jin. *A survey on graph processing accelerators: Challenges and opportunities*. Journal of Computer Science and Technology, vol. 34, no. 2, pages 339–371, 2019.

[Günnemann 2012] Stephan Günnemann, Brigitte Boden and Thomas Seidl. *Finding density-based subspace clusters in graphs with feature vectors*. Data mining and knowledge discovery, vol. 25, no. 2, pages 243–269, 2012.

[Gupta 2015] Y. Gupta. Kibana essentials. Packt Publishing, 2015.

[Hadian 2016] Ali Hadian, Sadegh Nobari, Behrooz Minaei-Bidgoli and Qiang Qu. *ROLL: Fast In-Memory Generation of Gigantic Scale-free Networks*. In Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, pages 1829–1842, New York, NY, USA, 2016. ACM.

[Han 2015] Minyang Han and Khuzaima Daudjee. *Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems*. Proceedings of the VLDB Endowment, vol. 8, no. 9, pages 950–961, 2015.

[Hartigan 1979] John A Hartigan and Manchek A Wong. *Algorithm AS 136: A k-means clustering algorithm*. Journal of the Royal Statistical Society. Series C (Applied Statistics), vol. 28, no. 1, pages 100–108, 1979.

[He 2008] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju and Tuyong Wang. *Mars: a MapReduce framework on graphics processors*. In Proceedings of the 17th international conference on Parallel architectures and compilation techniques, pages 260–269. ACM, 2008.

[He 2015] Wenting He, Huimin Cui, Binbin Lu, Jiacheng Zhao, Shengmei Li, Gong Ruan, Jingling Xue, Xiaobing Feng, Wensen Yang and Youliang Yan.

Hadoop+: Modeling and Evaluating the Heterogeneity for MapReduce Applications in Heterogeneous Clusters. In Proceedings of the 29th ACM on International Conference on Supercomputing, ICS '15, pages 143–153, New York, NY, USA, 2015. ACM.

[Hindman 2011] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker and Ion Stoica. *Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.* In NSDI, volume 11, pages 22–22, 2011.

[Hunt 2010] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira and Benjamin Reed. *ZooKeeper: Wait-free Coordination for Internet-scale Systems*. In Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[Huttenhower 2009] Curtis Huttenhower, Sajid O. Mehmood and Olga G. Troyanskaya. *Graphle: Interactive exploration of large, dense graphs*. BMC Bioinformatics, vol. 10, page 417, 2009.

[Imam 2012] Shams M Imam and Vivek Sarkar. *Integrating task parallelism with actors*. ACM SIGPLAN Notices, vol. 47, no. 10, pages 753–772, 2012.

[Inoubli 2018a] Wissem Inoubli, Sabeur Aridhi, Haithem Mezni, Mondher Maddouri and Engelbert Nguifo. *A comparative study on streaming frameworks for big data*. 2018.

[Inoubli 2018b] Wissem Inoubli, Sabeur Aridhi, Haithem Mezni, Mondher Maddouri and Engelbert Mephu Nguifo. *An experimental survey on big data frameworks*. Future Generation Computer Systems, vol. 86, pages 546–564, 2018.

[Isard 2007] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell and Dennis Fetterly. *Dryad: distributed data-parallel programs from sequential building blocks*. In ACM SIGOPS Operating Systems Review, volume 41, pages 59–72. ACM, 2007.

[Islam 2016] Nusrat Sharmin Islam, Md Wasi-ur Rahman, Xiaoyi Lu and Dhabaleswar K DK Panda. *Efficient data access strategies for Hadoop and Spark on HPC cluster with heterogeneous storage*. In Big Data (Big Data), 2016 IEEE International Conference on, pages 223–232. IEEE, 2016.

[Iyer 2018] Anand Padmanabha Iyer, Aurojit Panda, Shivaram Venkataraman, Mosharaf Chowdhury, Aditya Akella, Scott Shenker and Ion Stoica. *Bridging the GAP: towards approximate graph analytics*. In Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), page 10. ACM, 2018.

[Jha 2014] Shantenu Jha, Judy Qiu, Andre Luckow, Pradeep Mantha and Geoffrey C Fox. *A tale of two data-intensive paradigms: Applications, abstractions, and architectures*. In Big Data (BigData Congress), 2014 IEEE International Congress on, pages 645–652. IEEE, 2014.

[Kalavri 2017] Vasiliki Kalavri, Vladimir Vlassov and Seif Haridi. *High-level programming abstractions for distributed graph processing*. IEEE Transactions on Knowledge and Data Engineering, vol. 30, no. 2, pages 305–324, 2017.

[Kang 2009] U Kang, Charalampos E Tsourakakis and Christos Faloutsos. *Pegasus: A peta-scale graph mining system implementation and observations*. In 2009 Ninth IEEE international conference on data mining, pages 229–238. IEEE, 2009.

[Khayyat 2013] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams and Panos Kalnis. *Mizan: a system for dynamic load balancing in large-scale graph processing*. In Proceedings of the 8th ACM European Conference on Computer Systems, pages 169–182, 2013.

[Kozawa 2017] Yusuke Kozawa, Toshiyuki Amagasa and Hiroyuki Kitagawa. *GPU-Accelerated Graph Clustering via Parallel Label Propagation*. In Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, pages 567–576. ACM, 2017.

[Kwon 2012] YongChul Kwon, Magdalena Balazinska, Bill Howe and Jerome Rolia. *SkewTune: Mitigating Skew in Mapreduce Applications*. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12, pages 25–36, New York, NY, USA, 2012. ACM.

[Landset 2015a] Sara Landset, Taghi M. Khoshgoftaar, Aaron N. Richter and Tawfiq Hasanin. *A survey of open source tools for machine learning with big data in the Hadoop ecosystem*. Journal of Big Data, vol. 2, no. 1, pages 1–36, 2015.

[Landset 2015b] Sara Landset, Taghi M Khoshgoftaar, Aaron N Richter and Tawfiq Hasanin. *A survey of open source tools for machine learning with big data in the Hadoop ecosystem*. Journal of Big Data, vol. 2, no. 1, page 1, 2015.

[LaSalle 2015] Dominique LaSalle and George Karypis. *Multi-threaded modularity based graph clustering using the multilevel paradigm*. Journal of Parallel and Distributed Computing, vol. 76, pages 66–80, 2015.

[Lei 2016] Xiujuan Lei, Fei Wang, Fang-Xiang Wu, Aidong Zhang and Witold Pedrycz. *Protein complex identification through Markov clustering with firefly algorithm on dynamic protein–protein interaction networks*. Information Sciences, vol. 329, pages 303–316, 2016.

[Leskovec 2015] Jure Leskovec and Andrej Krevl. {*SNAP Datasets*}:{*Stanford*} *Large Network Dataset Collection*. 2015.

[Li 2014] Feng Li, Beng Chin Ooi, M Tamer Özsu and Sai Wu. *Distributed data management using MapReduce*. ACM Computing Surveys (CSUR), vol. 46, no. 3, page 31, 2014.

[Li 2015] Ren Li, Haibo Hu, Heng Li, Yunsong Wu and Jianxi Yang. *MapReduce Parallel Programming Model: A State-of-the-Art Survey*. International Journal of Parallel Programming, pages 1–35, 2015.

[Li 2016] Xiaoping Li, Tianze Jiang and Rubén Ruiz. *Heuristics for periodical batch job scheduling in a MapReduce computing framework*. Information Sciences, vol. 326, pages 119–133, 2016.

[Lim 2014] Sungsu Lim, Seungwoo Ryu, Sejeong Kwon, Kyomin Jung and Jae-Gil Lee. *LinkSCAN*: Overlapping community detection using the link-space transformation*. In 2014 IEEE 30th International Conference on Data Engineering (ICDE), pages 292–303. IEEE, 2014.

[Lin 2010] Jimmy Lin and Chris Dyer. Data-intensive text processing with mapreduce. Morgan and Claypool Publishers, 2010.

[Liu 2009] Yang Liu, Xiaohong Jiang, Huajun Chen, Jun Ma and Xiangyu Zhang. *Mapreduce-based pattern finding algorithm applied in motif detection for prescription compatibility network*. In International Workshop on Advanced Parallel Processing Technologies, pages 341–355. Springer, 2009.

[Liu 2014] Xiufeng Liu, Nadeem Iftikhar and Xike Xie. *Survey of real-time processing systems for big data*. In Proceedings of the 18th International Database Engineering & Applications Symposium, pages 356–361. ACM, 2014.

[Lourenço 2015] João Ricardo Lourenço, Bruno Cabral, Paulo Carreiro, Marco Vieira and Jorge Bernardino. *Choosing the right NoSQL database for the job: a quality attribute evaluation*. Journal of Big Data, vol. 2, no. 1, pages 1–26, 2015.

[Low 2012] Yucheng Low, Danny Bickson, Joseph Gonzalez and et al. *Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud*. Proc. VLDB Endow., vol. 5, no. 8, pages 716–727, April 2012.

[Low 2014] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin and Joseph Hellerstein. *Graphlab: A new framework for parallel machine learning*. arXiv preprint arXiv:1408.2041, 2014.

[Lu 2014] Xiaoyi Lu, Fan Liang, Bing Wang, Li Zha and Zhiwei Xu. *DataMPI: extending MPI to hadoop-like big data computing*. In Parallel and Dis-

tributed Processing Symposium, 2014 IEEE 28th International, pages 829–838. IEEE, 2014.

[Mai 2017] Son T Mai, Martin Storgaard Dieu, Ira Assent, Jon Jacobsen, Jesper Kristensen and Mathias Birk. *Scalable and interactive graph clustering algorithm on multicore CPUs*. In Data Engineering (ICDE), 2017 IEEE 33rd International Conference on, pages 349–360. IEEE, 2017.

[Malewicz 2010a] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser and Grzegorz Czajkowski. *Pregel: a system for large-scale graph processing*. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pages 135–146. ACM, 2010.

[Malewicz 2010b] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser and Grzegorz Czajkowski. *Pregel: a system for large-scale graph processing*. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pages 135–146. ACM, 2010.

[Marcu 2016] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu and María S Pérez-Hernández. *Spark versus flink: Understanding performance in big data analytics frameworks*. In Cluster Computing (CLUSTER), 2016 IEEE International Conference on, pages 433–442. IEEE, 2016.

[Oguntimilehin 2014] A Oguntimilehin and EO Ademola. *A Review of Big Data Management, Benefits and Challenges*. Journal of Emerging Trends in Computing and Information Sciences, vol. 5, no. 6, pages 433–438, 2014.

[Piro 2014] G. Piro, I. Cianci, L.A. Grieco, G. Boggia and P. Camarda. *Information centric services in Smart Cities*. Journal of Systems and Software, vol. 88, pages 169 – 188, 2014.

[Polato 2014] Ivanilton Polato, Reginaldo Ré, Alfredo Goldman and Fabio Kon. *A comprehensive view of Hadoop research—A systematic literature review*. Journal of Network and Computer Applications, vol. 46, pages 1–25, 2014.

[Prabhakaran 2012] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou and Maya Haradasan. *Managing large graphs on multi-cores with graph awareness*. In Presented as part of the 2012 USENIX Annual Technical Conference 12), pages 41–52, 2012.

[Resnick 1997] Paul Resnick and Hal R. Varian. *Recommender Systems*. Commun. ACM, vol. 40, no. 3, pages 56–58, March 1997.

[Rossetti 2018] Giulio Rossetti and Rémy Cazabet. *Community discovery in dynamic networks: a survey*. ACM Computing Surveys (CSUR), vol. 51, no. 2, pages 1–37, 2018.

[Said 2018] Anwar Said, Rabeeh Ayaz Abbasi, Onaiza Maqbool, Ali Daud and Naif Radi Aljohani. *CC-GA: A clustering coefficient based genetic algorithm for detecting communities in social networks*. Applied Soft Computing, vol. 63, pages 59–70, 2018.

[Saidi 2009] Rabie Saidi, Mondher Maddouri and Engelbert Mephu Nguifo. *Comparing graph-based representations of protein for mining purposes*. In Proceedings of the KDD-09 Workshop on Statistical and Relational Learning in Bioinformatics, pages 35–38, 2009.

[Sakr 2016] Sherif Sakr. Big data 2.0 processing systems - A survey. Springer Briefs in Computer Science. Springer, 2016.

[Salihoglu 2013] Semih Salihoglu and Jennifer Widom. *GPS: a graph processing system*. In Proceedings of the 25th International Conference on Scientific and Statistical Database Management, page 22. ACM, 2013.

[Samza 2014] Apache Samza. *LinkedIn's Real-time Stream Processing Framework, by Riccomini, C*, 2014.

[Satuluri 2010] Venu Satuluri, Srinivasan Parthasarathy and Duygu Ucar. *Markov clustering of protein interaction networks with improved balance and scalability*. In Proceedings of the first ACM international conference on bioinformatics and computational biology, pages 247–256, 2010.

[Schroeck 2012] Michael Schroeck, Rebecca Shockley, Janet Smart, Dolores Romero-Morales and Peter Tufano. *Analytics: The real-world use of big data: How innovative enterprises extract value from uncertain data*. IBM Institute for Business Value, 2012.

[Sengupta 2016] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L Willke, Jeffrey Young, Matthew Wolf and Karsten Schwan. *Graphin: An online high performance incremental graph processing framework*. In European Conference on Parallel Processing, pages 319–333. Springer, 2016.

[Seo 2017] Jung Hyuk Seo and Myoung Ho Kim. *pm-SCAN: an I/O Efficient Structural Clustering Algorithm for Large-scale Graphs*. In Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, pages 2295–2298. ACM, 2017.

[Shao 2013] Bin Shao, Haixun Wang and Yatao Li. *Trinity: A Distributed Graph Engine on a Memory Cloud*. In Proc. of the Int. Conf. on Management of Data. ACM, 2013.

[Shi 2015] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald and Fatma Özcan. *Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics*. Proc. VLDB Endow., vol. 8, no. 13, pages 2110–2121, September 2015.

[Shiokawa 2015] Hiroaki Shiokawa, Yasuhiro Fujiwara and Makoto Onizuka. *SCAN++: efficient algorithm for finding clusters, hubs and outliers on large-scale graphs*. Proceedings of the VLDB Endowment, vol. 8, no. 11, pages 1178–1189, 2015.

[Siddique 2016] Kamran Siddique, Zahid Akhtar, Edward J Yoon, Young-Sik Jeong, Dipankar Dasgupta and Yangwoo Kim. *Apache Hama: An emerging bulk synchronous parallel computing framework for big data applications*. IEEE Access, vol. 4, pages 8879–8887, 2016.

[Singh 2014] Dilpreet Singh and Chandan K. Reddy. *A survey on platforms for big data analytics*. Journal of Big Data, vol. 2, no. 1, page 8, 2014.

[Skeirik 2013] Stephen Skeirik, Rakesh B Bobba and Jose Meseguer. *Formal analysis of fault-tolerant group key management using ZooKeeper*. In Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on, pages 636–641. IEEE, 2013.

[Sparks 2013] Evan R. Sparks, Ameet Talwalkar, Virginia Smith, Jey Kottalam, Xinghao Pan, Joseph E. Gonzalez, Michael J. Franklin, Michael I. Jordan and Tim Kraska. *MLI: An API for Distributed Machine Learning*. In 2013 IEEE 13th International Conference on Data Mining, Dallas, TX, USA, December 7-10, 2013, pages 1187–1192, 2013.

[Stimmel 2015a] Carol L. Stimmel. Building smart cities: Analytics, ict, and design thinking. Auerbach Publications, Boston, MA, USA, 2015.

[Stimmel 2015b] Carol L. Stimmel. Building smart cities: Analytics, ict, and design thinking. Auerbach Publications, Boston, MA, USA, 2015.

[Stovall 2015] Thomas Ryan Stovall, Sinan Kockara and Recep Avci. *GPUSCAN: GPU-Based Parallel Structural Clustering Algorithm for Networks*. IEEE Transactions on Parallel and Distributed Systems, vol. 26, no. 12, pages 3381–3393, 2015.

[Sun 2019] He Sun and Luca Zanetti. *Distributed graph clustering and sparsification*. ACM Trans. on Parallel Computing (TOPC), vol. 6, no. 3, pages 1–23, 2019.

[Sundaram 2015] Narayanan Sundaram, Nadathur Rajagopalan Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Satya Gautam Vadlamudi, Dipankar Das and Pradeep Dubey. *Graphmat: High performance graph analytics made productive*. arXiv preprint arXiv:1503.07241, 2015.

[Takahashi 2017] Tomokatsu Takahashi, Hiroaki Shiokawa and Hiroyuki Kitagawa. *SCAN-XP: Parallel Structural Graph Clustering Algorithm on Intel Xeon Phi Coprocessors*. In Proceedings of the 2nd International Workshop on Network Data Analytics, page 6. ACM, 2017.

[Tatineni 2016] Mahidhar Tatineni, Xiaoyi Lu, Dongju Choi, Amit Majumdar and Dhabaleswar K. (DK) Panda. *Experiences and Benefits of Running RDMA Hadoop and Spark on SDSC Comet*. In Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale, XSEDE16, pages 23:1–23:5, New York, NY, USA, 2016. ACM.

[Toshniwal 2014] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal and Dmitriy Ryaboy. *Storm@Twitter*. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.

[Turner 2014] Stephen Turner. *Capitalizing on Big Data: Governing information with automated metadata*. Journal of Technology Research, vol. 5, page 1, 2014.

[Veiga 2016] Jorge Veiga, Roberto R Expósito, Xoán C Pardo, Guillermo L Taboada and Juan Tourifio. *Performance evaluation of big data frameworks for large-scale data analytics*. In Big Data (Big Data), 2016 IEEE International Conference on, pages 424–431. IEEE, 2016.

[Veldt 2018] Nate Veldt, David F Gleich and Anthony Wirth. *A Correlation Clustering Framework for Community Detection*. In Proceedings of the 2018 World Wide Web Conference on World Wide Web, pages 439–448. International World Wide Web Conferences Steering Committee, 2018.

[Vernica 2012] Rares Vernica, Andrey Balmin, Kevin S. Beyer and Vuk Ercegovac. *Adaptive MapReduce Using Situation-aware Mappers*. In Proceedings of the 15th International Conference on Extending Database Technology, EDBT '12, pages 420–431, New York, NY, USA, 2012. ACM.

[Vlasblom 2009] James Vlasblom and Shoshana J Wodak. *Markov clustering versus affinity propagation for the partitioning of protein interaction graphs*. BMC bioinformatics, vol. 10, no. 1, page 99, 2009.

[Wadkar 2014] Sameer Wadkar and Madhu Siddalingaiah. Apache ambari, pages 399–401. Apress, Berkeley, CA, 2014.

[Wang 2007] Fei Wang and Changshui Zhang. *Label propagation through linear neighborhoods*. IEEE Transactions on Knowledge and Data Engineering, vol. 20, no. 1, pages 55–67, 2007.

[Wen 2017] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang and Xuemin Lin. *Efficient structural graph clustering: an index-based approach*. Proceedings of the VLDB Endowment, vol. 11, no. 3, pages 243–255, 2017.

[White 2005a] Scott White and Padhraic Smyth. *A spectral clustering approach to finding communities in graphs*. In Proceedings of the 2005 SIAM international conference on data mining, pages 274–285. SIAM, 2005.

[White 2005b] Scott White and Padhraic Smyth. *A spectral clustering approach to finding communities in graphs*. In Proceedings of the 2005 SIAM international conference on data mining, pages 274–285. SIAM, 2005.

[White 2012] Tom White. Hadoop: The definitive guide. " O'Reilly Media, Inc.", 2012.

[Wu 2019] Changfa Wu, Yu Gu and Ge Yu. *DPSCAN: Structural Graph Clustering Based on Density Peaks*. In International Conference on Database Systems for Advanced Applications, pages 626–641. Springer, 2019.

[Xie 2013] Jierui Xie and Boleslaw K Szymanski. *Labelrank: A stabilized label propagation algorithm for community detection in networks*. In 2013 IEEE 2nd Network Science Workshop (NSW), pages 138–143. IEEE, 2013.

[Xin 2013] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin and Ion Stoica. *GraphX: A Resilient Distributed Graph System on Spark*. In First International Workshop on Graph Data Management Experiences and Systems, GRADES '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.

[Xu 2002] Ying Xu, Victor Olman and Dong Xu. *Clustering gene expression data using a graph-theoretic approach: an application of minimum spanning trees*. Bioinformatics, vol. 18, no. 4, pages 536–545, 2002.

[Xu 2007] Xiaowei Xu, Nurcan Yuruk, Zhidan Feng and Thomas AJ Schweiger. *Scan: a structural clustering algorithm for networks*. In Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 824–833. ACM, 2007.

[Xu 2015] X. Xu, Q. Z. Sheng, L. J. Zhang, Y. Fan and S. Dustdar. *From Big Data to Big Service*. Computer, vol. 48, no. 7, pages 80–83, July 2015.

[Yao 2014] Yi Yao, Jiayin Wang, Bo Sheng, Jason Lin and Ningfang Mi. *HaSTE: Hadoop YARN Scheduling Based on Task-Dependency and Resource-Demand*. In Proceedings of the 2014 IEEE International Conference on Cloud Computing, CLOUD '14, pages 184–191, Washington, DC, USA, 2014. IEEE Computer Society.

[Yin 2015] ChuanTao Yin, Zhang Xiong, Hui Chen, JingYuan Wang, Daven Cooper and Bertrand David. *A literature survey on smart cities*. Science China Information Sciences, vol. 58, no. 10, pages 1–18, 2015.

[Yin 2017] Hao Yin, Austin R Benson, Jure Leskovec and David F Gleich. *Local higher-order graph clustering*. In Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 555–564. ACM, 2017.

[Zaharia 2010] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker and Ion Stoica. *Spark: Cluster Computing with Working Sets*. In Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[Zaharia 2012] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker and Ion

Stoica. *Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing*. In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, pages 2–2. USENIX Association, 2012.

[Žalik 2018] Krista Rizman Žalik and Borut Žalik. *Memetic algorithm using node entropy and partition entropy for community detection in networks*. Information Sciences, vol. 445, pages 38–49, 2018.

[Zeng 2015] Jianping Zeng and Hongfeng Yu. *Parallel modularity-based community detection on large-scale graphs*. In 2015 IEEE International Conference on Cluster Computing, pages 1–10. IEEE, 2015.

[Zhang 2010] Bingjing Zhang, Yang Ruan, Tak-Lon Wu, Judy Qiu, Adam Hughes and Geoffrey Fox. *Applying twister to scientific applications*. In Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on, pages 25–32. IEEE, 2010.

[Zhang 2015a] Fan Zhang, Junwei Cao, Samee U. Khan, Keqin Li and Kai Hwang. *A Task-level Adaptive MapReduce Framework for Real-time Streaming Data in Healthcare Applications*. Future Gener. Comput. Syst., vol. 43, no. C, pages 149–160, February 2015.

[Zhang 2015b] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan and Meihui Zhang. *In-memory big data management and processing: A survey*. IEEE Transactions on Knowledge and Data Engineering, vol. 27, no. 7, pages 1920–1948, 2015.

[Zhang 2017] Shuhao Zhang, Bingsheng He, Daniel Dahlmeier, Amelie Chi Zhou and Thomas Heinze. *Revisiting the design of data stream processing systems on multi-core processors*. In Data Engineering (ICDE), 2017 IEEE 33rd International Conference on, pages 659–670. IEEE, 2017.

[Zhao 2017] Weizhong Zhao, Gang Chen and Xiaowei Xu. *AnySCAN: An Efficient Anytime Framework with Active Learning for Large-Scale Network Cluster-*

ing. In 2017 IEEE International Conference on Data Mining (ICDM), pages 665–674. IEEE, 2017.

[Zhou 2009] Yang Zhou, Hong Cheng and Jeffrey Xu Yu. *Graph clustering based on structural/attribute similarities*. Proceedings of the VLDB Endowment, vol. 2, no. 1, pages 718–729, 2009.

[Zhou 2017a] Jinhong Zhou, Chongchong Xu, Xianglan Chen, Chao Wang and Xuehai Zhou. *Mermaid: Integrating Vertex-Centric with Edge-Centric for Real-World Graph Processing*. In 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pages 780–783. IEEE, 2017.

[Zhou 2017b] Lina Zhou, Shimei Pan, Jianwu Wang and Athanasios V. Vasilakos. *Machine Learning on Big Data*. Neurocomput., vol. 237, no. C, pages 350–361, May 2017.

Analysis and Mining of Large Dynamic Graphs: Graphs clustering application

Abstract: Recently, the graph clustering has become one of the most used techniques to understand structures and inherent knowledge in graph data. This trend progressively attracts the attention of companies and research community. For example, in the industrial field, it is used for multiple applications like social networks (e.g. Facebook), where communities can be modeled as clusters in a graph. As for collaborative networks (e.g. DBLP), a cluster can represent a team with similar research interests. Several works have been established where their proposed approaches are based on advanced algorithms mainly graph clustering algorithms and modularity based-ones. The former has demonstrated their efficiency notably by providing supplementary information about clusters in a list. Besides, they can identify hub and outlier vertices. Despite their importance, the utility of such algorithms is limited by their high complexity particularly when dealing with Big and dynamic graphs. This limitation motivates us to propose new algorithms with higher performances in our thesis. For more details, our contributions can be summarized in the following points: (1) carrying out of a comparative study between the most popular Big Data platforms (2) proposing a distributed algorithm called DSCAN for large graphs clustering and (3) extending DSCAN to develop an incremental algorithm for dynamic and large graphs. A comparative study between our proposed algorithms and other baselines has shown their effectiveness and their scalability when dealing with large and dynamic graphs

Keywords: Dynamic graph clustering, graph mining, community detection, Big Data, graph clustering, big graph processing.