Optimisation Techniques for Finding Connected Components in Large Graphs Using GraphX

MAHER TURIFI

University of Salford

School of Computing, Science and Engineering

Submitted in Partial Fulfilment of the Requirements of the Degree of

Doctor of Philosophy

2017

Chapter 1: Contents

List of Figures

List of Tables

List of abbreviations

BFS	Breadth First Search
BSP	Bulk Synchronous Parallel
CC	Connected Components
DFS	Depth First Searches
DHT	Distributed Hash-Table
GAS	Gather-Sum-Apply-Scatter
HPC	High Performance Computers
HDFS	Hadoop Distributed File System
MPI	Message-Passing Interface
PGM	Property Graph Model
PRAM	Parallel Random-Access Machine
RDD	Resilient Distributed Dataset
RDF	Resource Description Framework
TLAV	Think Like A Vertex

Abstract

The problem of finding connected components in undirected graphs has been well studied. It is an essential pre-processing step to many graph computations, and a fundamental task in graph analytics applications, such as social network analysis, web graph mining and image processing. Recently, it has been a major area of interest within the field of large graph processing. However, much of the research has focused on solving the problem using High Performance Computers (HPC). In large distributed systems, the MapReduce framework dominates the processing of big data, and has been used for finding connected components in big graphs although iterative processing is not directly supported in MapReduce. Current big data processing systems have developed into supporting iterative processing and providing additional features other than MapReduce. Moreover, current connected component algorithms in large distributed processing system only use the traditional approach to choosing the component identifier based on the lexical ordering of the node ID value. This study investigates how to enhance the performance of finding connected components algorithm for large graph in distributed processing system. It uses the approach to considering the graph degree property in choosing the component identifier, reviewing how this can affect the efficiency of the algorithm. In the design of our proposed algorithm features provided by current new processing systems such as moving the computation more toward the data partition in Spark are integrated. This study thus review how this has affected the performance. The degree approach to choosing the component identifier is experimentally tested using different algorithms. The study then applies the proposed approach on the fastest existing algorithm, and experimentally compare the performance of the connected component algorithm using both the original and our modified algorithm. The results show that using the degree approach has played a vital role in the evolution of the graph size during the process, leading to a faster convergence and significant performance improvement when case vertex pruning is applied in the algorithms. Furthermore, they demonstrate that in many cases optimising the design of the algorithm with local pre-processing of the data has resulted in performance enhancement.

Chapter 1: Introduction

Big Data was the buzzword of the year 2013[1]. Ever since, the trend towards adopting big data processing system has increased and it is commonly seen in every aspect of life[2]. Therefore, it has become important for a wide range of scientific and industrial processes, especially as the cost of storage is decreasing and the ability to capture different kind of data is growing[3].

In view of the diversity of data acquired nowadays and the massive amount of data stored, there is a need to find new ways to deal with data beyond the traditional database, such as handling data that do not usually fit in a single machine memory or disk [4][5]. One approach is to look at data as a network or a graph with edges connecting things together, those edges can take different forms of relationships. This metaphor of graph is currently used in many areas: computer science, economics, sociology, biology, and many more[6]. Almost anything can be represented as a graph [7]. Figure 1-1 shows LinkedIn knowledge graph, where "entities" on LinkedIn, such as members, jobs, titles, skills, companies, geographical locations, schools, etc, and the relationships between them are represented using graph.

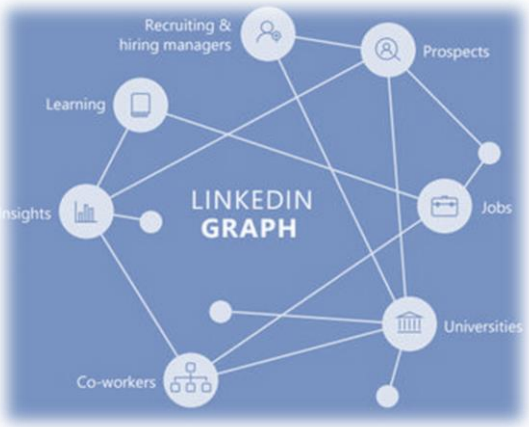


Figure 1-1: Example of LinkedIn knowledge graph [1]

[1] https://www.linkedin.com/pulse/machine-learning-linkedin-knowledge-graph-qi-he/

Graphs are considered to be a very flexible data model that can be used to express relationships between entities, and to recognize local and global characteristics of the system, and to analyse different features of the complex networks[8] [7] [9].

Extensive research has been carried out on graphs and graph processing, as graphs are one of the most widely used data representations and have been extensively used to efficiently process data and extract knowledge[10][7]. However, recent graphs are beyond the ability of traditional systems to handle, either because the sizes of current graphs are very big, and they usually do not fit in a machine's memory, or because current algorithms cannot process such graphs efficiently, particularly when using the current distributed systems[11].

Our focus in this research is on the problem for finding Connected Components (CC) efficiently in an undirected graph. A component represents a graph (or subgraph) where any two vertices inside that graph are connected via paths, and there is no edge that connects any vertex outside the component[6]. This problem has been well studied, as it is an essential pre-processing step to many graph computations, and is a building block in complex graph analysis such as clustering[12][13][14]. It has been a major area of interest within the field of large graph processing and much of the research so far has focused on solving the problem using High Performance Computers (HPC), with high computation power and equipped with very large memory capacity[15].

Large-scale graphs (or big graphs) are usually stored using a distributed file system, like Hadoop, either in the cloud or locally[16]. Hadoop[17] is an open-source framework that allows for the distributed storage and processing of large datasets across clusters of commodity computers. The MapReduce[18][19] framework dominates the processing of large-scale data on Hadoop, and it is commonly used for mining big graphs[20]. However, iterative processing is not directly supported in MapReduce. Nonetheless, some recent

works[21][22] show that it is possible to outperform other processing models for finding connected components using MapReduce. Yet, only a few studies have investigated this problem in big data distributed system using MapReduce[14].

1.1 Research Motivation, Aim, and Objectives:

Our work builds on the knowledge that current big data processing systems have become more advanced with features beyond MapReduce. For example, a processing system, like Spark[23][24], supports iterative processing and provides additional features other than MapReduce such as data partitioning and caching. Spark also supports graph processing using GraphX[25], which is an open source Spark API for graph-parallel computation. Moreover, current connected component algorithms in large distributed processing system only use the traditional approach in choosing the component identifier for each connected component based on the lexical ordering of the node ID value. Hence, the aim of this research is to enhance and optimise the performance of finding connected components in large undirected graphs using MapReduce.

This could be achieved by addressing the following questions:

- Based on the *Graph Structure* properties, how to increase the efficiency of the algorithm by considering graph structure properties in choosing the component identifier?
- Based on the *Processing System* properties, how to increase the efficiency of the algorithm in modern processing systems using the new features provided beyond MapReduce?

To achieve this research aim, the following objectives have been identified:

1. To adopt a new approach to enhance the performance of CC algorithms by considering the graph structure in choosing the component identifier. In our case, we use the degree property in choosing the component identifier for each connected component.
2. To enhance the performance of the algorithm by using the features provided by the new current processing systems. In our case, we based our optimisation on the concept that moving the computation process towards where the data is stored could help enhance the performance. This is essentially the concept behind MapReduce also. However, we could benefit further from systems like Spark that provide extra features by caching the data in memory and controlling the partitioning process across the cluster.
3. To review current MapReduce algorithms for finding connected components and choosing the most recent one that outperforms other algorithms to implement proposed optimisations.
4. To use the best practices and design patterns that have been proven to be efficient in implementing MapReduce algorithms.
5. To apply the new approach by developing an algorithm for finding connected components in large-scale graphs and implementing it in the distributed processing system (GraphX/Spark).
6. To evaluate the efficiency of the new algorithm by analysing its performance using known tested graph datasets and experimentally compare the performance against fastest existing algorithms.

1.2 Research Methodology:

Our approach for finding connected components in large scale graphs is based on the degree of the nodes in the graph and not only the ID of nodes as it is commonly used. Development and implementation would be applied using GraphX on Spark, an open source Spark API for graph-parallel computation. The research will apply some of the best practices and design patterns that have been proven to be efficient in implementing large-scale graph mining algorithms in MapReduce. Spark's ability to cache some parts of the data in memory for use in later iterations will be used. This could help to decrease both the number of iterations and the intermediate communication load between iterations, and eventually greatly enhance performance. This could be achieved without losing the ability to expand a graph that does not fit in memory where Spark can split the cache file on local disks with the need to compute again.

We adopt an empirical approach method in our research to estimate the effectiveness and the efficiency of all techniques applied to enhance the performance of finding connected components algorithm. The approach implemented is tested using large synthetic and real-world datasets. An overview of our methodology is presented in figure 1-2, with the corresponding chapter in this thesis.

Initially, we start by exploring Big Data and give overview of the main processing systems and techniques used with it (Chapter2). In particular, we focus on Apache Hadoop system and the MapReduce programming model and its limitations, in addition to Apache Spark as they dominate the big data processing systems currently used. Next, we review graphs and big graphs, and focus on big graph processing systems (Chapter 3). Our focus in this research is on the problem of finding connected components in large graphs, and more specifically

using a distributed processing system. Thus, we review available algorithms that use the MapReduce programming model (Chapter 4).

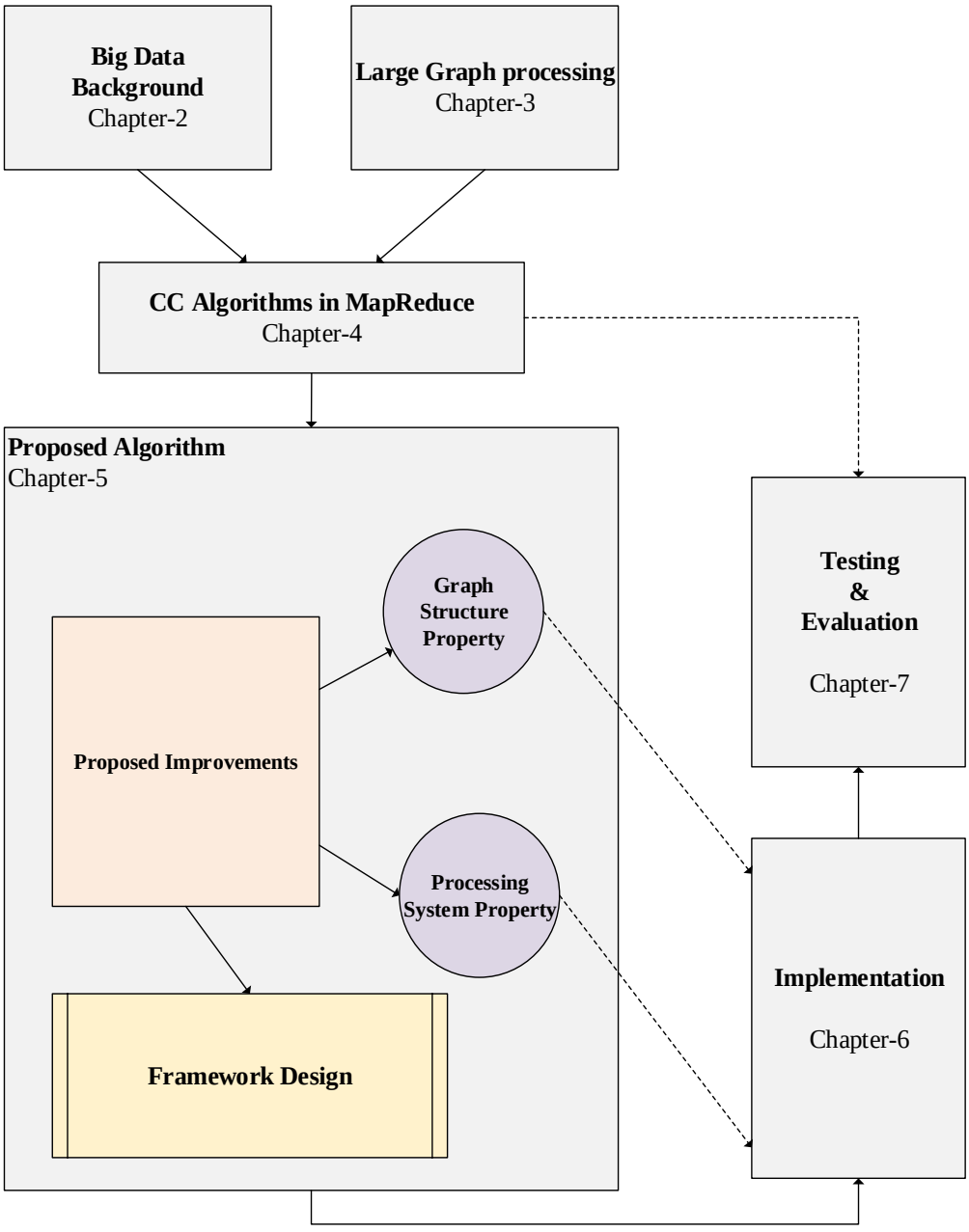


Figure 1-2:Overview of the research methodology followed in this thesis

Then, we introduce our proposed approach in trying to fulfil the objective of optimising and enhancing the performance of the connected components algorithm, and approach it from two angles; using graph structure degree property, and using properties of the processing

system, such as caching and partitioning in Spark. Afterwards, we design the framework model for our algorithm (Chapter 5). For implementation, we use the open source system Spark, namely its graph processing library GraphX, to implements each enhancement introduced in the designed framework (Chapter 6). Next, we test and evaluate our implementation using synthetic and real-world datasets and compare the results to the results of other algorithms. When evaluating the optimisation, we compare our implementation results with the results of the original unmodified algorithm. For choosing the datasets, we used open public datasets that often appear in the evaluation of similar algorithms from related researches. All our tests run in a cloud environment using a virtual cluster (Chapter 7). Finally, we conclude and summarize the result of this study and describe some of the limitations faced in the process of conducting this research, and then we suggest future work to overcome some limitations or to further extend this research (Chapter 8).

1.3 Contributions:

The contribution in this thesis are as follows:

(i). **Using the node degree approach in finding connected components algorithm**:

using the degree approach in choosing the connected component identifier will always result in less number of iterations until convergence, however it adds some overload on the system due to the extra work required to calculate the degree for each node and the increased size of messages due to the attachment of the degree to the node. Nonetheless, this approach showed significant performance improvement when applied to algorithms which apply vertex pruning; where unuseful nodes for the computation are excluded from the process after each iteration. In this kind of algorithms (Cracker in our case) the number of iterations decreases and the size of graph shrinks faster when this approach is applied, leading to better runtime.

(ii). **Using the local computation for connected components approach:**

Moving more computation towards where the data is stored, and trying to apply computation on a data partition before the need to do computation on the cluster can effectively improve the performance of the algorithm. In the case with the Cracker algorithm, despite the inconsistency in results, in general there is a noticeable performance improvement especially in the *seed propagation phase* for the larger datasets. This approach should to be wisely considered and implemented as it could increase the load on the system and lead to performance degradation.

(iii). **Considering different level of computation in the design of the algorithm.**

In big data processing system operations are applied at different level, by identifying the level of processing, and integrating them in the process of the algorithm design can help to increase the efficiency of the algorithm. For example, start by processing the data partition, then process the collective data of partitions inside a cluster worker node, and finally process all the data at the cluster driver node. Customising operation in the algorithm for each level could increase the performance of the algorithm. In this study, processing has been customised and applied on the data partitions in the cluster driver nodes. However, additional operations could be added to process the data inside a cluster worker node using multi-core structure of the cluster nodes.

(iv). **Guidelines to be implemented in different context**

It is worth noting that one of the major contribution of this work is to encourage active researcher in the field to consider features provided by the current new processing systems in the design of their algorithms using MapReduce. This could be considered as useful guidelines to be implemented in different context.

1.4 Outline of the Thesis

The content of this thesis is structured as follow: Initially, explore big data processing systems and techniques (Chapter2). Next, review graphs and big graphs processing systems (Chapter 3). Review available algorithms for finding connected components using MapReduce (Chapter 4). Then proposed our approach to enhancing the performance of the connected components algorithm (Chapter 5). Describe the implementation process for the proposed enhancement on the fastest existing algorithm (Chapter 6). Next, experimentally test and evaluate the proposed approach using both the original and modified algorithm implementation on synthetic and real-world datasets (Chapter 7). Finally, present conclusions and some limitations and future works.

Chapter 2: Big Data Background

2.1 Big Data:

2.1.1 Big Data definition

"Big Data" was described as the BUZZWORD for the year 2013-2014 [1], as the discussion about it is growing with the expectation that the digital universe of data would reach over 35 Zettabytes in 2020 [3][26].

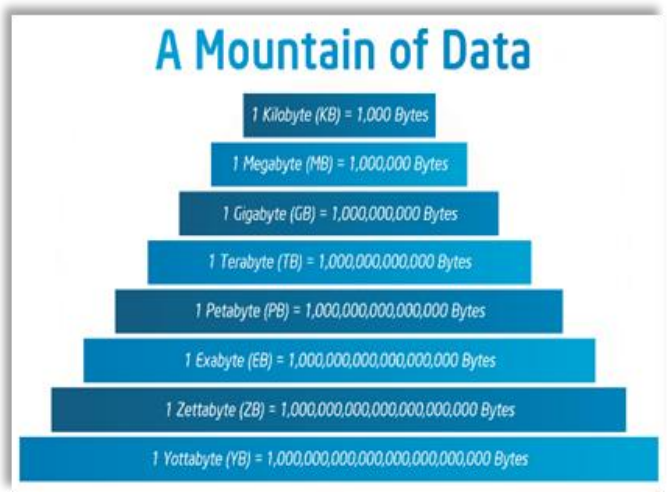


Figure 2-1: A Mountain of Data represent by multiple of the unit byte[27].

In 2015, it was estimated that data reached close to 8 Zettabytes, with a network of 15 billion connected devices. This ocean of data could be imagined as 18 million libraries of Congress, which are 462 Terabytes each [28]. (*See Figure2-1)*

For example in the field of social media, the daily generated data in 2011 by Facebook is 10 Terabytes (TB) and by Twitter is 7 TB [3], and in 2012 it was reported that Facebook social graph contains over a billion nodes and more than 140 billion edges[26]. Multimedia places a huge load on the internet. Google alone has more than a million servers around the world. In 2013, Over 6 billion mobile subscribers in the world send 10 billion SMS every day[29].

In 2016 there were 300 hours of video uploaded to YouTube every minute and more than 1.3 billion unique visitors and over 3.25 billion hours of video watched each month. This is expected to increase in 2017[30]. In order to exploit the value of this huge amount of data, organizations must consider three things:

1. Data usually has the characteristics of continuous flow.

2. Analysing the data now is a job that requires significant skill. This is where a *Data Scientist* is needed, i.e. a professional in analytics and IT who has a deep understanding of the field being investigated and with the management skills and ability to effectively communicate with decision makers.

3. Real and appropriate outcomes will need both business users and IT people to work together when analysing large-scale data[31] [32].

2.1.2 What is Big Data?

Big Data is broadly defined as data that is too big, fast, and hard to deal with using conventional database tools [4][5].

A more technical view is provided by Katal et al.[33], Hunter[34], Kraska[1], Chaudhuri[35] who define Big Data as data that requires new technologies and architectures. This is because the database management tools or traditional data processing applications are unable to process the data in a timely, cost effective way, because it is too large to be stored and processed and too complex and varied to be analysed and visualised[36]. However,

Venkatram et al. argued that the definition of Big Data varies between organisations and people based on the data characteristics and the use cases of the data analysis[37]. The name "Big Data Analytics" is also given to the process of research into Big Data to disclose hidden and secret patterns [29].

2.1.3 Components (Three Vs & +V)

Early in 2001 Doug Laney presented the 3Vs concept in a published research note about the three challenges of increasing data: data Volume, Variety and Velocity [38].

Later the Three V's became the main components or characteristics that are used to explain what Big data is [33][29][39][40] (figure 2-2).

Figure 2-2: Big Data Components, the 3 Vs[2]

- ***Volume***: is the word associated with "BIG" in big data. It includes the increasing massive amount of data collected and produced and goes beyond the ability to hold and process easily.
- ***Variety***: data come from many sources. These include, for example, web logs, sensor data, social media data, emails, images, documents and audio. Data in general comes in three types: structured, semi-structured and unstructured. Data Variety is probably the hardest to manage when processing a large amount of data.
- ***Velocity***: is concerned with the speed of the data coming from various sources. For example, streaming data and sensor data or data that is required to be handled in real-time.

[2] https://www.datameer.com/images/product/big_data_hadoop/img_bigdata.png

In addition to the main 3Vs some researchs [29][33][40] introduced extra Vs, that could relate to specific business needs and which depend on how the data would be used to facilitate business decision-making[37]:

- *Value*: how useful is the data in finding useful insights that helps in making better decisions [33] [40] .
- *Verification*: ensuring appropriate data security and that added value should be made to the organization [29] .
- Components such as Veracity, Validity and Volatility were also introduced[37].

According to Madden [4] in this explosion of data and the process to adopt the Big Data's three Vs, some commercial Relational Databases managed to handle the ***volume*** problem (e.g. Greenplum, Teradata, Vertica). On the other hand, open source systems such as MySQL and Postgres were unable to manage this problem. However, both commercial and open source traditional database systems struggle with the ***velocity*** and ***variety*** problems. Furthermore, they are not efficient when handling streaming data and lack statistics and modelling support adoption.

As a result, many research projects attempted to fill the gap between data analysis and data processing. They usually adopted three approaches:

1. Extending the relational model – for example projects by Oracle and Greenplum.
2. Extending the MapReduce/Hadoop model for example projects like: Apache Mahout, Spark, HaLoop, Twister, and Daytona.
3. Building new systems – for example projects like GraphLab and SciDB.

2.1.4 Big Data Benefits

A huge amount of data will be provided to be investigated and analysed by applying big data analytics in different fields and many sectors. Therefore, rapid advances and discoveries in many disciplines are expected, in addition to the success and increasing profits for many enterprises [39]. Big data Analytics can be financially beneficial as well as helping an organization to have deeper insights into its data whilst enabling faster decision by processing the data in real time and moving the data processing to where it is stored. When data scientists and IT experts work closely with business users more efficient solutions to the problems being studied become possible[28] and this helps decision makers to make better-informed decisions and develop better strategies[41].

Sagiroglu & Sinanc[29] listed some business benefits that arise when applying big data analytics. These include more focused marketing, more direct business insights, client based segmentation, discovery of market opportunities, and automated decision making.

2.1.5 Associated Challenges with Big Data:

Big Data promises beneficial opportunities. However, to be achieved many challenges must to be addressed. Kraska [1] divided big data issues into:

1. *Big Throughput*, which concerns the problem associated with storing and manipulating a large amount of data.
2. *Big data analytics- which are* those issues related to transforming data into knowledge.

Bhatia & Vaswani [39] highlighted the following issues that appear during each phase of big data analytics: issues of scale, heterogeneity, lack of structure, error-handling, privacy, and visualization.

For a successful Big Data project, questions about data integration, volume, skill availability and solution costs should be considered [42]. Katal et al. [33] brought into light various challenges and issues associated with adopting Big Data solutions. These are:

(a) Technical challenges, which include issues like, fault tolerance, scalability, quality of data and heterogeneous data.

(b) Storage and processing issues.

(c) Analytical challenges.

(d) Skill requirements.

(e) Privacy, security, data access and sharing of information.

Issues related to privacy, security, data access and sharing of information are very sensitive issues that all need to be well addressed [33][34] as Big Data Applications could be used for malevolent intent and will not be in an organization's best interest. For example, by aggregating enough information about individuals from their environment with other information from different sources such as social media, an intrusive profile that has considerable personal information about an individual could be built [43].

2.2 Big Data Technologies:

Many projects have attempted to develop a distributed system that can handle large-scale data. For example:

- **Hadoop** [17], is a project to develop open-source software for reliable, scalable and distributed computing.
- **Naiad** [44]**,** is Microsoft system for data-parallel dataflow computation that focusses on low-latency streaming and cyclic computations.

- **Apache Spark**[23], [24]**,** is an open source cluster computing system that has in-memory nature and aims to make data analytics fast.
- **HPCC** (High Performance Computing Cluster)[45], is an open source massively parallel-processing computing platform that solves Big Data problems.
- **Prege**l [46]**,** is a framework for processing large graphs in which nodes exchange messages between each other and update their own states in memory. It has an efficient, scalable and fault-tolerant implementation on clusters of thousands of commodity computers.
- **Storm** [47]**,** is a free open source distributed real-time computation system. Storm makes it easy to reliably process unbounded streams of data and can be used with any programming language.
- **S4** [48], is a distributed, scalable and fault-tolerant system for processing continuous unbounded streams of data.

There are other approaches available, which differ according to the problem area or the application they were designed to address. However, *Hadoop* is the most dominant platform for distributed processing and many other projects were built using of Hadoop's framework. Projects can also work side by side with it or use the Hadoop Distributed File System (HDFS).

2.2.1 Hadoop & MapReduce:

Therefore, this study will describe Hadoop and explain in more detail its programming model MapReduce with an example. It will then discuss some MapReduce limitations and detail some of the alternatives available.

Dealing with massive amounts of data is a reality. New software has developed, starting with the development of a new file system that can handle large files [49]. MapReduce was

subsequently proposed by Google [18][19] as a programming model to deal with large datasets in scalable and distributed fashion [50].

Hadoop is an open-source framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is based on MapReduce. HDFS was developed for reliable, scalable, distributed computing [17]. It allows working with thousands of computers and dealing with petabytes of data [16].

HDFS (Hadoop Distributed File System) is based on the Google File System [49]. It operates on commodity computers to store data across hundreds of computers. *Data nodes* will host files, files are divided into chunks (usually 64 megabytes size), which are replicated on different disks (usually three times, one disk should be on a different rack). A *Master node* has a directory that records where each file is stored and replicated [51].

MapReduce gives the programmer the advantages of not needing to consider the details of data distribution, parallel executing, replication and load balancing. Its programming concept is familiar [52] and allows parallelised and distributed execution for jobs across clusters of computers. It requires two functions [51].:

1. *Map function*, which is defined by the programmer to process Key-Value data. each chunk or more of a given data will be processed by the map function and gives output as key-value pairs. During the shuffling phase, pairs are collected by a master controller and sorted by their keys value. They are then divided among reducers in such a way that each group with the same key goes to the same reducer.
2. *Reducer function*, takes the key-value pairs and combines all the values associated with the same key and carries out any computation defined by the programmer. It then outputs the new value. The reducer output could be in key-value pairs to feed another mapper in an iterative way

The Hadoop cluster is at least one machine running the Hadoop software. In each cluster, there is a single master node with a varying number of slave nodes. Slave nodes can act as both the computing nodes for the MapReduce and as data nodes for the HDFS. This is illustrated in figure *2-3*.

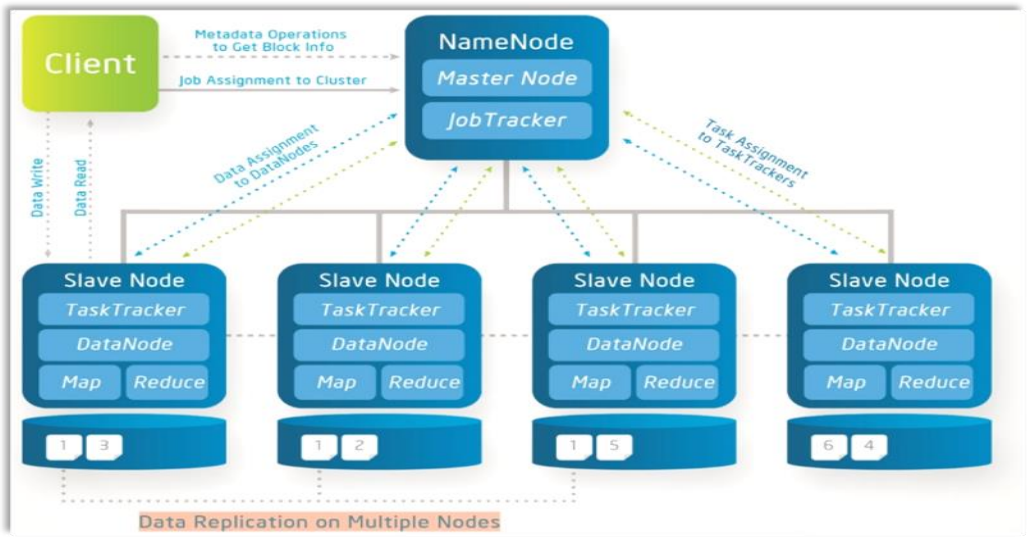


Figure 2-3: Hadoop HDFS and MapReduce[3]

A client submits a job to the master node, which manages it with the slaves in the cluster. JobTracker controls the MapReduce job, reporting to TaskTracker. TaskTracker will process the map or reduce operations task. Once the map function has finished a task, the output is sorted and divided into several groups, which are distributed to the reduce functions. Reducers may be located on the same node as the mappers or on another node. TaskTracker reports to JobTracker when it finishes a task. JobTracker then schedules a new task for TaskTracker [28] [53].

Apache Hadoop is in continuous development and is used in both commercial and research sectors[53]. Many packages have been developed to run on Apache Hadoop. These include: Ambri, Avrp, Cassandra, chukwa, Hbase, Hive, Pig, Spark, Tez, ZooKeeper and others.

[3] https://blogs.oracle.com/financialservices/big-data:-from-hype-to-insight-part-2-infrastructure-and-technology

Hbase is a column oriented, scalable, distributed database. Pig is a high-level language and Mahout is a scalable data mining library [17].

Yarn[54] is a resource manager for managing distributed applications which separated cluster resource management capabilities from the original MapReduce. It gives Hadoop better reliability, availability and improved cluster utilization. It also supports programming paradigms besides MapReduce (figure 2-4).

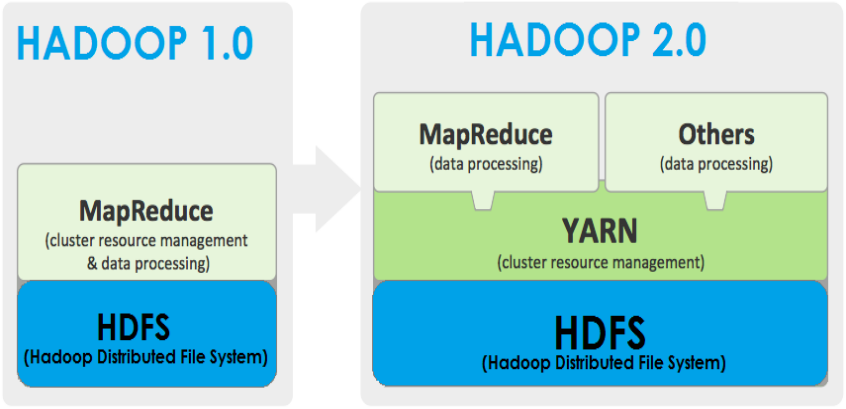


Figure 2-4: Apache Hadoop with YARN[4].

[4] https://hortonworks.com/webinar/yarn-code/

i. Word Count Example:

MapReduce will be explained using a word count example shown in figure 2-5. The example assumes a collection of documents files Doc1, Doc2, Doc3, and each document has a textual content v1, v2, v3, respectively.

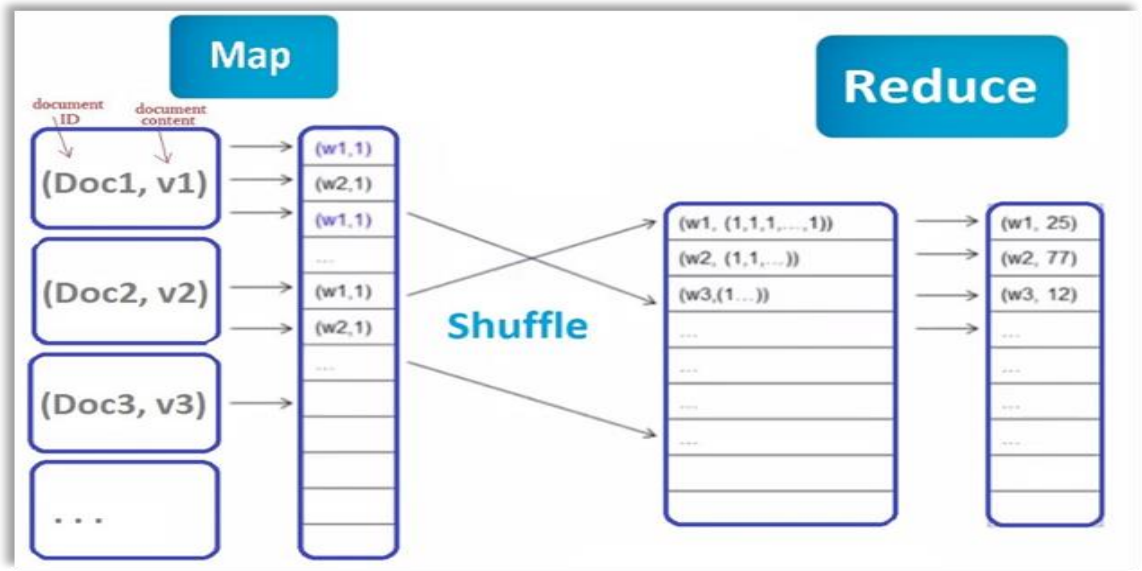


Figure 2-5: MapReduce word count Example

Initially the file is stored in the HDFS file system in chunks, (*see Figure 2-6*), where in this case each chunk is one document "Doc", and each chunk of data is passed to the Mapper.

In the *map phase* the map function (mapper) will divide the content of each document into words and emit a key-value message that has the word as a key and number 1 as a value (indicating that this word appeared once).

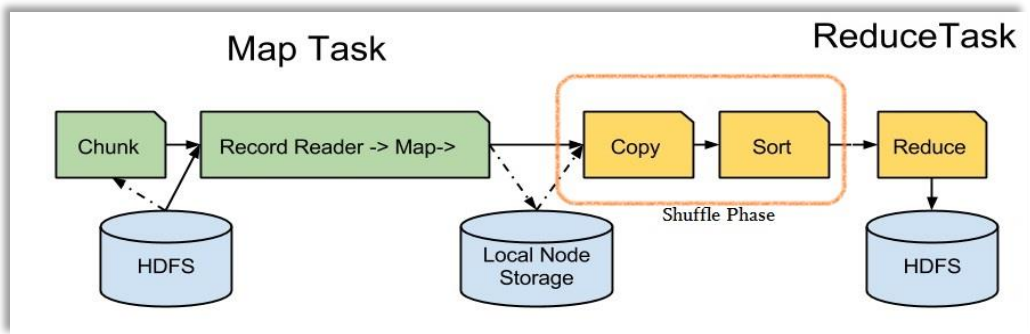


Figure 2-6: Map task and Reduce task in Hadoop

In the shuffling phase, the output from the mapper will be aggregated and sorted and all the messages that share the same key (which is the word here) will be sent to the same reducer. The intermediate results are stored locally (not in HDFS) as temporary files and then passed to the reducer.

In the reduce phase, the reducer will receive messages that each has a pair of a words and a list of values. In this case the reducer will sum all the values for each word to count the words. The output is stored in the HDFS. The pseudocode for word count example is shown in figure 2-7.

```
// count word example:
map(String input_key, String input_value):
        // input_key: document name
        // input_value: document contents
        for each word w in value:
                EmitIntermediate ( w , 1);
reduce(String intermediate_key, String intermediate_values):
        // intermediate_key: word
        // intermediate_values: ???
        int result = 0;
        for each v in intermediate_values:
                result+=v;
                Emit(result);
```

Figure 2-7: MapReduce count word example pseudo code

ii. MapReduce Alternatives

a) MapReduce Limitations

MapReduce is one of the most used paradigms for processing distributed file systems. MapReduce is very flexible as there is no schema or index, however this may give poor performance when compared to relational databases[55]. For low-latency processing systems it is not suitable as MapReduce computation uses batch processing unlike the stream computation which has continuous jobs[55]. Furthermore, much development is addressing the way it is implemented, as it is not efficient when applications require repeated MapReduce iterations. This is because MapReduce has no memory since it assumes the input is too large to fit in memory, and at each iteration it writes to three replicas in the distributed file system which is an overhead, The map tasks for subsequent iterations cannot begin until all the previous stages are complete [56]. Improving the performance of MapReduce and enhancing large-scale data processing have become a very important area of research, with MapReduce parallel programming being applied to many data mining algorithms [57].

b) MapReduce Evolution

Rajaraman & Ullman [51] identified three approaches to improve the performance of MapReduce:

- *Iterate MapReduce*: enhance iterated MapReduce run-time and make it more efficient by avoiding the data copy between each iteration and pipelining the output of the reducer directly into the map phase of the next MapReduce iteration. This approach has been added to Hadoop as an extension to support iterative algorithms. For example:
    - **Twister** [58], is an iterative MapReduce framework that provides a long running Map and Reduce tasks the "do not terminate after the execution of each iteration" capability It also differentiates between two types of data: variable data and static data which remain fixed throughout the computation

in each iteration (usually it is the larger of the two). The mapper in Twister will stream its output directly to the reducer[59].

- **HaLoop** [60], extends the Hadoop MapReduce framework by supporting iterative MapReduce applications, adding various data caching mechanisms and making the task scheduler loop-aware.
- **Tez** [61] is a project is aimed at building an application framework which allows for a complex directed-acyclic-graph of tasks for processing data which allow for dynamic performance optimizations[61]. It enables a user to run interactive jobs on the top of YARN.

- Generalize data-flow graph of MapReduce tasks. This generalizes the MapReduce paradigm to a system that supports any acyclic collection of functions, where map and reduce are simply two types of operations, each one can be instantiated by many tasks. Each is responsible for executing that function on a portion of the data. Examples of such data flow systems are: DryadLINQ[62],Naiad [44], Hyracks [63], Clustera [64].
  - **Spark** [23], [24], is a framework that supports iterative applications, it focuses on caching the data between different MapReduce-like task executions by introducing resilient distributed datasets (RDDs) that can be explicitly kept in memory across the machines in the cluster.
  - **Naiad**[44]**,** is a Microsoft system for data-parallel dataflow computation that focusses on low-latency streaming and cyclic computations.
  - **Stratosphere**[65], is an open-source software stack for analyzing Big Data. Stratosphere recently became an Apache project under the name Apache Flink[66]. it tries to bridge the gap and combine the flexibility of MapReduce and the efficiency of parallel DBMSs. It exploits in-memory data streaming and integrates iterative processing deeply into the system runtime, as it

introduces special kinds of iterations (delta-iterations) that can significantly reduce the amount of computation as iterations proceed.

- Direct Implementation of recursion in MapReduce [56] to try to solve the problem of recovering from non-blocking tasks failing, without the need to restart failed tasks. There are two main models:
    - *Graph based models* such as Pregel [46] and Giraph[67], by using the Bulk Synchronous Parallel (BSP) paradigm, which is considered more efficient than MapReduce for graph processing. However, it places a restriction of needing to have a combined memory size of the machines processing the graph larger than the graph size.
    - *Stream based models* such as S4 [48] or Storm[47].

2.2.2 Apache Spark

Apache Spark [23], [24] is a fast and general-purpose cluster computing system which has in-memory nature, It provides similar scalability and fault tolerance properties to MapReduce using high-level APIs in Java, Scala, Python, and R that enable interactively querying big dataset on clusters. In addition, it supports a set of tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming (shown in figure 2-8).

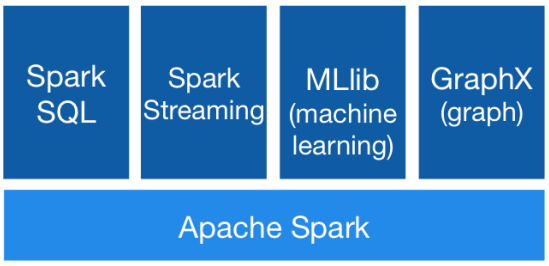


Figure 2-8: Apache Spark[5].

Spark evaluation shows a performance which is up to 20 times faster than Hadoop for iterative applications, speeds up a real-world data analytics report by 40 times, and can be used interactively to scan a 1 TB dataset with 5–7 seconds latency[68].

The main abstraction Spark provides, is a Resilient Distributed Dataset (RDD)[69], which is a read only collection of elements partitioned across the nodes of the cluster that can be operated on in parallel and can be rebuilt if a node is lost. In addition, it provides two shared variables:

(1) *Broadcast variables*, which only copied to each worker once and cache values in memory,

[5] https://spark.apache.org/

(2) *Accumulators*, in which workers can only add to using an associative operation such as counters and sums.

In Spark, developer writes a driver program that connects to a cluster of workers. In the driver program one or more RDDs are defined through transformations (e.g., map and filter) which are lazy operations create a new dataset from an existing one, then action operations are invoked (e.g., count, collect, save) to run the computation on the dataset and return a value to the driver (figure 2-9).

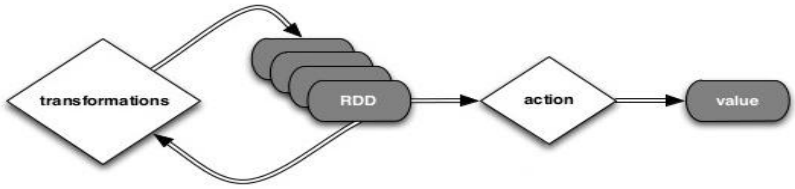


Figure 2-9: RDD Operations[6]

This design increases the efficiency of Spark as transformations are lazy and are only computed when the first time an action is used to return a value to the driver program. In addition, Spark can persist an RDD in memory and keep the elements around on the cluster for much faster access the next time it will be needed. Persisting can be on disk in case we want to save memory and we don't want a heavy processing operation to be recomputed.

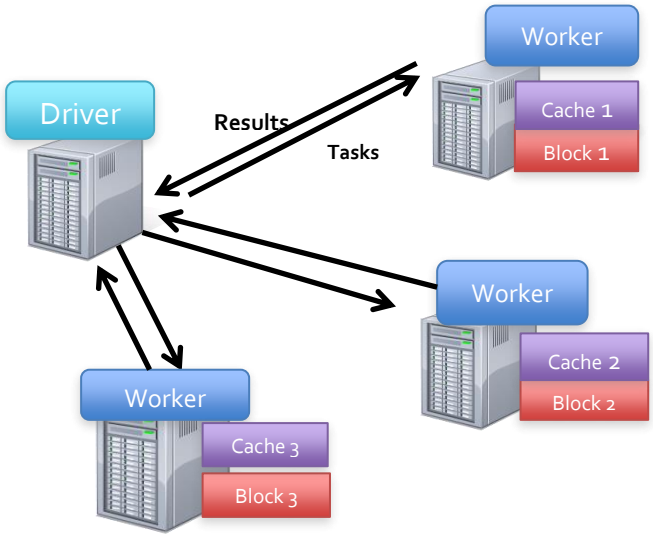


Figure 2-10:Spark System[69]

[6] https://spark.apache.org/docs/latest/rdd-programming-guide.html

Which could be significantly efficient in applications that need iterative algorithms and interactive data mining tools (figure 2-10).

As mentioned before RDDs [69] [23] is the main abstraction Spark used to perform in-memory computations on large clusters where RDD's elements are partitioned. RDDs created in transformation operation in four ways:

- From a file in a shared file system
- By "parallelizing" a Scala collection
- By transforming an existing RDD.
- By changing the persistence of an existing RDD, either by keeping it in memory or writing it to a disk.

When an action operation is invoked (e.g., count, collect, save) on RDD, Spark will build a directed acyclic graph (DAG) of stages to execute based on RDD's lineage graph. RDD has enough information about how to compute its partitions from data in stable storage it does not need to exist on a physical storage and achieves fault tolerance using a notion of lineage rather than the actual data, thus when data on a partition is lost it will automatically recovered just on that partition using the transformations that originally created it. In addition, RDD can be cached or persisted on the cluster for later reuse.

There are three options for storage of persistent RDDs:

- in-memory storage as deserialized Java objects,
- in-memory storage as serialized data,
- On-disk storage.

Internally, RDD interface are represented using five pieces: a list of partitions; preferred location of a partition; dependencies on parent RDDs; a function to compute based on its parents; metadata about how the RDD is partitioned. Two kinds of dependencies are distinguished between RDDs: narrow dependencies, where parent RDD is transformed to only on child RDD; wide dependencies, where multiple child partitions may depend on it. Knowing the type of dependencies in RDD helps to make efficient decision to recover a node failure, as with a narrow dependency only the lost parent partitions need to be recomputed (figure 2-11).

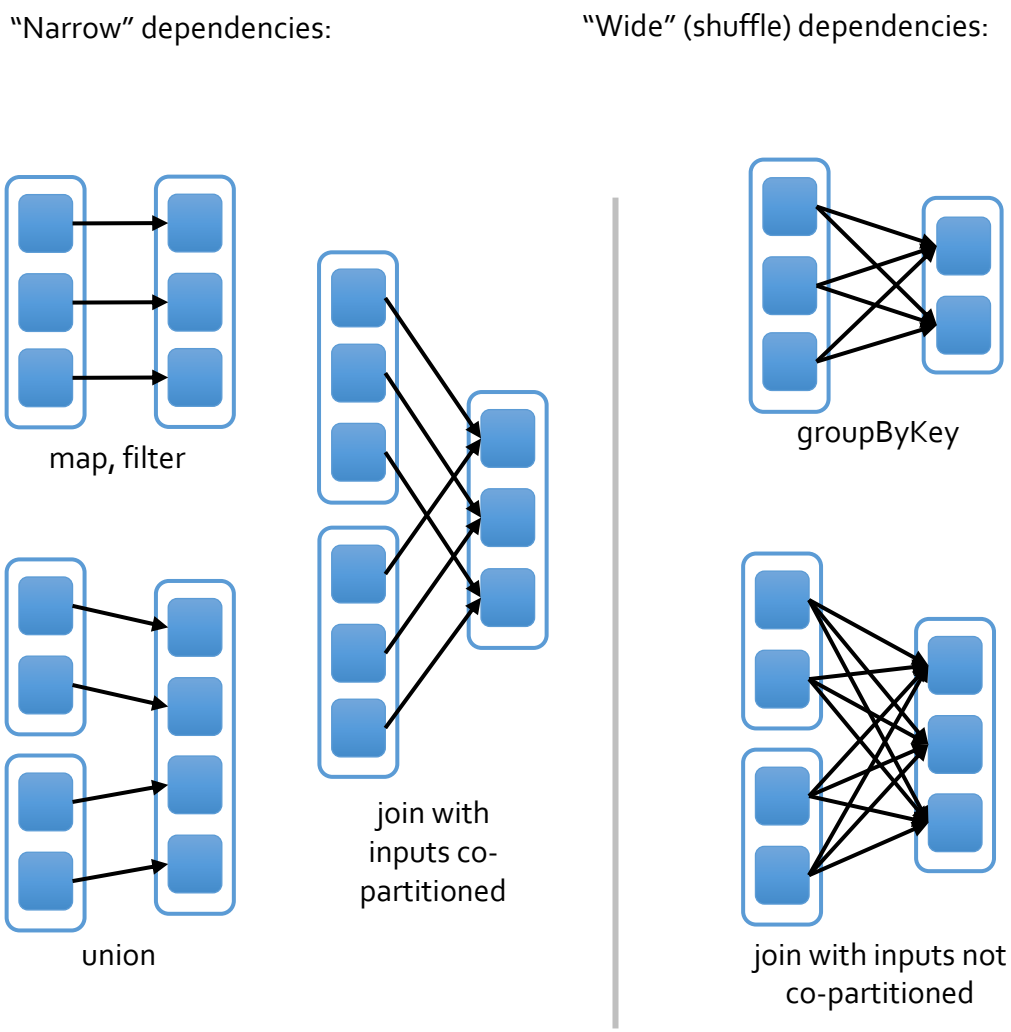


Figure 2-11: RDD dependencies[69].

Spark use RDDs to perform in-memory computations on large clusters, similarly to what Distributed Shared Memory (DSM) do. However, RDD has many advantage over DSM such as: DSMs use checkpoint to roll back the whole program upon failure. On the other hand, RDDs can be recomputed in parallel on different nodes using lineage. Spark can detect slow nodes and use RDDs to run backup copies on different nodes. When RDDs does not fit in memory can be stored on disk in a similar way to MapReduce.

RDDs evaluated and used to express a number of other cluster programming frameworks and help to optimise performance by caching wanted data in memory, partitioning it to minimize communication, and providing efficient fault tolerance. For example, Spark can express MapReduce model using *flatMap* and *groupByKey* operations, or *reduceByKey* if there is a combiner. Furthermore, RDDs can be persisted in memory to simply and efficiently implement Iterative MapReduce model such as **HaLoop**[60] and **Twister**[58] through a series of MapReduce jobs to loop[69].

Currently Spark is one of the most widely used open source processing engines for big data. It provides rich language-integrated APIs with a wide range of libraries, and both the usability and performance of Spark are continuing to improved[70].

Chapter 3: Literature Review:

3.1 Graphs

3.1.1 Introduction to Network

Over the past decade the way we live and work has enormously changed due to the boost advances in technologies and the increase in complexity of the communication systems available, this has been reflected mainly in how we become dependent on such technologies and systems. Moreover, it was predicted that this trend will continue according to the Gartner's report "Top 10 strategy technology trends for 2015"[2], which shows that there has been an increase in adoption and investment in new concepts such as Internet of Things (IoT), where billions of everyday devices or equipment will be connected to the Internet using smart machines where smart technologies and devices are evolving rapidly. In another word, our real world is merging with the virtual world by going beyond many of the geographic limitations and especially with the increased interactions with social network systems. In addition, our environment is becoming more intelligent, with the mass volume of data generated; analytics is now deeply embedded everywhere seeking for a better and smarter understanding.

With the huge amount of data collected, there is an urgent need to efficiently deal with it and extract knowledge that no one has discovered before. One approach is to look at this data as a network with links connecting things together, those links can take different kind of forms of relationships. This metaphor of networks is currently used in many areas: computer science, economics, sociology, biology, and many more. It can effectively address many of the challenges in each area by understanding the "connectedness" of these complex systems. By "connectedness" two aspects are considered, (1) *Graph Theory*: the study of the network

structure - who is connected to whom, and (2) *Game Theory:* the study of strategic behaviour in the network - by understanding each individual action in correlation with everyone in the systems and how the system will react to this action [6].

3.1.2 Graph Theory:

Modelling the relationship in a network by graphs helps to generate a natural human interpretation and simple mechanical analysis [10]. This concept is not new, back few centuries in 1736 Leonhard Euler in his paper on the Seven Bridges of Königsberg laid the foundation of graph theory. Since then mathematicians extensively studied graph and its properties [71].

Almost anything can be represented as a graph [7], if a system contains many single units interacting with each other through a certain kind of relationship, each node of the graph stands for one of the units of the system and relationships between different units are indicated by edges[72]. Graph are considered to be a very flexible data model that can be used to express relationships between entities, and to recognize local and global characteristics of the system, and to analyse different features of the complex networks [8][7][9].

Graphs have been used to understand complex human and natural phenomena [73]. In general, graph is used in any domain when there is a need to find a network representation of logical or physical links between entities. Its applications spread on wide variety of domains such as: linguistics, economics, sociology, biology, chemistry, and pharmacology (e.g. graphs model the complicated structure of chemical compounds and protein structures), and computer science (e.g. Worldwide Web, workflows, XML documents, computer networks, physical connections, computer vision, video indexing, text retrieval and social

networks) and many more, where graph algorithms have been developed to solve different kinds of problems [7] [8] [74] [75].

3.1.3 Definition

With the diversity of data acquired nowadays, a need to find a way to deal with data beyond the multi-dimensional model used in traditional database. **Graph** is way to represent structured and heterogeneous data as set of objects that are linked to each other in different ways. A Graph G = (V, E) is consists of set of nodes V (Vertices) that are connected with each other by links called edges E. Usually Graph represented as directed, undirected graphs, with weighted edges and nodes, tree graphs, and in many variants. It's used to help studying the relationship between objects such as paths, positions, associations, sequences and structures[8] [6] [11].

3.1.4 Characteristics

According to the survey conducted by IBM [8], many graph techniques and algorithms has been developed showing how data is represented, interoperated and analysed. Usually graph algorithms categorised as follow:

1. Structural algorithms (network analysis algorithms), that try to understand the structure of the network and analyse the relationships between network entities and explore topological properties of a graph, such properties as:
    a) Order and Size: the number of nodes and edges.
    b) Degree: the number of edges incident for the node, in-degree which is the number of incident on the node, out-degree which is the number of incident from the node.
    c) Distance: the number of edges in the shortest path between two nodes.

d) Diameter: the longest of all shortest paths between two nodes.

e) Girth: the length of the shortest cycle.

f) Connectivity coefficients: the minimum number of nodes that when removed the graph will be disconnected.

g) Clustering coefficients: a measure to show how nodes cluster together.

h) Centrality: which determines the importance of a node in a graph. The four main measures of centrality are:

    1) Degree Centrality: the degree for that node normalized with the total number of edges.

    2) Closeness Centrality: is a measure for distance between a node and all other nodes in the graph.

    3) Betweenness Centrality: is a measure for the number of shortest paths go through a node divided by the number of shortest path in the graph.

    4) Eigenvector Centrality: is a measure for the importance of a node in the graph.

2. Traversal algorithms, which navigate paths in a graph to solve problems such as: (a) *route problems* by trying to optimize path lengths under certain conditions. (b) *Flow problems,* for example, investigate flow of oil or gas over a directed graph. (c) *Coloring problems* as partition the graph by labelling its entities. (d) *Searching problems* by traversing nodes to answer query or find a problem stated.

3. Pattern-matching algorithms, by finding different graph patterns (e.g., cliques, cycles, sub-graphs, network motifs) in a graph. Interesting application for this type of algorithms include: social analytics, organizational analytics, epidemiology, financial network modelling, pharmacology, and neuroscience.

Applying the mining algorithms on graph data is a challenge, as the graph miners need to adapt or redesign their algorithms to be able to handle the new measures and properties of graphs and be able to store, query and explore graphs in a similar way as in traditional databases. Because of the fact that the structure of data is different, it hard to defined graph measures and properties using classical data mining algorithms. In addition to the fact that usually recent graphs are very big and does not fit in memory to be handled using traditional mining algorithms[11] .

3.2 Big Graph

According to Skhiri [11], nowadays there is an urgent need to deal with structured, unstructured, and heterogeneous data instead of the traditional one. Thus, graphs are now widely used because of its expressive power and the ability to connected object in different way. However, mining is hard to implement because of the structure of the data, and size of the data as the real-world graphs are very big, and usually does not fit in a machine memory. Adding to that, there is no single model that efficiently fits all the types of graph algorithms and application, nonetheless many have been developed to solve specific problems or to meet some special classes of applications.

3.2.1 Big graph History:

Graph processing is not new, it is a well-investigated area of research, in addition big graph has been always a problem. However, the perception of defining how large is big graph has been evolving.

Usually the big graph problem was dealt with by adding more power to the system (*Scaling up*); increase the processing power, adding more core and more memory. Thus, the move

was toward using High-performance computing (HPC) using shared memory parallel systems, which is an active area of research and development.

Nowadays, a rising trend to capture and store any data available, especially as the cost of storage decreasing and ability to capture different kind of data increase. Pushed by the world getting more connected; more connected devices and embedded sensors and expanding networks and others all contribute to found the area of the Internet of Things (IoT); in addition, people life is more digital nowadays than ever before, and there is increasing presence of social media in our life (Facebook, Twitter, Snapchat, Instagram, and many more).

The existing real-world dataset is getting large enormously, these datasets reflect different kind of relationships and can be generally efficiently represented using graph structures. However, as the graph grow larger their size and complexity go beyond single processing machine ability and make processing it with HPC systems a challenging task which is not always suitable for it[14].

Appearance of the MapReduce concept and its implementation in Hadoop equipped researchers with a powerful tool to process large graphs, and a new trend toward processing large graphs in cluster using distributed systems with commodity hardware raised (*Scaling out*).

It is challenging to ensure the traditional set of properties ACID (Atomicity, Consistency, Isolation, Durability) in graph database because of the different data structure, as a result, new tools and data models were developed to adapt to the new data structure. For example, new querying languages proposed like, Datalog, Xpath, Gremlin, Cypher, SPARQL to provide graph support and give richer queries[76].

Likewise, new techniques for handling large graph processing developed, Skhiri[11] introduce three categories: (1) high-performance graph DB such as DEX or Titan, (2) in-memory and HPC/MPI graph processing such as SNAP, and (3) distributed approach based on Bulk Synchronous Processing (BSP) such as Pregel. Hence Graph Management Systems (GMS) solutions developed could be categorised as[56], [77]:

a) Transactional GMS such as: Neo4j (centralized graph database), Jena, HyperGraphDB, RDF3x.
b) Analytic GDM such as Pregel (open source implementation Giraph)
c) Both such as Trinity, Horton, Titan (distributed graph database).

3.2.2 Big Graph Systems categorisation

The development in graph processing has recently flourish especially with unprecedented amount of data acquired and captured, in addition it is motivated and inspired by the latest advances in big data processing. Pushed by the big data processing move, many systems have been developed to process, manage and analyse graph data.

We could identify two main categories in the big graph systems:

- ***Graph Databases systems:*** which is a database founded on the graph structure (vertices and edges with their properties) to represent the graph data and store it, and provide the means to query and retrieve data efficiently.
- ***Distributed graph processing systems:*** which provide the ability to do graph analytics using iterative processing algorithms in distributed manner on cluster more efficiently and reliably than the graph database systems can do.

3.2.3 Big graph system requirements:

Junghanns and Guerrieri [76] [78] both indicate that in order to have systems that can flexibly manage big graphs and can efficiently analyze them the following requirement should be met:

1. The graph systems should be adaptable with powerful graph model that is not restricted to fixed schema, but it would be able to process graph with heterogeneous vertices and edges with different kinds of data and provide tools to process and analyze it.
2. Provide a powerful query language to retrieve and analyze graph data, and support processing complex graph analysis jobs.

3. High performance and scalability in graph systems should be offered, to achieve that in graph databases the emphasis is on how to support query optimization, indexing and efficient graph storage that can expand as the size of the graph increase. On the other hand, with distributed graph processing systems, the main focus is on how to efficiently implement graph operator and partition the large graph in a distributed cluster, in addition offer expanding processing power when needed by expanding the cluster by increasing the number of nodes.
4. Providing persistent graph storage and offering support for ACID compliant transactions on persistent data, reading it, analyze it, and storing it back in distributed systems.
5. Graph processing system should not offer the user hard and complex experience when analyzing the data, instead it should offer powerful tool to query the data, in addition the ability to interactively explore the data and visualize it.
6. Failure is most likely to happen in big clusters and it is very important for the system to be resilient to failure so the computation will continue even when a node or process fail.

3.2.4 Graph databases

Graph databases are used to store data that is based on graph structure with Create, Read, Update, and Delete (CRUD) methods, they provide graph operators which are designed to enhance the performance on graph transformation and computation. They are generally designed to be used with online transactional processing (OLTP) systems, where special optimizations for performance, integrity, and availability are considered[79].

Usually one or more graph data model is supported in Graph Databases. A graph data model is the conceptual representation that is used to model the real world entities and the relations among these entities as a graph[79], [80].

- Majority of graph databases support the *Property Graph Model (PGM)*, in which a set of key-value pairs can be associated with any vertex or edge in a directed multigraph, however only edges with one start vertex and end vertex are permitted.
- From the Semantic Web movement comes another graph model *Resource Description Framework (RDF)*, it has in its structure collections of triples (subject-predicate-object), where vertices are (subjects, objects) and edges are (predicates). These triples form a directed labelled multigraph [76].
- Only a few graph database systems use the *Generic Graph Model* called Hypergraph in which it supports arbitrary user-defined data structures to be attached to vertices and edges. In contrary to PGM, in hypergraph It is permitted to have edges with any number of vertices at each end similar to many-to-many relationships in traditional databases[79]. Such systems provide flexibility to model another graph models, but also restrict the ability to provide optimized operators for graph transformation and computation [76].

In a recent work by Junghanns [76], he reviewed some of the most recent graph database in term of their data model, application scope, and storage technique for graph in large scale. Mainly they support either PGM or RDF but some like IBM SYSTEM G[7], Stardog[8], and Blazegraph[9] support both of them, the last two store PGM using RDF. Only few support the Generic graph model such as IBM System G and HypergraphDB[10]. In term of *application scope*, many focus on providing transaction and querying functionality where only small part of the data is accessed. However only few provide support for graph analytics where whole data need to be processed; such systems either provide built-in algorithms for graph analytics or provide the ability to run custom graph algorithms on the database such as Blazegraph. Apache Tinkerpop[11] gives both the functionalities of graph databases (OLTP) system and graph analytic system for the user in one system by virtually integrating graph-processing system in graph database. Furthermore, Tinkerpop also offer Gremlin[12] graph traversal query language. In term of storage, some use the native approach where data is stored in graph-optimised form such as adjacency list. In non-native approach, the graph database is implement on the top of other systems such as relational or document-oriented database, OrientDB[13] is an example. Other systems such as IBM SYSTEM G and Titan[14] offer choice of different kind of storage options[78].

[7] http://systemg.research.ibm.com/
[8] http://docs.stardog.com/
[9] https://www.blazegraph.com/
[10] http://hypergraphdb.org/
[11] https://tinkerpop.apache.org/
[12] http://tinkerpop.apache.org/gremlin.html
[13] http://orientdb.com/orientdb/
[14] http://titan.thinkaurelius.com/

3.3 Big Graph Processing Systems

Big graph processing is very new growing trend, nonetheless it is very active research area in academia with leading companies (Google, Facebook, Microsoft, IBM, and many others) investing and pioneering in its development. However, the development move is not in one direction, one direction is to manage large scale graphs using single machine, the other is efficiently process graph algorithms on parallel systems, and the last one is inspired by the big data move and based on processing big graph in a distributed system, which our main focus here.

Development of distributed big graph systems impose extra challenges other than those inherited from big data processing, as a consequence of to the irregular structure of graphs and its algorithms. For example, it involves extensive communication and message passing between vertices due to its iterative processing nature, adding to that graph algorithms are most likely to be explorative where whole data need to be processed. Furthermore, computation performance can be seriously affected by different partitioning strategies because of the problem of load balancing and increased communications between nodes in cluster[81].

3.3.1 Features of Big Graph Processing Systems:

We can possibly categorize the big graph systems along multiple dimensions, as a consequence of the existence of many features that play rules in defining system performance and its application domain. Based on the work done by [82][83], the key features can be discussed under the following headings, which are:

i. Graph Programming Model:

A graph programming model is a way to abstractly specify the underlying computing infrastructure components like interface, methods, events that helps in describing graph data structures and algorithms. Kalavri[82], [84] distinguished between two levels of graph programming model:

- High-level programming model: in which the programmer can mainly focus on the logic of the algorithm rather than computing environment where graph partitioning and communication mechanisms are hidden. however, he will have less degree of control and limited ability to do customisation [82].
- Low-level programming model: in which programmer have more flexibility on the control of computing environment and more degree to customisation than the previous model, however at the price of losing simplicity and user-friendly programming interface provided in high-level programming model. Usually this programming model is used to address a specific class of graph problems, and generally can handle arbitrary graph computations[83].

ii. Expressiveness:

Here, expressiveness means the clarity in identifying system performance advantages and recognizing good example of its application classes, in addition to clarifying any problematic example cases or hidden costs or presumed conditions.

iii. Timing Execution model:

Generally, in big graph systems, graph algorithms run in iterations until convergence or they reach a termination condition. Yan et al [83] described two modes: synchronous and asynchronous.

- In ***synchronous execution model,*** a global barrier separates between iterations, where vertices can only have access to information from the previous iteration, in another word, all updates and outgoing messages are only available in the next iteration after the barrier. This model is also known as the Bulk Synchronous Parallel (BSP) model and was introduced by Valiant[85].
- In ***asynchronous execution model,*** vertices have direct access to the last updates happened during the same iteration, in another word, view of the most recent state of the graph. This model could effectively improve the performance for algorithms in the case where some vertices could converge very much faster than other, as in the case of PageRank algorithm. Though, this come at the cost that an approximate result is produced instead of the exact result produces in synchronous model, which is in many cases considered as good results as in PageRank. However, asynchronous model is not applicable in many algorithms where approximate results are not accepted. Furthermore, more work should be done to assure data consistency issues such as 'Race condition' where different attempts to update vertex's value at the same time could happen.

Kalavri [82] discussed two extra models furthermore. ***Hybrid execution model***, where the previous two models exist in the same system. ***Incremental execution model***, where system can apply updates as it arrived and change the current state of the graph without the need to do recomputing of the whole process.

Inspired by BSP model, Pregel system [46] can process big graph using synchronous execution model, GraphLab[21] adopts the asynchronous execution model, while Giraph++[86] use the hybrid model by using synchronous model with the computation and communication inside partition and asynchronous model for computation and communication between partitions. Another example for hybrid execution model (Hsync) is

PowerSwitch[87] that switches between the two models according to predictions for future optimal performance, it constantly collects execution statistics to help in predicting the when a model switch could be profitable.

iv. Communication mechanism

According to Kalavri [82] the following models are distinguished:

- ***Message-Passing model***: a vertex updates to other vertices' local states (data values) can only happen by sending and receiving messages. Additionally, Yan [83] talked about to two modes of the message-passing communication model for a vertex:
    a. Edge-based communication: where a vertex can only communicate with its neighbours via the edges connecting them, so one hop at a time.
    b. ID-based communication: where a vertex can communicate with other vertices via the vertices' ID even if it is not directly connected to them, the sender here needs to know the ID of the receiver.
- ***Shared memory mechanism***: by shared memory here it does not mean a shared memory maintained across machines (like in PRAM), however it means vertices can have a direct access to the state of other vertices and edges by keeping data in main memory and access it asynchronously [83]. Therefore, vertex only has access to its neighbours' values. Here, more work should be done to assure data consistency and consider issues such as 'Race condition'. This abstraction is adopted in two kind of distributed systems as in systems GraphLab and PowerGraph, and in single machine for processing big graph as in GraphChi, X-Stream & Chaos, and GridGraph**.**
- ***Dataflow model*:**

  The data flows in graph from a stage to another using stateless operators where no state is maintained and processing is done one by one (such as using filter and map functions) over a cluster of distributed compute nodes. Thus it is usually difficult to

implement graph algorithms and achieve good performance, yet systems such as Apache Spark and Apache Flink try to overcome this issue by efficiently implementing cashing mechanisms[82].

v. Other features:

To further evaluate distributed big graph systems and its **application** domain, the algorithm used in testing should be chosen to efficiently represent performance efficiency and its implementation issues. Kalavri [82] surveyed some of the most recent applications in distributed graph processing systems papers (34 examined systems), and grouped graph algorithms and their most commonly used applications and sorted them by appearance frequency. Moreover, according to Yan [83] identifying the **Execution environment** of a system and what processing power it has and all resources available will help to evaluate its performance in compared to other systems and identify its ideal application domain. Such an environment could be distributed, as in a cluster of machines, or a single machine environment using a commodity PC machine or high-performance computer (Supercomputer). For example, in commodity PC machine where the memory is limited, big graph processing is done by reading the graph piece by piece from the storage in streaming manner, whereas for supercomputer the graph could be loaded in memory. Similar to single-machine PC systems where there is limited memory and the graph is kept on disk, some distributed system processes the graph on disk where the graph size is larger than the collective size of memory on the cluster, such as Pregelix[88], GraphD[89].

It is important to understand that in general, considering all these features in big graph systems makes evaluating and comparing between systems a very complex task. An open research question is, what combination of these feature can best enhance the performance related to a specific algorithm?

Our main objective for now is to review the current programming model at high level of abstraction. Although low-level abstraction gives programmer more flexibility in controlling the computing environment and more degree to do customisation, however the high-level abstraction provides simplicity and user-friendly programming interface that helps to represent algorithms and to some extent without the need to worry about data partitioning techniques and communication mechanisms used in the background. So programmer can easily implement graph algorithms, which is sometimes considered a more important feature than pure performance [83]. In addition, it allows system developer to implement some automatic optimisation when applicable.

3.4 Approaches in the Developments of Big Graph Processing Systems

With graphs getting bigger in size, systems have to deal with the problem of limited memory capacity within a single machine, and even with solutions such as partitioning the graph and processing it from storage disk or from a distributed memory, scalability and parallelization are hard to implement. Due to graph algorithms' nature such as iterative processing and explorative random-access patterns and extensive communication cause extra overhead and complexity to the system to process and analyze such graphs[90].

There are many systems developed to solve the problem of big graph processing in a single machine. Which requires high-performance machine with large memory capacity to fit the graph, but this solution is expensive and not always efficiently scalable, the other solution is to store the graph or part of it on disk which also result in a lot of reading from the disk and therefore negatively affect the performance.

In this study, our focus is on graph systems that are built in a distributed environment. The following is a brief overview of the most common distributed graph programming models at high level of abstraction.

Distributed Graph Systems could be categorized in two main categories[76]:

- Dedicated Distributed graph processing systems: include vertex-centric approaches such as Google Pregel and its variations and extensions including Apache Giraph, GPS, GraphLab, Giraph++ etc.
- Distributed graph dataflow systems: are graph-specific extensions (e.g., GraphX and Gelly) of general-purpose distributed dataflow systems such as Apache Spark and Apache Flink.

i. Dedicated Distributed graph processing systems:

Here systems are specially designed for distributed graph processing, it is built around the graph data structure and optimized to easily represent and implement graph algorithms. In the following we describe high-level programming models for the distributed graph processing system based on the categorization made by Kalavri et al.[82], these are: vertex-centric, scatter-gather, gather-sum-apply-scatter, subgraph-centric, filter-process, graph traversals model. We will go into details talking about vertex-centric, as it is the dominant model used for graph processing.

a) Vertex-centric (Think like a vertex):

Vertex centric was developed to deal with the issue of iterative nature of a board set of graph algorithms, in addition to make graph analysis programs easier to develop and more efficiently perform[78]. The main concept is to express computation from a vertex point of view where a user-defined program (*vertex function)* is iteratively executed on graph vertices, where the programmer only need to define the behaviour of only one vertex.

Therefore, it is also known as “Think Like A Vertex” approach (TLAV). The vertex-centric model shows that it is a good fit to represent a wide set of graph algorithms, especially when the algorithm computation is mainly related to a vertex and its adjacent neighbours, such as iterative value propagation algorithms and fixed-point methods algorithms[82]. TLAV was first introduced by Google in a Pregel paper[46], which is based on Bulk Synchronous Parallel (BSP) parallel programming model[85].

Bulk Synchronous Parallel (BSP)[85]

BSP was developed as conceptual model that address scalability challenges to efficiently execute parallel program and algorithms across nodes. It was introduced as “a bridging model for parallel computation” to facilitate the design of software for parallel hardware. BSP uses message-passing interface (MPI) to address challenges such as high latency reads, deadlocks, and race conditions[91].

Conceptually, there are three main phases they need to execute iteratively[78]:

- *Components Parallel Computation*: where each component executes specific tasks on its local data.
- *Communication phase*: where components exchange messages among themselves, using to the results from the components parallel computation phase.
- *Synchronisation barrier phase* (superstep barriers): this phase makes sure that all components have finished the previous two phases, only then do synchronization between the components participating at the superstep barriers.

The first two phases (*Components Parallel Computation* / *Communication phase*) together called a ***superstep***, each superstep is followed by a synchronization of the parallel tasks reaching at the superstep barriers.

BSP model is not new concept, but only recently BSP model gained a lot of attention as a graph framework, and has been implemented on almost the majority of recent distributed frameworks to define graph algorithms[92], [93].

Furthermore, considering the communication mechanism, the systems based on vertex-centric model can be divided into two categories:

- Vertex-Centric Message Passing Systems:
- Vertex-Centric Systems with Shared Memory Abstraction:

Vertex-Centric Message Passing Systems:

Where vertices communicate with each other by sending messages. Pregel is the most known system based on vertex-centric and use message passing for communication between vertices:

Pregel

Based on MapReduce[18][94][19] Google started developing a new system with the aim to efficiently process large graphs and do graph analysis. inspired by the BSP model they created Pregel[46] in C++. Which provided a native API for programmer to develop algorithms based on the TLAV model while hiding the complexity of communication and data distribution.

In Pregel, the graph is partitioned and the vertices are distributed on the cluster where each vertex and its neighbours are located on the same node, to preserve data locality (computation is done locally).

By analogy, each vertex is a component in BSP model that has a state (value) and initially all vertices are active. Each vertex will exchange messages with its set of neighbours and update its value according to a user-defined function (vertex *function*). Pregel computation proceeds in iterations (supersteps), in each superstep, each vertex executes the vertex

function ***vf(msg)*** which take a message input ***msg*** which is the incoming messages from the previous superstep. During the executing of the vertex function each vertex may do any of the following:

- Update its value.
- Send a message to other vertices
- Deactivate itself (vote to halt) when no messages are received, so it will not run the next superstep unless it was reactivated by receiving a message.

In Pregel supersteps are synchronous, as each superstep will end with synchronisation barrier, that grantees that all the active vertices in that iteration have finished computation and all the message exchange between them has finished. If an active vertex did not receive any message, it will be deactivated and if an inactive one received a message it will be activated. In the next superstep only the active vertices will run the vertex function. When all the vertices are inactive the process will terminate.

Based on Master/Worker architecture in distributed system, where the graph is partitioned across the cluster nodes (workers) and each node will load its graph portion in memory or processing. The master node responsibility is to do the synchronisation process at each superstep barrier[81].

Pregel Like Systems

Pregel system was developed and used by Google and it is not open source project, it is not available outside Google. Thus, a lot of alternative were developed to fill that gap in big

graph processing area based on the Pregel system and inspired by the "Think Like A Vertex" model, such as: Apache Hama[15], Apache Giraph[16], GPS[17], Pregel+[18], Pregelix, and Mizan.

The vertex-centric with message passing computing model allows programmer to design and implement scalable distributed algorithms easily and debug their code, while the system handles all the low-level details. However, systems based on this model, usually comes with few limitations in performance[82][83]:

- For Non-iterative and asynchronous graph algorithms, it challenging to express using vertex-centric model, in addition algorithms such as Graph transformations and single-pass graph algorithms do not fit for this model, such as Triangle counting.
- For algorithms where some vertices converge faster than others (asymmetrically), there is no priority given for specific vertices over others, in addition no priority also given for local message over remote messages and superstep will only finish when the slowest worker has finished, therefore communication can be often encountered in the vertex-centric message-passing model.
- Concurrency is limited by the global barriers and could cause unnecessary synchronization that will slow down the process.

Yan et al [83] discuss how some systems such as Maiter, GiraphUC have been developed in order to avoid some of the limitations mentioned before.

Vertex-Centric Systems with Shared Memory Abstraction:

Where vertices communicate with each other using shared memory programming abstraction, which does not mean a shared memory maintained across machines (like in PRAM), however it means vertices can have a direct access to the state of other vertices and

[15] https://hama.apache.org/
[16] http://giraph.apache.org/
[17] infolab.stanford.edu/gps/
[18] http://www.cse.cuhk.edu.hk/pregelplus/

edges by keeping data in main memory and access it asynchronously[83]. Therefore, vertex only has access to its neighbours' values. Here, more work should be done to assure data consistency and consider issues such as 'Race condition'. This abstraction is adopted in two ways: as in distributed systems such as in GraphLab and PowerGraph, and when a single machine is used for processing big graph as in GraphChi, X-Stream & Chaos, and GridGraph.

b) Scatter-Gather

Also known as Signal/Collection model [95]. It shares the same philosophy a "Think Like A Vertex" and uses the message-passing for communications. It provides an elegant and concise abstraction for describing some graph algorithms, such as value-propagation algorithms. Vertices interact with each other by means of signals messages that go through edges. Vertices then collect signals and update the vertex state according to the old state and all signals received. Here, the superstep is divided into two phases, which are scatter and gather. In *scatter,* a user-specified function will produce the messages, and in *gather,* a user-specified function will update its value using the received messages[76]. One of the limitations is that, in scatter phase, there is no access to the received message on gather phase unless all messages received are stored in the vertex value during the gather phase, which will often require more memory and increase the complexity of the implementation [82].

c) Gather-Sum-Apply-Scatter (GAS)

It was proposed in PowerGraph[96] as a solution to efficiently process power-low graphs. Usually in vertex-centric model, when only a few vertices have high degrees, most of the computation work will be on those vertices, which cause bottlenecks and slow the execution time. GAS model addresses this issue by distributing the computation more efficiently across the cluster. It divides the vertex program into four separate phases, each will execute a user-specified function. During the gather phase, user function is applied on each edge. The output

is aggregated for each vertex using an associative and commutative user-defined function in the *sum* phase. The result from the sum phase together with the current value of the vertex are passed to the *apply* function, which will define the new value for the vertex. In the *scatter* phase, a function is applied on the updated values for each vertex and new messages generated on each edge, which will be processed in next iteration. The last phase in this model (*scatter phase*) is optional, where a variant model is introduced and called Gather-Sum-Apply (GSA). The GAS model help to balance the computation workload and reduce the amount of network traffic[76]. However, it requires high memory and communication overhead during the computation on low-degree vertices. To solve this problem, the *differentiated vertex computation model* is introduced, where high-degree vertices can be processed using the GAS model, and low-degree vertices can be processed using one of the vertex-centric models (such as using GraphLab) [82][83].

d) Subgraph centric (or Block-centric)

It is also known as "Think Like A Graph"[86]. It was developed to deal with the issue high communication overhead with fine-grained abstraction such as TLAV [82], [84]. It is a coarse-grained abstraction for distributed graph programming, which is considered a subgraph as a computation block. Blocks can internally process vertices, update their values, and exchange messages between them internally. In addition, blocks can externally interact with other blocks through messages exchange. It can help reduce the number of messages between cluster nodes and decrease the number of iterations for vertex-centric algorithms [97]. In vertex-centric model, the user-specified function run on each vertex independently from the others. However, in subgraph- centric model, the user-specified function takes as an input all the vertices inside the subgraph and processes all of them together [76]. Two distributed graph processing models could be categorised based on the subgraph centric model:

Partition-Centric Model

When graph is stored on a distributed system it is partitioned across the cluster into small partitions. This approach consider stored graph partitions as subgraphs of the input graph[86]. Where vertices can interact freely with each other inside the same partition, which can reduce the communication and help to achieve a faster convergence[78]. Performance in this model is highly determined by the quality of the partitions[82]. An example of such processing model is GIRAPH++[86], it was developed as an extension to GIRAPH[19][67]. This work was further optimised in BLOGEL[98].

Neighborhood-Centric Model

In this approach, custom subgraphs of the input graph are defined. Usually, the subgraph is determined based on vertices and their multi-hop neighbourhoods. A partition could contain more than one subgraph. This model is preferred for analysis that requires multi-hop local neighbourhoods in large graphs, such as the analysing ego networks [99]. However, determining the subgraphs in this model is an expensive process in term of both execution time and memory [82].

e) Filter-Process Mode

It is also called "think like an embedding". It was introduced in ARABESQUE[100]. An embedding is a subgraph instance of the input graph that is dynamically generated during the process that matches a user-specified pattern. Computations proceed through a sequence of exploration steps. At each step, two main functions are executed: (1) *Filter* function to determine if an embedding subgraph should be processed. (2) *Process* function which apply some action of the embedding. During each exploration step, subgraphs are explored and passed to the application, which will compute outputs and decide whether the subgraph

[19] http://giraph.apache.org/

should be further extended. This model is preferable in graph mining problems such as: frequent subgraph mining, counting motifs, and finding cliques [82] [100].

f) Graph Traversals Model

It was introduced in the Apache Tinkerpop project[20], a graph computing framework for both graph databases (OLTP) and graph analytic systems (OLAP). Tinkerpop uses the Gremlin [101] graph traversal language to help users model their domain as a graph and analyse that graph. In graph traversal model, the traversers walk through the input graph following the instructions specified in the traversal, traversal are distributed using BSP model. When a vertex receives a traverser, it executes its traversal step, and either generates other traversers to be sent as messages to other vertices, or store halted traverser in the vertex attribute. When all traverser are halted and no more traversal are sent the process terminate and returns the location of the halted traversers[82].

ii. Distributed graph dataflow systems:

General-purpose distributed systems can also support graph processing by providing graph processing libraries on top on them. These libraries enable integration of the graph processing operation in the generation data processing without the need to change the system. This is the common scenario in real-world data processing, where the graph processing could be only one step of a pipeline of data processing operations[102]. In the following we review some of those graph processing libraries:

a) Pegasus[21]

Pegasus (Peta-scale graph mining library)[20] is an open source package and a few years ago became very popular in both academia and industry [103] [22]. It was tested using

[20] http://tinkerpop.apache.org/
[21] http://www.cs.cmu.edu/~pegasus/index.htm

Graphs with billions of nodes and edges, and provides important large-scale graph mining algorithms on Hadoop.

The main idea in Pegasus is built on the basis that many of the algorithms used in graph mining use repeated matrix-vector multiplication in computation. In another word, message exchanges in a graph can be represented by multiplication between adjacency matrices and vectors of the current states of nodes. Pegasus introduced Generalized Iterative Matrix-Vector multiplication (GIM-V), which generalise the three internal operations of general iterative matrix-vector multiplication (multiply, sum, assign).

Let $m_{i,j}$ denote the (i, j)-th element of M. Then the usual matrix-vector multiplication is

$$M \times v = v' \; where \; v_i' = \sum_{j=1}^{n} m_{i,j} \; v_i$$

The algorithm is applied in two stages [20]. Stage1 performs *combine2() function* to combine the columns of the matrix with the rows of the vectors ($m_{i,j} \times v_i$). The output then becomes the input to stage2, which combines the results from stage1 and applies the *combineAll() function* $\sum_{j=1}^{n}$ and assign() functions.

Throughout the process two files are used:

(1) *edge file* which is an immutable file that describes the graph where each record corresponds to a non-zero element in the adjacency matrix M With each record is an edge e:$(v_{src}, v_{dst}, mval)$ where v_{src} connect to v_{dst},

(2) *vector file* which is a mutable file where each record is an edge $(v, vval)$ where each v corresponds to an element in the vector V.

This is implemented in two steps:

- Stage 1: one MapReduce job

- It takes an input which is Matrix $M = \{(v_{src}, (v_{dst}, mval))\}$ and Vector $V = \{(v, vval)\}$ and performs *combine2()* function on columns of matrix (v_{dst} of M) with rows of vector (v of V). The output is a (key, value) pairs vector $V' = \{(v_{src}, \text{combine2}(mval, vval))\}$.

- Stage 2: one MapReduce job is required:
  - It takes the output of Stage1 and combines all partial results using the *combineAll() function* and assigns the new vector values to the old vector.

Stage 1 and 2 run iteratively until convergence is met, resulting in the new vector file.

In Pegasus, GIM-V was used to define different algorithms such as PageRank, Random Walk with Restart, diameter estimation, and connected components [20], [50], [52].

b) GraphX

GraphX is a Spark API for graphs and graph-parallel computation[25], [104], [105] it offers a graph abstraction that is implemented using Spark, which combines both specialized graph system optimizations together with the partitioning, lineage, and effective fault tolerance in distributed dataflow frameworks. GraphX also adopt the GAS model (Gather, Apply, Scatter) to efficiently distribute computations across the cluster[105].

GraphX API gives the user the ability to construct a graph and express it either as a graph or as a collection, which gives the flexibility in applying a wider range of operations and optimisations without data movement or duplication.

It introduces Resilient Distributed Property Graph based on Spark RDD, which is a directed graph that has a pair of collections (RDDs) for vertex and edge with some additional operations and specific optimizations for graph computation (figure 2-12)[104].

```
class Graph [VD, ED] {              //VD, ED are types for vertex and edge
    val vertices: VertexRDD[VD]
    val edges: EdgeRDD[ED]
}
```

Figure 3-1:GraphX Graph Class

In vertex collection (*VertexRDD*), vertices keyed by a *unique* 64-bit long identifier (*VertexID*) with constraint that each *VertexID* occurs only *once* and stored in a reusable hash-map data-structure. In edge collection (EdgeRDD), edge properties keyed by the source and destination *VertexID.* As explained earlier Spark RDDs are immutable, distributed, and fault-tolerant, similarly graphs GraphX. When change is made on a graph a new graph is created from the original one, but the new graph inherits the original graph indices, attributes, and structure (if it was unaffected) and reuse them.

GraphX also introduce a new RDD called triplet view, a logical representation that joins the *source* and *destination* vertex properties with the *edge* properties (figure 2-13).

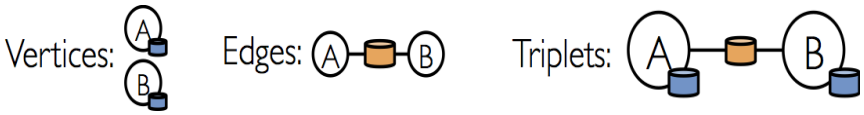


Figure 3-2: Triplet View [22]

As a graph processing framework embedded within Spark the distributed dataflow system, GraphX Includes the operators available in Spark on RDDs (e.g. map, filter, and reduceByKey) in addition to some specialized graph operators[106]:

- ***Property Operators***, in which new graph generated with vertex or edge properties modified but the structure is unaffected (*mapVertices, mapEdges, mapTriplets*)
- **Structural Operators**, in which new graph generated with a modified structure (*reverse, subgraph, mask, groupEdges*)

[22] https://spark.apache.org/docs/latest/graphx-programming-guide.html

- **Join Operators**, in which RDD is joined with the graph(*joinVertices, outerJoinVertices*)
- **Graph Builders**, provides several ways of building a graph(*fromEdges, fromEdgeTuples, edgeListFile*) and to repartitions the graph's edges use(*partitionBy*)
- **Neighbourhood Aggregation**, in which information about adjacent triplets are aggregated (*aggregateMessages, mapReduceTriplets*)
- **Collecting Neighbours**, to collect neighbouring vertices and their attributes at each vertex (*collectNeighborIds, collectNeighbors*)
- **Computing Degree Information** (*inDegrees, outDegrees, degrees*)
- **Caching and Uncaching operators**, for caching graphs in memory or removing it from memory (persist, cache, unpersistVertices)

In addition, Pregel (*pregel*)operator is available based on bulk-synchronous parallel messaging abstraction, which is also used to implement a set of graph algorithms that can be used directly in GraphX for analytics tasks (*pageRank, connectedComponents, triangleCount, stronglyConnectedComponents*)

Optimisation in GraphX

On the top of Spark distributed dataflow framework with GraphX operators, graph specific optimisation techniques are implemented, vertex-cut partitioning approach is used to minimizes communication and storage overhead, however another range of built-in partitioning strategies are included to choose. Because GraphX treats graphs as a joined collection of RDDs and when an operator derived a new graph without modifying the structure (e.g., *subgraph*) or create a new RDD without changing indices of the original graph (e.g., *mapVertices*). Index reuse increases the efficiency by enabling faster joins and reducing memory overhead[105][25].

In term of Triplet view, which is a three join operations between vertices and *edge* properties, to do these join efficiently and improving system performance, some optimisation techniques applied such as: Vertex Mirroring, Multicast Join, Partial Materialization, Incremental View Maintenance[104] **.**

c) Gelly

Gelly is a graph processing library on top of Apache Flink [107], and it is implemented on top of its dataset API [66]. It supports the implementation of different graph processing models, such as vertex-centric, gather-sum-apply, and scatter-gather[84]. In Gelly, graph is represented by a dataset of vertices and a dataset of edges (figure3-3):

```
class Graph [K, VV, EV] {                    //K represents the vertex id type
        DataSet <Vertex <K, VV >> vertices    // VV the vertex value type and
        DataSet <Edge <K, EV>> edges          //EV the edge value type
}
```

Figure 3-3: Gelly Graph Class

A Gelly graph provides operations for creating graphs and performs simple graph *metrics operations* to retrieve graph properties. In addition to *transformation operations*, which return a new, possibly modified graph from the input graph, it also provides *neighbourhood operations*, which allow vertices to perform an aggregation on their direct neighbour vertices.

Gelly provides similar functionality to GraphX and it benefits in its iterative methods from the native efficient delta iteration operators in Apache Flink. However, GraphX offers more optimizations for graph processing such as graph partitioning and more efficient join operations by reducing network traffic between workers through vertex mirroring and multicast joins[76], [102].

3.5 Summary

This chapter provided an overview of the graph concept in modelling relationships in datasets. The concept of graph has been used to understand complex human and natural phenomena through the algorithms developed to solve different kind of problems. Followed by reviewing graph processing systems focusing on big graphs. It is very active research area and its development is moving in different directions. However, the interest of this study is the processing of big graphs in distributed systems. The Key features of big graph processing was briefly discussed. In addition, a description of the two main approaches used in the developing of big graph processing systems was given at high level of abstraction. The first approach is the dedicated distributed graph processing. The second approach is the general-purpose distributed graph processing. Both approaches will be used in this study.

Although the main focus of this study is on the connected component algorithms, it is important to understand the different environments and processing systems where these algorithms were developed and applied. Different systems provide different features with different kind of limitations which might affect the way algorithms are designed.

Chapter 4: Finding Connected Components in Large Graphs

In this thesis, we focus on the algorithms for finding connected components in an undirected graph, which is one of the main concepts that has been studied in Graph Theory[10]. In the following, we defined finding connected components algorithm and its application, then review its models of study, and explain its importance, and finally review available algorithms that were implemented using MapReduce.

4.1 What is finding connected components in graphs?

A component represents a graph (or subgraph) where any two vertices inside that graph are connected via paths, and there is no edge that connects any vertex outside the component. Isolated vertices are considered connected components themselves and a component could include all the vertices in a graph, in which the whole graph will be a single component[6].

4.2 Why it is important to study Connected Components algorithms?

The problem of finding disjoint subgraphs (connected components) has been well studied, as it is an essential pre-processing step to extract knowledge about the graph[10]. It is also a fundamental operation for some graph computations such as pattern recognition, reachability, graph compression, graph partition, and random walk[12]. It also makes an essential first step in some sophisticated graph techniques [14], and is a building block in complex graph analysis such as clustering[13].

4.3 Application of Connected Components algorithms?

Finding connected components has uses on a broad range of applications that include e calculation of betweenness centrality, community detection, image processing[108], complex graph analysis like clustering [22], analysis of Coherent cliques in social media, image segmentation[13][9][109] and many more.

4.4 Models of study for finding Connected Components:

The problem of finding connected components in an undirected graph has been well studied for several years. Different solutions and results have been produced based on the environment settings and graph types in which the problem was investigated. However, our focus here is on large graph processing, and the problem of finding connected components in big graph whose size exceeds the memory capacity of a single machine. Different ways have been used to study the problem of finding connected components in big graph. At a high level, we could group them into three groups:

4.4.1 In Single machine systems

i. Traditional

There are well known effective solutions to find connected components in small to medium size graph, usually using graph traversal algorithms such as Depth First Searches (DFS) and Breadth First Search (BFS). However, efficiency decreases as a graph's size increases, given the limited memory available on single machines. Another memory-based approach is proposed by [110] to increase the speed by exploiting the multi-core architecture available in recent computers. This approach is based on the use of the disjoint-set data structure, where each component is a set and connected components are unified in an algorithm based

on Union-Find algorithm. However, this approach is limited by graph size that cannot exceed available memory capacity[12].

ii. Disk-based Systems

To overcome the problems with finding connected components for large graphs on a single machine with limited memory size, new disk-based systems were used. To some extent, this approach can efficiently manage large graphs by utilising both SSDs and rotational hard desks and by using I/O efficient algorithms.

GraphChi, uses label- propagation to iteratively propagate the node with the minimum label to its neighbours until it reaches all the nodes in the same components. Another work by Kim et al. [111] proposed a DSP-CC algorithm to find connected components for billion-scale graphs in a single PC based on a union-find algorithm[12].

Such systems benefit from the ability to perform graph mining on large graphs and avoid tricky and expensive tasks such as cluster management or high-performance computer configuration. However, they offer limited scalability when processing graphs with hundreds of billions of edges. Moreover, for such big graphs with billions of edges their size will become extremely large and it is not always practical or possible to store them on the hard disks of one single machine[14].

4.4.2 In distributed systems

Many algorithms were developed to implement finding connected components algorithms in parallel using Depth First Searches (DFS), Breadth First Search (BFS), propagation, or contraction. However, they are unable to handle large-scale graphs [112][20].

For example, an optimised parallel BFS is not efficient when there are components with large diameters or a large number of components with small diameters [15].

i. PRAM

The Parallel Random-Access Machine (PRAM) model is a shared-memory abstract machine where processors compute in parallel using a common shared memory. It is commonly the classical model used in to analyse the performance of parallel algorithms.

The classic Shiloach-Vishkin (SV) algorithm is widely used for finding connected components in the PRAM model, requires the PRAM model to handle concurrent reads, and writes to the shared memory (CRCW PRAM). The algorithm will begin with single trees corresponding to each vertex and maintain a forest of trees of connected components. It then iteratively applies pointer chasing operations (change the pointer of a vertex from pointing to its parent to pointing to it grandparent in the tree), or hooking (merging two different trees into one larger tree) at each iteration. The algorithm bounds the number of iterations to O (log n) where each iteration requires O (n+m) processors (n is the number of vertices and m is the number of edges). Many improvements and optimisations have been introduced to the SV algorithm and PARM model such as the recent work by Jain et al.[15]. Generally, this approach assumes computing processors have access to shared memory and can perform concurrent writes. However, this may not always be efficiently implemented in large-scale distributed processing paradigms such as MapReduce [22].

ii. Bulk Synchronous Parallel (BSP)

Implementations for finding connected components can be found in almost all dedicated large graph processing systems which are based on the Bulk Synchronous Parallel (BSP) paradigm[82]. For example, systems such as Pregel, GraphLab, PowerGraph use a vertex-centric programming model (explained earlier in section 3.4.i.a) to implement the algorithm based on a label propagation. The approach benefits from the reduced overheads between iterations compared to MapReduce. This model could be very efficient for finding connected components in a large graph.

iii. MapReduce

MapReduce has been the dominant programming model used for processing large-scale data in recent years, because of its ease of use, fault-tolerance, and scalability. Therefore, much work has been carried out which attempts to migrate many previous algorithms and implement them using the new paradigm[113], this includes algorithms for finding connected components in graphs. However, it is not always an easy or efficient step for large graphs. For example, to find connected components using a breadth-first search[114] will require a number of iteration equal to the sum of the diameters of each connected component which is not acceptable for medium or large graphs. Work in Pegasus[20] and zone[10] reduce the number of iterations to O(d), where d is the diameter of the largest component, but it is not efficiently scalable to large graphs[14]. Moreover, a problem of prohibitive communications load per iteration could occur in some other implementation of connected components algorithm[14]. Later in this thesis most of the work carried out to find the connected components of a large graph using MapReduce will be reviewed.

4.5 Why CC algorithms may perform poorly in practice?

Due to the huge size of many current graphs, tackling the problem of connected components has become a challenge and traditional processing solutions are not feasible especially when the graph size does not fit within the collective memory of a parallel computing cluster. Furthermore, highly scalable parallelized algorithms that can adapt to different types of real-life graphs are very complex to develop, and hard to maintain and have limited portability [115], [116], [117].

Adopting the new MapReduce paradigm is promising, but requires care when designing graph algorithms using MapReduce. This is because:

- Graph algorithms usually require multiple iterations which is an inefficient process in MapReduce because of the I/O overhead during each iteration. Therefore, it is very useful to reduce the number of iterations to the minimum.
- Some algorithms could increase the graph size with newly generated edges, which will increase the communication cost during each iteration. Therefore, minimising the volume of message passing during each iteration could help increase the performance.
- Many real-life graphs have a degree of skew which could lead to an unbalanced workload resulting in few reducers in the cluster taking almost all the workload and slowing the whole process as the other reducers need to wait.

4.6 Why it is important to use MapReduce in Graph Processing?

The most common choice nowadays is to store large-scale data in a distributed format using distributed file systems such as HDFS. This includes large-scale graph data, which makes it very suitable to process using MapReduce. MapReduce might not be ideal for cases like iterative graph algorithms, but the huge popularity of Apache Hadoop make it important to find an adequate implementation for graph algorithms based on MapReduce.

The BSP model could be very efficient in large graph processing. However, some recent works[21][22] show that it is possible to outperform BSP model algorithms for finding connected components in MapReduce. MapReduce can have better latency than BSP in congested cluster situations [14] and when the graph size is more than the collective shared memory of the system[12]. Therefore, there is a need to design efficient graph algorithms

using MapReduce that could give both good performance and better integration with general distributed processing systems.

4.7 Previous algorithms for Finding CC in MapReduce

4.7.1 Using zones to finding connected components

One of the earliest algorithms to find CC in a large graph was introduced by Cohen [10]. His algorithm uses zones to find connected components based on MapReduce. The main concept is by forming "zones" each of which includes a set of vertices that belong to the same component. As the algorithm proceeds, the zones merge to form larger zones.

Throughout the process two files are used:

(1) *edge file* which is an immutable file where each record is an edge e:(v_1,v_2) and v_1 connects to v_2.

(2) *zone file* which is a mutable file where each record is an edge (*v*, *z*) and each vertex *v* is assigned to a so-called zone *z*. Initially the *zone file* is constructed with each vertex assigned to itself as its own zone (*v, v).*

The basic idea is that the algorithm iterates testing to determine if each edge connects two vertices from different zones so that zones will be merged forming larger zones. The algorithm will iterate until no further expansion is possible. Finally, each zone will correspond to a component.

This is implemented in two steps:

- Step 1 requires two MapReduce jobs:
    - MapReduce 1 merges records from the edge and zone files and associates edges with zones, each output record connects an edge to a zone (e, z).

- MapReduce 2 reads all the zones for the same edge Z, to find the minimum zone $z_m \in Z$ such that $z_m \leq z, \forall z \in Z$, for each $z \in Z - \{z_m\}$. The output records then each connect one zone to a better one (z, z_m).

- Step 2 requires only one MapReduce job to update the zone file:
  - MapReduce 3 updates the old zone file by merging the output from Step1 in mapper (z, z_m) with the old zone file records (v, z). The result would be $(z, \{V \cup Z\})$ where V is all the vertices pointing to the old zone, and Z is the new better zones for the old zone (which could be zero). The reducer then finds the best new zone for this vertex: $Z_b = \min\{t \mid t \in Z \cup z\}$ and outputs a record (v, z_b) for each $v \in V$ assigning each vertex to the new zone.

Steps 1 and 2 alternate until the first step produces no records, at which time the result is in the zone file.

The main drawback of this algorithm is that all edges need to be processed at each iteration and each iteration requires execution of three MapReduce jobs. Each MapReduce job could move every graph record across the cluster, which could inefficiently increase the bandwidth used. This algorithm requires $O(d)$ iterations where d is the diameter, and $O(m+n)$ number of messages per iteration. In addition, Cohen [10] suggested a further improvement on MapReduce3 in step2 to create load balancing on the output of map 3 when there are very large zones.

4.7.2 Pegasus HCC

HCC is the proposed algorithm in Pegasus (explained in section 3.4.ii.a) for finding connected Components in large graphs[20], [50], [52]. It uses Pegasus Generalized Iterated

Matrix-Vector (GIM-V) primitive, to apply a generalized of iterative Matrix-Vector multiplication using MapReduce.

The following steps illustrate the execution of the algorithm in GIM-V:

(1) First, initialize the component ID of each vertex a by initiating a component vector $C = \{ (v, cval)\}$, where $cval$ represent the component id for vertex v. Initially $cval = v$.

(2) Second, each node sends its component ID to its neighbours. This is done in Stage1 in a MapReduce job by performing the function:

$$combine2(m_{i,j}, C_j) = m_{i,j} \times C_j$$

(3) Third, in Stage2 in the second MapReduce job, each node updates its component ID by the smallest component ID received. This is achieved by performing the function

$$combineALL_i(x_1, \ldots, x_n) = min\{ x_j \mid j = 1 .. n \}$$

Find for each vertex the minimum value among the current component id and all the received component ids from its neighbours and then update the component id in the component vector C using the function:

$$assign(C_i, C_{new}) = min(C_i, C_{new})$$

(4) Finally, iterate until no change is required.

The main drawback of this algorithm is that each vertex sends its component id to only its direct neighbours and updates each adjacent one hop at each iteration. This means the upper bound of iterations requires a maximum of *d* iterations where *d* is the diameter of the graph, in addition, two MapReduce jobs are required for each iteration, and each MapReduce job requires disk I/O and shuffling which decreases running time.

Both this algorithm, Pegasus HCC and the previous Zones algorithm do not scale well for a graph with a large diameter according to Rastogi[22].

Time Complexity of GIM-V: One iteration of GIM-V takes $O\left(\frac{m+n}{M}\ log\frac{m+n}{M}\right)$ time, where M is the number of machines. Space Complexity: GIM-V requires $O\ (m+n)$ space, and requires $O(d)$ iterations, with $O(m+n)$ number of message per iteration.

4.7.3 Hash-to-Min

Hash-to-Min [22][22] was developed and tested to compare with Pegasus HCC and showed a better performance regarding runtime. The algorithm tries to enhance performance by minimizing the number of iterations and communication per step.

The main idea is to maintain a cluster file where each vertex points to its cluster, and iteratively merge overlapping clusters to compute connected components.

Hash-To-Min is the proposed algorithm for finding connected components:

(1) First, a *Cluster file* is created as a mutable file where each record is an edge (v, C_v) where each v corresponds to a vertex of the graph G, and C_v is a cluster of vertices. Initially, the algorithm assumes that each vertex and its neighbours constitute a connected component. $C_v = \{v\} \cup\ nbrs(v)$

(2) Second, the map stage, where the mapper applies Hash-to-Min function to (v, C_v) where it:

   a. Finds v_{min} which is the smallest vertex in the cluster C_v .

   b. Sends the entire cluster C_v to reduce vertex v_{min} .

   c. Sends $\{v_{min}\}$ to all reducers of all vertices $u \in C_v$.

(3) Third, the reduce stage, where each reducer for a key v aggregates tuples emitted by different mappers ($C_v^{(1)}, \dots, C_v^{(k)}$). The reducer applies the merging function over $C_v^{(i)}$ to compute a new value C_v by taking the union of all vertices received.

(4) Repeat from step (2) until no change is made to any in cluster C_v .

(5) Finally, export connected components C from the final clusters C_v using one MapReduce iteration.

Input: An undirected graph G = (V, E),
Hashing function *h*,
Merging function *m*,
Exporting function *EXPORT*.
Output: A set of connected components $C \subset 2^V$
1. Initialize $C_v = \{v\} \cup nbrs(v)$
2. **repeat**
3. **mapper for node** *v:*
4. Compute *h* (C_v), which is a collection of key-value pairs (u, C_u) for $u \in C_u$.
5. Emit (v_{min}, C_v), and (u, {v_{min}}) for all nodes $u \in C_v$.
6. **reducer for node** *v:*
7. Let $\{C_v^{(1)}, \dots, C_v^{(K)}\}$ denote set of values received from different mappers.
8. Set $C_v \leftarrow m(\{C_v^{(1)}, \dots, C_v^{(K)}\})$.
9. **until** C_v does not change for all v.
10. **return** C = *EXPORT* ($\cup_v \{ C_v \}$)

Figure 4-1 Hash-to-Min Algorithm[22].

The main drawback of this algorithm is that it requires the largest connected components to fit in the memory of a single reducer, which is not usually the case in large-scale graphs. This is especially for graphs with degree skew. In addition, there will be a heavy load on that reducer, which could cause a communication bottleneck and decreases the performance. Some later modifications to this algorithm enhanced performance and scalability by using

secondary sorting in MapReduce and load balancing. However, it is still does not have good load balancing properties[103].

For the worst-case scenario as in path graph, Hash-to-Min can be shown to complete in $O\log(l)$ number of MapReduce iterations with communication $O(\log l\ |m|\ +|n|)$, where l is the size of the largest connected component. However, Rastogi et al[22] claim that in practice the algorithm completes in at most $2\log(d)$ iterations and $3(|m|\ +|n|)$ communications per iteration where d is the diameter of the graph.

4.7.4 CC-MR

A similar approach to Hash-to-Min appeared the same year by Seidl et al [13]. The CC-MR algorithm outperforms Pegasus-HCC and zones Algorithms in terms of the number of iterations, communication costs, and execution runtime.

The basic idea is based on the zones algorithm. However, it improves it by adding additional edges on the graph as a shortcut to reduce the number of iterations needed to converge. In each iteration, edges are added and deleted in such a way that vertices with larger IDs are connected to the vertices with a smaller ID. Each component has one of two states, operating either in locally maximal state where no further steps need to be performed or in merge state where it should be merged with another sub-component. CC-MR is the proposed algorithm for finding connected Components[13]:

The input file contains the graph itself, where each record represents an edge $(v_{\text{source}}, v_{\text{dest}})$ in a graph $G = (V, E)$ where V is a set of vertices and E is a set of edges.

Initially, a MapReduce job is required to create a representation of the graph based on the adjacency list of all vertices $(v,\ adj(v))$, where $adj(v)$ is a list of neighbours of vertex v.

- In the *map stage*, the mapper used is called *identity mapper*, which only forwards the data to the reducer without performing any changes.
- In the *reduce stage*, each reducer for a key v aggregates tuples emitted by different mappers($u_1, u_2, .., u_k$)which represent $adj(v)$. Here Hadoop secondary sorting is used, which means the reducer will receive values in order (first value u_1 is the minimum in $adj(v)$).

  Check for vertex v and its adjacent neighbour vertices in $adj(v)$ if v has the smallest id or not ($v > u_{first}$):

  a) If no (locallyMaxState), assign all $u \in adj(v)$ to v by sending (v, u).

  b) Else (mergeState), there is a vertex $u \in adj(v)$ in which $u < v$, in this case it is u_{first}, assign v and $adj(v)$ to u_{first} by sending (v, u_{first}) and sending (u_{first}, u) and (u, u_{first}). This will merge the components of v with the components of u.

The process will iterate until the merge State situation does not occur anymore. The graph will transform into star-like subgraphs, where each one represents a component and the centre vertex is the component ID.

```
1.  newIterationNeeded = false                    // global variable
2.  void reduce (Int v_source , Iterator<Int>  values)
3.      isLocMaxState = false
4.      v_first = values.next();                  // take first element
5.      if ( v_source < v_first )
6.          isLocMaxState = true
7.          emit(v_source, v_first)
8.  v_dest-old = v_first
9.  while ( values.hasNext() {
10.     v_dest = values.next()
11.     if ( v_dest == vdestold ) continue        // remove duplicates
12.     if ( isLocMaxState )                      // locMaxCase
13.         emit( v_source, v_dest)               // only fwd. edge
14.     else                                  // cases stdMergeCase, optimizedMergeCase
15.         emit( v_first, v_dest)                // fwd. edge and
16.         emit( v_dest, v_first)                // backwd. edge
17.         newIterationNeeded = true
18.     v_dest-old = v_dest
19. }
20. // stdMergeCase
21. if ( v_source < v_dest  &&  !isLocMaxState )
22.     emit( v_source, v_first)                  // backwd. Edge
```

Figure 4-2 Reducer of the CC-MR algorithm[13].

Similar to Hash-to-Min, the main drawbacks for this algorithm are that it requires the largest component in the graph to fit in the memory of a single reducer. In, a very large component will put a heavy load on one reducer. When this happens, we have skew degree graph with a very large component. This situation is also called 'the curse of the last reducer', when a connected component is very large and all the vertices of this component need to be sent to and processed by the same reducer. This may then cause computation problems and place a large communication load on one reducer. CC-MR also lacks any analytical guarantees.

Both Hash-to-Min and CC-MR algorithms address the "curse of last reducer" problem and provide solutions to carry out load balancing:

- in Hash-to-Min, for a vertex (v, C_v) when its cluster C_v gets larger than a specified threshold. In the cluster C_v, for all the vertices u, $where\ u \in C_v\ and\ u \leq v$ send u to v_{min} reducer and send v_{min} to u reducer. For all vertices u, $where\ u \in C_v\ and\ u > v$, send u to v reducer, and send $\{ v \}$ to u reducer. To ensure that v_{min} does not receive more than the threshold.
- in CC-MR the solution for the problem is by augmenting hash values in the map phase to vertex which has adjacent list larger than a specified threshold which is then sent to different reducers.

4.7.5 CCF

Later, Kardes & Agrawal [9] presented their Connected Component Finder (CCF)algorithm that they had been using regularly for two years on massive graphs.. The main idea is similar to Hash-to-Min but the algorithm will run iteratively using a chain of two MapReduce jobs in each iteration called CCF-Iterate, and CCF-Dedup.

Initially, the input file contains the graph itself, where each record represents an edge (v_1, v_2) where v_1 connects to v_2 in a graph $G = (V, E)$ and V is a set of vertices and E is a set of edges.

- *CCF-Iterate stage*:
    - The mapper will send values so that an adjacency list will be generated in reducer like the initial step in CC-MR.
    - The reducer will receive the data $(u_1, u_2, .., u_n)$ sorted as the MapReduce secondary sort approach is used It will take the first value which should be the minimum (as the values are sorted $(min = u_{first})$) and compare it with the vertex id v :

      If $(v > min)$ then v is not the minimum:
        - Send (v, min).
        - Send (u_i, min) for each node u_i in the adjacency list.
        - Increase a global counter which is used to indicate if a new component is found.
- *CCF-Dedup stage* which only reproduces CCF-Iterate output with no duplication to increase efficiency in term of speed and I/O overhead.

Iterate until the counter is zero, which means that no new component has been found.

According to the author CCF outperformed PEGASUS in terms of total runtime and was slightly behind CC-MR, unless the diameter is very large. It finishes with $d + 1$ iterations in a worst-case scenario where d is the diameter of the graph.

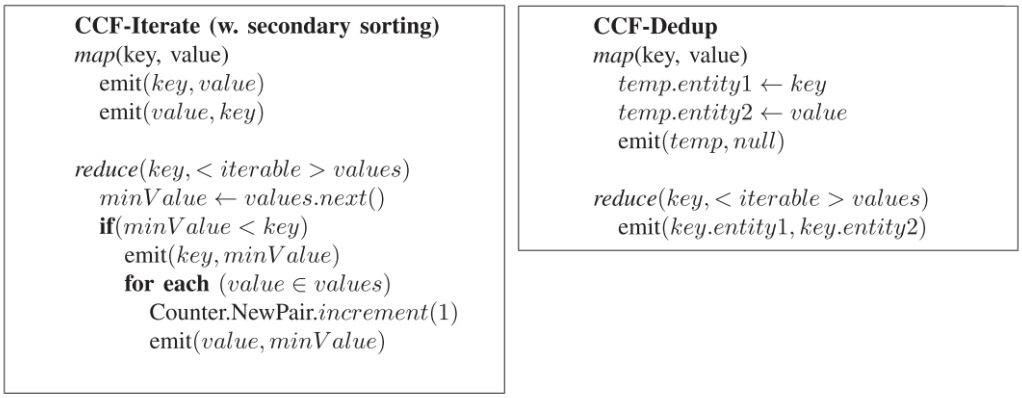


Figure 4-3 CCF Algorithm[9].

4.7.6 MemoryCC

In MapReduce inputs are in the form (key, value) pairs. Map and reduce jobs will process each input record one by one. Lin and Schatz [118] proposed a set of enhanced design patterns that can be used to accelerate a large class of graph algorithms. One of the "rules" mentioned is building a hash table that contains the graph structure for the graph partition processed in the mapper, and perform messages passing from source to destination locally between vertices that are included in the Hashtable. Therefore, reducing the data traffic on the network will increase the speed of the algorithm as network traffic dominates the execution time of MapReduce algorithms. This concept was adopted by Varamesh & Akbari [119] in their algorithm for finding connected components. Initially each map job loads all its input records to a hash table that could be accessed during the map job lifetime.

MemoryCC is the proposed algorithm for finding connected components in large graph:

Firstly, for a graph $G = (V, E)$ a mutable *Graph file* is generated in a similar way to the initiation step in the previous algorithm. Each record represents a vertex and its component id and adjacency list $(v, (comID, adj(v)))$ where $compID$ is the component ID for vertex v, and $adj(v)$ is its list of neighbours. Initially, it assumes that each vertex is a connected component itself and set $comID = v$.

- In the Map stage, the mapper takes the input in a <key, value> format and then loads the entire partition assigned to it and generates a HashTable that has the graph structure of this partition. By doing so the mapper will have access to the structure data of its part of the graph during its lifetime by using the hash table generated previously:
    a. In the map stage, before emitting any message, it updates the component ID with the minimum for each vertex that has internal neighbours in the hash table and this process will repeat until no internal updates occur. This will reduce the number of Component ID updates that need to be sent to the reducer.
    b. If a connected component is split over more than one mapper, each mapper will process the partial connected component internally and define the smallest Component ID and emit it, which will to converge faster and reduce the number of iterations.
- In the reduce stage, each reducer for a key v aggregates tuples emitted by different mappers $(C_v^{(1)}, \dots, C_v^{(k)})$ in sorted order using secondary sorting in Hadoop MapReduce and takes the smallest one as the new Component ID.

Repeat the MapReduce iteration until no change is made to any in cluster $ComID$.

According to the author, the MemoryCC algorithm communication complexity is O(n) where n is the number of vertices and runtime should be in direct relation to the number of

vertices. The algorithm helps in reducing both the intermediate data communications and number of iterations and in practice it performs up to ten times faster than PEGASUS and CC-MR However further analysis of time and space complexity is needed to prove this[119].

```
1.  Map
2.  Hashmap subgraph<key,value>
3.   Input <Key, Value> : <(node n, Comp_ID_n), adjacency list of n>
4.      subgraph.put(node n, <Comp_ID_n , adjacency list of n>)
5.      while (any component ID updates) do
6.          for each node n in subgraph do
7.              for each node i which is neighbor of n do
8.                  if i is in subgraph & Comp_ID_i is smaller than Comp_ID_n do
9.                      replace Comp_ID_i with Comp_ID_n
10. for each node i in subgraph do
11.      emit <i, Comp_ID_i >
12.      emit <i, adjacency list of i >
13. for each node i not in subgraph
14.                      & has at least a neighbor in subgraph do
15.      emit <i, smallest Comp_ID of i's neighbours in subgraph >
16. Reduce
17. Input<Key, Value> = <node n, received IDs and adjacency list of n >
18.     component_ID_n = smallest id received
19.     emit < (n, component_ID_n) , adjacency list of n >
```

Figure 4-4 MemoryCC Algorithm[119].

4.7.7 CC-MR-mem

Kolb et al.[120] proposed CC-MR-mem which is an algorithm for finding connected Components in large graphs. It is an optimized version of MR_CC[13]. One of the major

improvements achieved by CC-MR-Mem is memory-based connection of subgraphs in the map phase and is similar to the approach adopted in MemoryCC by Varamesh & Akbari [119]. In addition has the capability to identify stable components that do not grow more to avoid increased processing times and this will enhance performance by reducing the amount of intermediate data communicated and the number of iterations.

Initially, the input file contains the graph itself, where each record represents an edge $(v_{\text{source}}, v_{\text{dest}})$, where v_{source} connects to v_{dest} in a graph $G = (V, E)$, V is a set of vertices and E is a set of edges.

- Unlike the map job in MR_CC, the map stage inputs records of the entire partition assigned to the mapper which will be buffered in memory and where a HashMap Table will be generated. Accordingly, the map output will be generated in the following two steps:
    a. *Generate Hash Table*: The hash table is constructed with each vertex mapped to a set of vertices connected to it, $HashMap\ (v, ComponentSet)$. As new edges are added, component sets for their vertices are created, updated, or merged with each other.
    b. *Generate map Output*: Depending on the hash table the mapper will defined the minimal vertex (smallest ID) for each component set, and accordingly generate an output that connects each vertex to the minimal vertex and vice versa. This reduces the amount of processing required in the reduce job This will also reduce the amount of intermediate data communicated and the number of iterations at an extra cost of using more memory and processing in the map phase.

The *reduce stage* is the same as the reduce job in MR_CC, where each reducer for a key v aggregates tuples emitted by different mappers $(u_1, u_2, .., u_k)$which represent $adj(v)$. Here

Hadoop secondary sorting is used, which means the reducer will receive values in order (first value u_1 is the minimum in $adj(v)$).

Check for vertex v and its adjacent neighbour vertices in $adj(v)$ if v has the smallest id or not ($v > u_{first}$):

c) If yes (locallyMaxState), assign all $u \in adj(v)$ to v by sending (v, u).

d) Else (mergeState), there is a vertex $u \in adj(v)$ in which $u < v$, in this case it is u_{first}, unlink CC-MR here we only need to send $(u_{first}, u), u \in adj(v)$.

Repeat the MapReduce iteration until the mergeState situation does not occur anymore. The graph will transform to star-like subgraphs where each one represents a component and the centre vertex is the component id.

Performance is also enhanced by identifying stable components that need no more processing. Stable components will be augmented by a stable flag so that they will be separated in the following iteration when they will be written to a different file and will not be read by mapper in the later iterations. This enhancement is achieved with virtually no additional overhead in terms of data volume and memory requirements.

A further modification tested by the author was to consider the vertex degree instead of the minimum vertex id in defining the component id. However, this approach required more iterations to identify the degree for each vertex, and increased the data transferred between the mapper and reducer. This was in addition to the overhead of reading each vertex degree from the distributed cache during this first iteration.

Algorithm 2: CC-MR-Mem (map phase)

```
1  map_configure(JobConf job)
2      max ← job.getBufferSize();
3      components ← new HashMap<Vertex,MinSet>(max);

4  map(Vertex u, Vertex v)
5      if components.size() ≥ max then
6          generateOutput();
7      comp1 ← components.get(u);
8      comp2 ← components.get(v);
9      if (comp1 ≠null) ∧ (comp2 ≠null) then
10         if comp1 ≠comp2 then                          // Merge
11             if comp1.size() ≥ comp2.size() then
12                 comp1.addAll(comp2);
13                 foreach Vertex v ∈ comp2 do
14                     components.put(v, comp1);
15             else
16                 comp2.addAll(comp1);
17                 foreach Vertex v ∈ comp1 do
18                     components.put(v, comp2);
19     else if comp1 ≠null then                        // Add Vertex
20         comp1.add(v);
21         components.put(v, comp1);
22     else if comp2 ≠null then                        // Add Vertex
23         comp2.add(u);
24         components.put(u, comp2);
25     else                                            // New component
26         MinSet component= new MinSet(u, v);
27         components.put(u, component);
28         components.put(v, component);

29 map_close()
30     generateOutput();

31 generateOutput()
32     foreach component ∈ components.values() do
33         if ¬component.isMarkedAsProcessed() then
34             component.markAsProcessed();
35             min ← component.min;
36             foreach Vertex v ∈ component do
37                 if v≠min then
38                     output(min.v, v);      // Forward edge
39                     output(v.min, min);  // Backward edge
40     components.clear();
```

Figure 4-5 : CC-MR-mem Algorithm (Map Phase) [120].

4.7.8 Two-Phase & ALT-OPT

Kiveris [14] introduced two MapReduce CC algorithms, which can easily scale to large graph with hundreds of billions of edges. These algorithms also outperform Hash-to-Min algorithm by an order of magnitude. Both algorithms make use of the small-star and large-star operations, where they perform both operations until convergence is reached.

- Small-star: for edges to neighbours with smaller or equal ids, replace each edge with an edge to the minimum vertex.

  $N = \{ u \in \Gamma(v) , \forall\, l_u \leq l_v \}$, replace the edge (u , v) with $(u, m(v))$
- Large-star: for edges to neighbours with greater ids, replace each edge with an edge to the minimum vertex.

  $N = \{ u \in \Gamma(v) , \forall\, l_u > l_v \}$, replace the edge (u , v) with $(u, m(v))$

The algorithms will transform the graph into a collection of star graphs where each represents a connected component. They will proceed in a way that guarantees the number of edges never increases, instead it does decrease. Both operations can be easily implemented in MapReduce:

Small Star Operation

Map $< v, u >$

  If $(l_u \leq l_v)$ then:

    Emit (v , u)

  Else:

Emit (u, v)

Reduce $< v, N \subseteq \Gamma(v) >$

  Let $m =$ arg $min_{u \in N \cup \{v\}} l_u$

  Emit $(u, m) , for\ all\ u \in N$

Large Star Operation

Map $< v, u >$

  Emit (v , u)

  Emit (u , v)

Reduce $< v, \Gamma(v) >$

  Let m = arg $\min_{u \in \Gamma^{++}(v)} l_u$

  Emit $(u, m) , for\ all\ u , \forall\, l_u > l_v$

Figure 4-6 Large Start and Small Star operations[14]

The difference between the Two-Phase and the alternating algorithms is the order in which they run large-star and small-star operations:

- In the Alternating algorithm, the large-star and small-star operations run sequentially until convergence.
- In the Two-Phase algorithm in each iteration, large-star is repeated until convergence which then performs the one small-star operation.

Further optimisations have been proposed for both algorithms:

- Similar to CC-MR-mem and MemoryCC this algorithm uses a distributed hash table (DHT) to improve the runtime by reducing the total number of iterations of MapReduce. For the large-star in the Two-Phase algorithm, the DHT creates a table for mapping each vertex to its smallest neighbour, with vertices initially pointing to themselves. Then each vertex repeatedly updates its smallest neighbour in the DHT until convergence. Then only one MapReduce iteration is required to finish. The algorithm will finish in $O(logn\ loglogn)$ MapReduce iterations using a DHT of size $O\left(\frac{n}{log\ n}\right)$.

- The other optimisation is to solve the problem of "the curse of the last reducer" in skew degree graphs and carry out load balancing. The proposed optimised algorithm is called Optimized Alternating (Opt-Alt) It ensures that the reducer for a vertex v will not take all neighbours of v when their number exceeds the threshold defined τ. When the number of neighbours of a vertex exceeds τ, create τ copies of that vertex and attach to the vertex label of each copy an infinitesimally small number and connect the main vertex to each of the newly created copies and distribute the neighbours to the copies created.

4.7.9 Cracker

Lulli [115], [116] introduce the Cracker algorithm, an efficient iterative MapReduce-Like algorithm to find connected component in large graph. Cracker build a spanning tree of connected components in the graph by adding nodes to the tree with each node has its component identifier as it root in the tree. It also provides node-pruning mechanism, which effectively reduce the number of active vertices after each iteration; each node added to the spanning tree is discarded from the computation in the following iterations.

It is organized in two main phases, shown in Figure 4.7:

1. Seeds identification phase, where seed nodes for each component are identified, during this process at each iteration non-seed node will be added to the spanning tree and excluded from the processing graph. This is achieved in two steps:
    a. Min Selection Step: where nodes that are guaranteed not to be a seed node for any component is identified.
    b. Pruning Step: where identified nodes from the previous step are excluded from computation and added to the spanning tree

    This process will iterate the two steps until only the seed nodes for each component are left, where they will be added to spanning tree as the main root node for the component.
2. Seeds Propagation, in which the all seed nodes will propagate their ids to their child nodes until all nodes belonging to the same component will share the same root node id as their component identifier. The result will be a spanning tree of connected components with the root node for each tree as the component identifier.

The algorithm will finish in $O(logn)$ iterations with number of message $O\left(\frac{n\,m}{log\ n}\right)$ per iteration. Thus, the Cracker algorithm outperforms its competitor algorithms, such as

Hash-to-Min, PEGASUS, CC-MR, CCF with best competitor being 9% to 75% slower, also it achieves the least message volume among all its competitors.

Input: an undirected graph G = (V,E)
Output: a graph where every vertex is labelled with the seed of its CC
1. u.Active = True $\forall u \in G$
2. T $\leftarrow$(V, $\emptyset$)
3. t$\leftarrow$1
4. $G^t \leftarrow G$
5. **repeat**
6. $H^t \leftarrow$ Min_Selection_Step(u) $\forall u \in G^t$
7. $G^{t+1} \leftarrow$ Pruning_Step(u,T) $\forall u \in H^t$
8. $t \leftarrow t+1$
9. **until** $G^t = \emptyset$
10. $G^* \leftarrow$ Seed_Propagation(T)
11. **return** G^*

Figure 4-7: The Cracker Algorithm[115]

4.8 Summary

Previously mentioned algorithms are presented in table 4-1 with some information about the complexity analysis and features used in each one such as using Distributed Hash-Table (DHT), if load balancing is considered in the algorithm design, how is the component identifier is selected, and if the algorithm provide vertex pruning. In addition to, what processing system has been used for testing the algorithm. The table also reports the other algorithms used in the performance comparison for each algorithm. Different criteria were used for measuring the performance of each algorithm, mainly the runtime, and the number of iteration, and the intermediate data has been reported for most algorithms.

The Cracker algorithm is the only algorithm that fully use the vertex pruning, where non-active nodes are removed from the processing graph. Thus, it tracks how the size of the graph changes after each iteration by reporting the number of active nodes. Other algorithms also

use load balancing to avoid the problem of 'the curse of the last reducer', when a connected component is very large and all the vertices of this component need to be sent to and processed by the same reducer. Therefore, the computation assigned for the reducer of high degree node are distributed to copies of that nodes across the cluster as presented in figure 4-8.

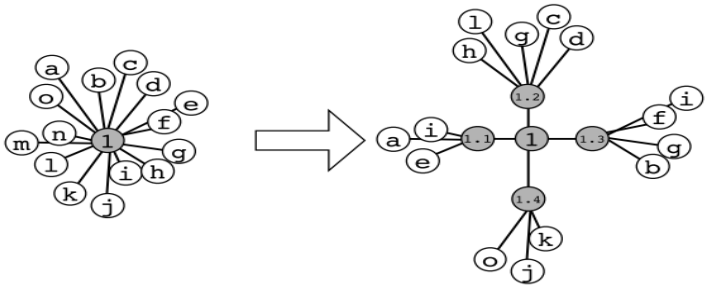


Figure 4-8: Load Balancing[14]

The component identifier for all the algorithms are selected based on the lexical ordering of the node id values. Although, Kolb et.al [120] mentioned using the degree of node for selecting the component identifier, he did not adopt this approach as it requires pre-computing of the degree which would add computation overhead.

Different environment setups were reported in the experimental evaluation for each algorithm. Hadoop processing system is the mainly processing system used except in the Cracker algorithm which used more recent and advance processing system such as Spark. However, there is a variety in the environment setup even when Hadoop is used, different number of the worker nodes and different nodes' specification were used. In addition, different type and size of datasets reported for experiment. Thus, there is a need used common benchmarked datasets and experiments' setups to efficiently compare and evaluate the performance of each algorithm.

Algorithm Name	The algorithms compared to	Round Complexity	Communication Complexity	Measures	DHT	Load Balancing	Component Identifier	Vertex pruning	Experiment environment
Zones	N/A	$O(d)^3$	$O(n + m)^3$	N/A	No	No	ID	No	Hadoop Cluster
Pegasus HCC	N/A	$O(d)^3$	$O(n + m)^3$	- Runtime	No	No	ID	No	Hadoop Cluster 9 nodes
Hash-to-Min	- HCC	$O(\log(d))^4$	$2 (n + m)$	- # of iterations - Intermediate data - Runtime	No	Limited	ID	No	Hadoop Cluster
CC-MR	- HCC - zones	N/A	N/A	- Runtime - # of iterations - # of edges remaining	No	Yes	ID	No	Hadoop Cluster 14 nodes
CCF	- HCC - CC-MR	O(d)	N/A	- Runtime - # of iterations	No	No	ID	Limited	Hadoop Cluster 80 nodes
MemoryCC	- HCC - CC-MR - Hash-to-Min	N/A	O(n)	- Intermediate data - # of iterations - Runtime	Yes	No	ID	No	Hadoop cluster 8 nodes
CC-MR-mem	- CC-MR	N/A	N/A	- # of iterations - Runtime - Intermediate data - overall data volume	No	Yes	ID/ VD	No	Hadoop Cluster (on Amazon EC2) + 20 worker nodes + 40 worker nodes - 100 worker nodes
Alternating Optimized (Alt-Opt)	- Hash-to-Min	$O(\log n)^5$	O(m)	- # of iterations - Runtime- - # of edges remaining	Yes	Yes	ID	No	N/A
Cracker	- Alt-Opt - CCF	$O (\log n)^2$	$O (\frac{n\,m}{\log n})$	- # of edges remaining - Runtime - Active nodes	No	No	ID	Yes	Spark Cluster 5 nodes

Table 4-1: Finding Connected Component Algorithms using MapReduce (n is the number of nodes, m is the number of edges, d is the diameter)

The Cracker algorithm outperforms all its competitors in practice, and theoretically it could achieve the best results. Thus, this algorithm was chosen to be studied in more details. In the next chapter, the Cracker algorithm is selected as the foundation used to implement our proposed enhancement and extend optimisations. In addition, we mainly compare the performance of our proposed approach to the performance of the original Cracker algorithm.

Chapter 5: Proposed Algorithm

5.1 Introduction

All the algorithms previously described in Chapter 4, are design based on the MapReduce programming model. However, current large distributed systems provide more flexibility and further features beyond the MapReduce paradigm, such as the ability to cache and partition the data. In addition, all those algorithms choose the component identifier based only on the lexical ordering of the node id values, ignoring the existing structure of the graph.

Hence, to enhance and optimise the performance of the connected components algorithms, we use two types of properties:

- Graph Structure properties: where we consider the graphs structure properties in choosing the best candidate as component identifier, instead of blindly choosing it based on its lexical ordering.
- Processing System properties: where we consider features and functionality provided by new processing systems to improve the design of the MapReduce implementation of the algorithm in order to increase efficiency or improve the performance of the algorithm.

Lin and Schatzand [118] proposed a set of enhanced design patterns that can be used to improve the performance of a large class of graph algorithms based on message passing. In the process of design and implementation of our new algorithms, we try to follow some of those patterns such as:

- **Role(1)**: Using the Range Partitioning technique to partition the graph into multiple blocks and allowing different mappers to execute in parallel on stored portions of the graph.
- **Role(2):** Building a hash table that contains the graph structure for the graph partition processed in the mapper, and passing messages from source to destination locally between nodes that are included in the Hashtable. Thereby, reducing the data traffic on the network, which in turn increases the speed of a MapReduce algorithm, as the network traffic dominates the execution time.
- **Role(3):** Ensuring the mapper outputs the node structure (< Node ID, {Adjacency_List} >) to the reduce phase to perform multiple iterations.

In the approach we introduce, we will consider the properties from the graphs structure and from the processing system, whilst using some of the MapReduce best practices in the design and implementation of our algorithm for finding connected components in a large-scale graph using GraphX in Spark.

5.2 Proposed Improvements

Finding connected components in large graphs requires iterative processing, however iterative processing is not directly supported in MapReduce. Our aim is to enhance the performance of finding connected component algorithm for undirected graphs in big data processing systems using MapReduce, this could be achieved by addressing the following questions:

- How to reduce the number of iterations while minimising the communication load in the shuffling phase between iterations.

- How to increase the efficiency of the algorithm in modern processing system using the new features provided.

In the Objectives of this study, we follow best practices used in designing MapReduce algorithms and apply it on the algorithm for finding connected components in a graph. Furthermore, in the design of the algorithm, we consider both the properties of the graph structure, and the advanced features supported in current large distributed graph processing systems

We approach our objectives from two angles:

- First, adopted a new approach to enhance the performance of CC algorithms in general. In our case, instead of choosing the component identifier for each connected component based only on the lexical ordering of the node id values, we integrated graph structure degree property and use it in chosen the component identifier.
- Second, we reviewed current algorithms and choose the most recent one that outperforms other algorithms. Then using the properties provided by the processing system, we apply few modifications that could boost the performance of the algorithm. In our case, we based our modifications on the concept of moving the computation process toward where the data is stored could help enhance the performance. This is essentially the concept behind MapReduce also, however, we could benefit here from systems like Spark that provides extra features by caching the data in memory and controlling the partitioning process across the cluster. The algorithm we choose to introduce our improvements is the Cracker algorithm[115], [116], it was selected based on its performance, as it provides the best performance among its competitors (all the algorithms described in chapter 4), in addition it has the feature of graph contraction, where the graph size shrink after each iteration.

With respect to our objective, we design and implement improvements for the Cracker algorithm using Spark, a general-purpose data processing system that provides support for iterative processing. In addition, we use its graph-processing library GraphX, which provides a distributed graph representation with many optimisations implemented that can significantly enhance processing performance. Our work is inspired by the Cracker algorithm, in which a graph is iteratively transformed into a set of trees, each representing a connected component. During each iteration, the graph size is reduced by identifying nodes that do not have impact on other connected components. These nodes are excluded from computation in the next iteration and are added to the trees. The result will be a spanning tree of connected components with the root node for each tree is its component identifier.

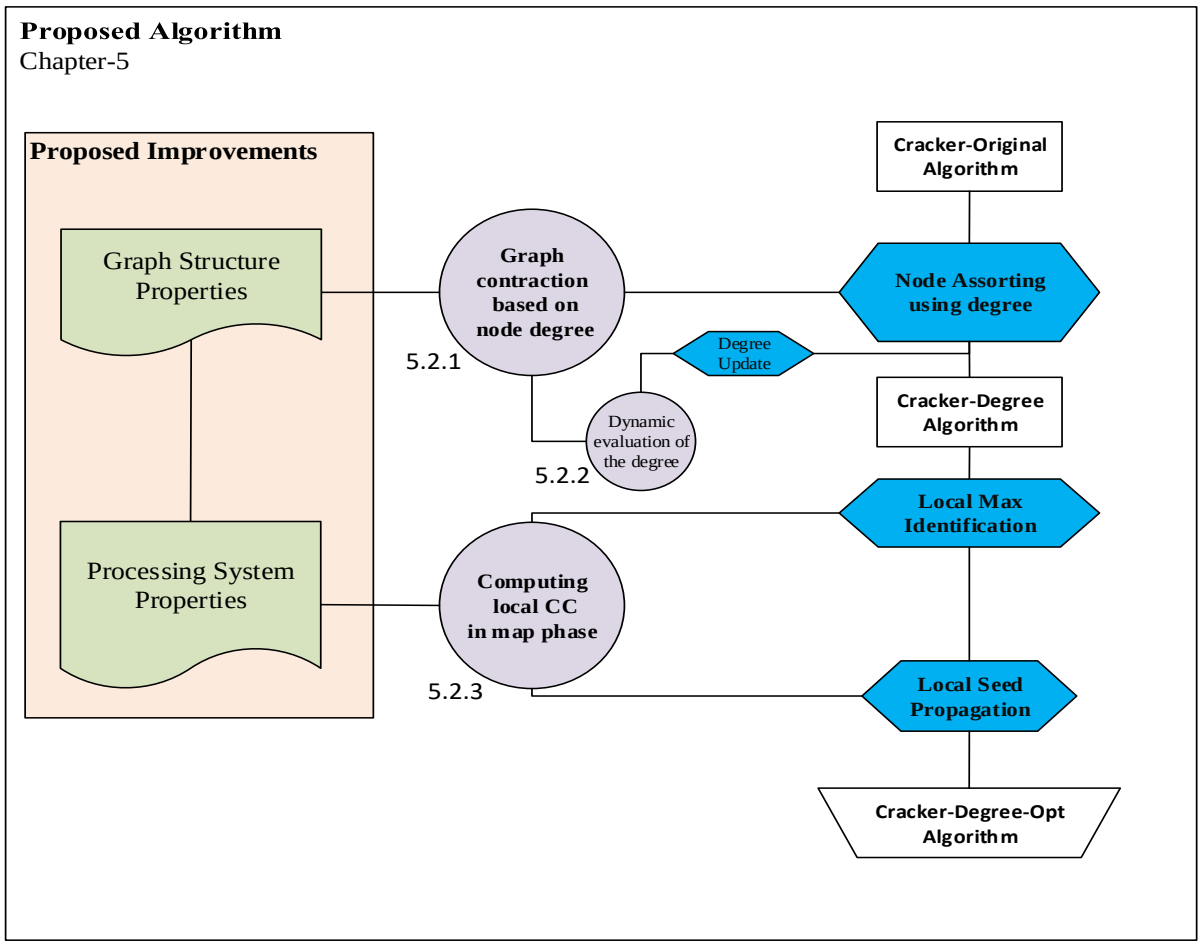


Figure 5-1: Proposed improvements diagram

The key improvements proposed in this research are presented in figure 5-1 , in which base on the graph structure properties and the processing system properties we introduce our improvements and apply them on the original Cracker algorithm, these can be listed as follows:

5.2.1 Graph contraction based on node degree:

Graph contraction is the process in which a node id is selected as a component identifier, then all the other nodes belonging to the same component are contracted into it[121]. In all the algorithms previously reviewed in Chapter 4, the graph contracts to the nodes with the smallest id, based on their lexical ordering, ignoring the existing structure of the graph. However, real-world graphs are usually scale-free graphs and follow the power-law degree distribution, whilst the majority of nodes have a small number of edges only a few have a large number of edges. Existing algorithms use label propagation to propagate the node with the smallest id regardless of any other properties of the node such as degree. This could eventually cause additional iterations by not choosing the node with the highest degree, such nodes help the algorithm to propagate and converge faster.

In the approach we present, we attached the degree to each node and based on that information we choose the component id identifier. When the two nodes have the same degree, we compare the node ids and choose the node with largest id as the component identifier. Therefore, we introduce “node assorting step” to the Cracker algorithm, as shown in figure 5-1, which will be further explained in details later in this chapter.

5.2.2 Dynamic evaluation of the degree in the graph:

Usually, graph contraction based on the smallest id is a straightforward approach, as the node id never changes even if the structure of the graph changes. Change only happens to the node label that indicates its component id or to the adjacency list of the node. However, when using contraction based on degree, the degree of each node might change. This is because, after each iteration the graph structure changes usually by updating the adjacency list for the node. Therefore, updating nodes degree after each iteration could help the algorithm to converge faster following the previous suggested improvement.

5.2.3 Computing local CC in the map phase

In the proposed sets of enhanced design patterns for MapReduce and based on the Role(2)[103](section 5.1). We could reduce the data traffic and increase the speed of a MapReduce algorithm, by building a distributed hash table (DHT) that contains the graph structure for the graph partition processed in the mapper and performing operations locally before emitting messages between nodes in the cluster. This approach was implemented and followed in different algorithms to improve processing time, as in [88], [104], [105]. The main idea is to take nodes in each partition of data and load them with their components' ids in a hash table data structure, and then identifying the local components in each partition. In the hash table, each node is mapped to the components it belongs to. The approach then repeatedly scans and updates the component id for each node until no further updates are needed. At this stage, some components would merge locally, which comes at some cost as it requires more memory and processing. It also places additional overheads on the system during the map phase. However, this step would affect the performance of the algorithm in different ways. For example, the number of emitted messages would be reduced and as a

result less network traffic. Likewise, a number of computations carried out in the reduce phase would also reduce and fewer iteration would be required. With reference to the objectives of this work, we implement this type of improvement on the Cracker algorithm, thus, we introduce two further steps in the algorithm, which are, (a) local max identification step, and (b) local seed propagation step, as shown in figure 5-1 , which will be explained in detail later in this chapter.

Throughout this thesis, the term ‘Cracker-Degree’ is used to refer to the Cracker algorithm after applying nodes’ degree approach in finding connected components, and the term ‘Cracker-Degree-Opt’ is used to refer to the algorithm after apply the improvements based on the local CC computation in the map phase.

5.3 Preliminaries:

Let G = (V, E) be an undirected graph which consists of a set of nodes V (Vertices) uniquely identified by values in ℤ. Nodes that are connected to each other by links called edges E. Let n = |V| be the number of nodes and m = |E| is the number of edges. For a node v, we denote by $\Gamma(v) = \{u \mid (v, u) \in E\}$ the neighbours of v and $\Gamma^{+}(v) = \Gamma(v) \cup \{v\}$ denotes a set that contains the node v itself and its neighbour nodes.

5.4 The Framework Model:

We use features in GraphX to develop and optimised the algorithm for finding connected components in a large graph. Using GraphX helps to achieve a significant impact on performance by providing features such as indices reuse, in memory processing, and controlling partitioning strategies. To better understand the approach we follow, we put the proposed approach into a simple framework pipeline model, shown in figure 5-2.

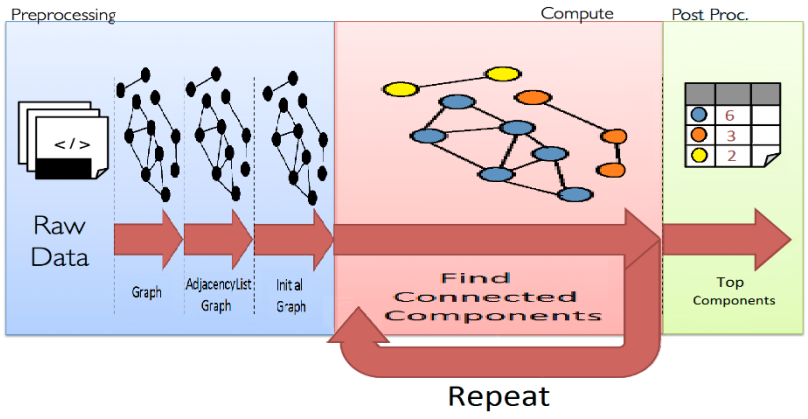


Figure 5-2:Framework Pipeline Model

The framework uses features and operations available in GraphX to enhance data flow in the algorithm. To have a better understanding the data processing is divided into three main stages: (i) *Pre-processing stage* takes input data in its raw format and prepare it with the format required for processing. (ii) *Computing stage*, where we apply the improvements we

introduced to the algorithm. (iii) *Post-processing stage*, where we present the results for evaluation.

In the following, we expand the pipeline framework model and give a more detailed description about each data processing stages, as illustrated in figure 5-3.

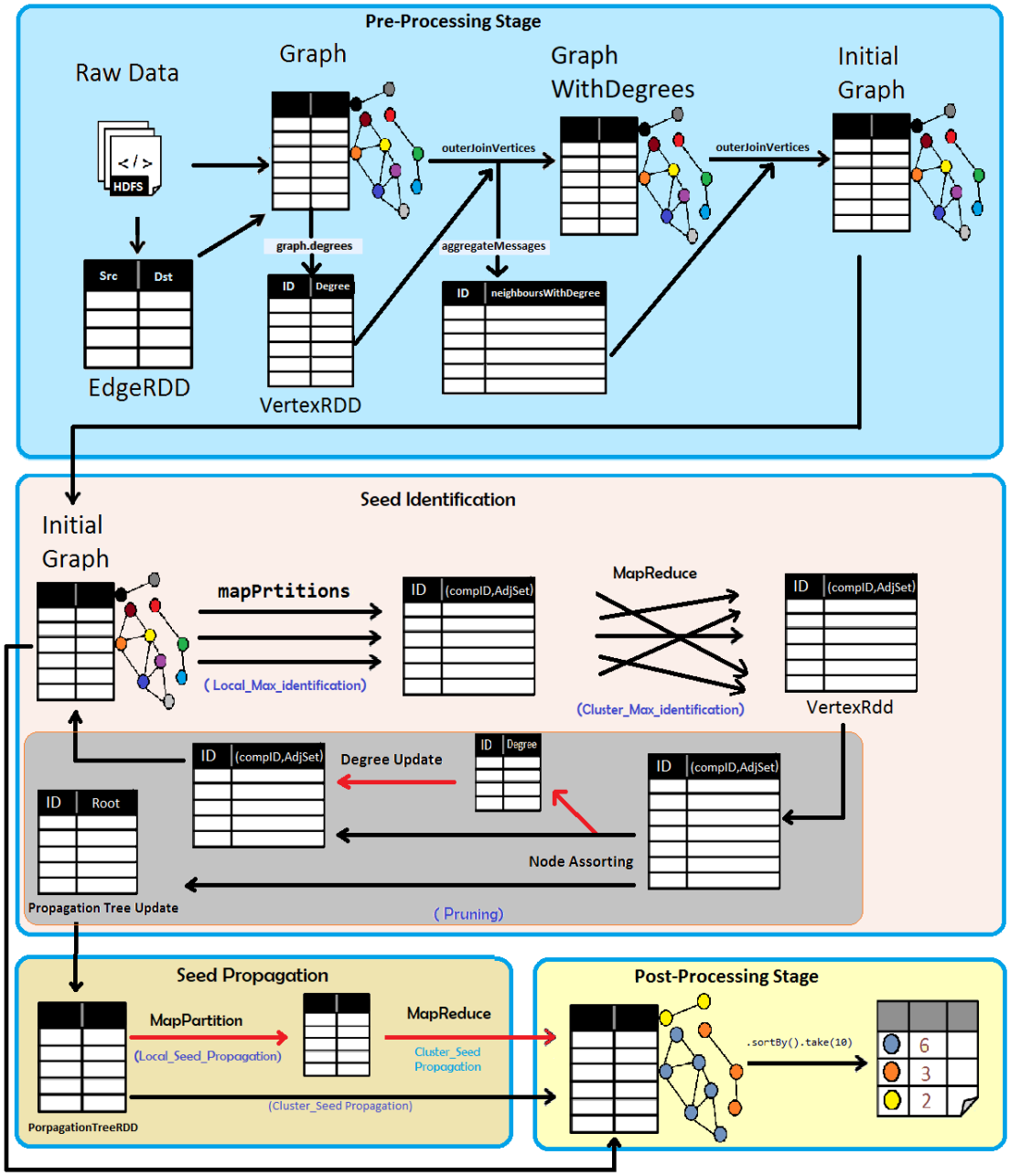


Figure 5-3: Algorithm Framework Model

i. Pre-Processing Stage:

At this stage, we take the raw data and prepare the initial graph needed in the Computing stage. Using GraphX, we build the graph from the raw data in datasets, which could be structured or unstructured. The graph used in our algorithm is built based on the adjacency list graph representation, where each node is aware of all the neighbours it is connected to. Here, each node stores in its property field a set of all its adjacent nodes $<u \rightarrow \text{adjSet} > \forall \underline{u} \in G = (V,E)$. Furthermore, to use the degree of the node as the main criteria for choosing the component identifier, we attach the node degree to the node id $u = (u_{\text{id}}, u_{d\text{egree}})$. Therefore, this stage will process the raw data and build a graph based on the adjacency list representation with each node aware of its degree and the degree of all other adjacent nodes.

ii. Computing Stage:

In a similar way to the Cracker algorithm, we divide this stage into two main phases: Seed identification and seed propagation.

In the first, we try to identify seed nodes, which are the nodes that will become the component identifier for the component it belongs to. This could be achieved by iteratively excluding non-seed nodes, which are nodes that are guaranteed not to be seed nodes, and they do not have any effect on chosen seed nodes. During this phase, a seed propagation tree is iteratively built, by adding the non-seeds nodes. Each identified node is rooted to a potential seed node in the propagation tree and deactivated in the processing graph. This phase finishes when all nodes are added to the tree, and there are no active nodes in the processing graph. Initially, computation in this phase is performed locally on each graph partitions across the cluster, as suggested in the proposed improvements. Then, computation is performed on cluster levels, similar to the original Cracker algorithm.

In the second phase, the seed nodes in propagation tree start to propagate their components' id to all their children nodes, and every node that receives a new components id will in turn propagate it to its children. This processes is performed iteratively until all the nodes belong to the same component are propagated with the same identifier, forming a tree rooted to a node represents the components identifier. Computation here is initially carried out locally similar to the previous step, and then across the cluster nodes. Further detailed explanation will follow in this chapter.

iii. Post-Processing Stage:

In this stage, we take the output graph from the previous stage, count the number of nodes in each component, and return the number of connected components with top largest 10 components, to check the accuracy of the algorithm.

5.5 Computing Stage:

The focus in this chapter is on explaining the proposed approach in finding connected components in large graphs. Thus, we only describe the *computation stage* here. A detailed description of the implementation of all stages is documented in the next chapter.

In this stage, we only work on the VertexRDD of the graph, which is an extended RDD that represent vertices (nodes) in GraphX to ensure there is only one entry for each node, and to pre-index the entries for fast and efficient join operations, where two VertexRDD with the same index can be joined efficiently. In VertexRDD each node is stored with its properties, such properties could be a set of adjacent neighbours or the degree property for each node. As mentioned before, Resilient Distributed Datasets (RDDs) is used to perform in-memory computations on large clusters where RDD's elements are partitioned across a cluster of nodes, so they can be operated on in parallel. Spark will control this process. Moreover, it

provides operators that help in implementing user specific optimisation as required, which fulfil the Role (1) described in the section 5-1.

In the *computation stage* finding CCs could be achieved using two phases (shown in figure 5-4), each phase contains multiple steps. Similar to Cracker algorithm[115] the two main phases are seed identification and seed propagation as presented in figure 5-5.

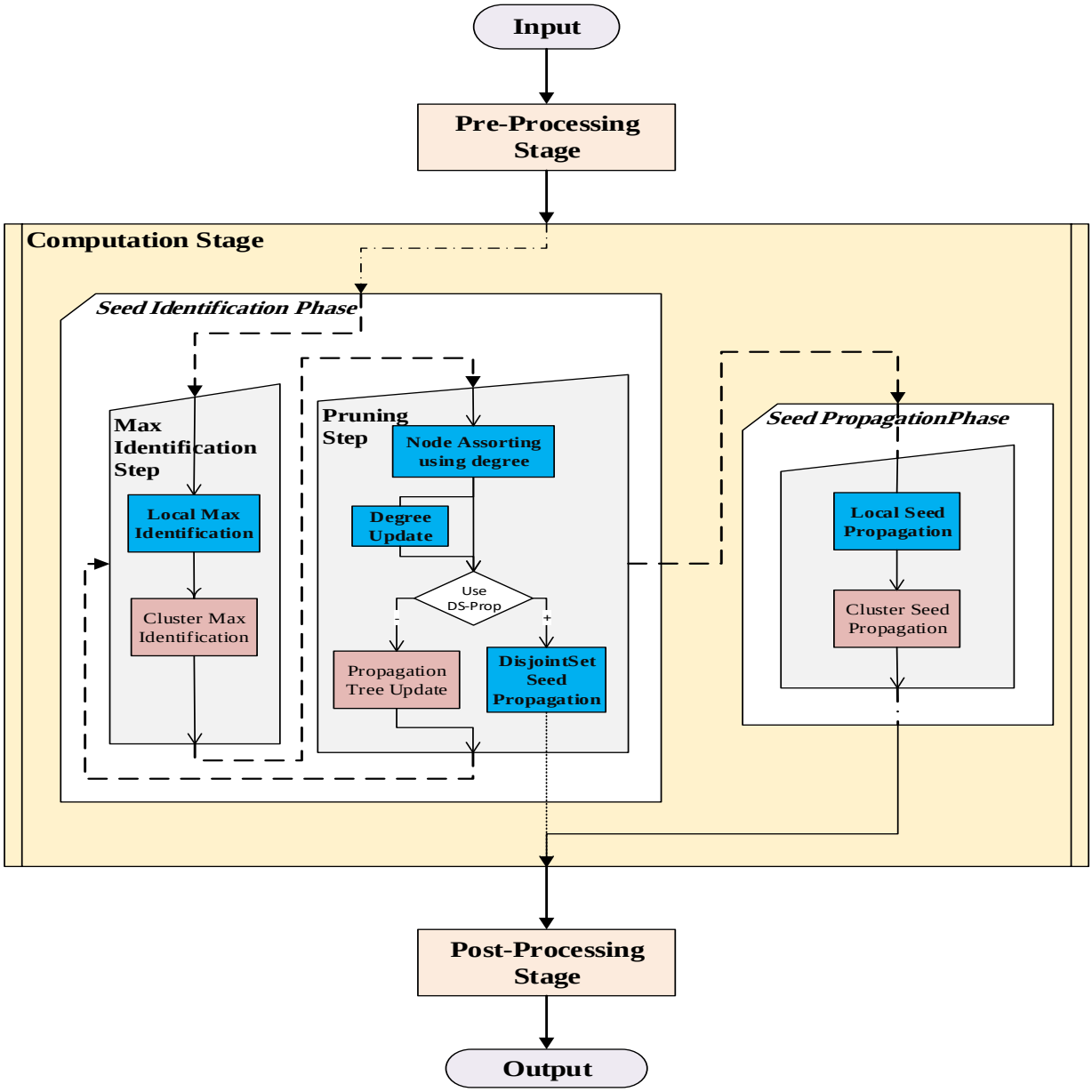


Figure 5-4: Computation Stage

Input: an undirected graph G = (V,E)
Output: a graph where every vertex is labelled with the seed of its CC

1. u.Active = True $\forall u \in G$
2. T ←(V, ∅)
3. t←1
4. G^t ←G
5. **repeat**
6. G^t_{update}← Local_Max_identification(u) $\forall u \in G^t$
7. HC^t ← Cluster_Max_identification(u) $\forall u \in G^t_{update}$
8. G^{t+1} ← Pruning(u,T) $\forall u \in HC^t$
9. t←t+1
10. **until** $G^t = \emptyset$
11. G^* ← Local_Seed_Propagation(T)
12. G^* ← Cluster_Seed_Propagation(T)
13. **return** G^*

Figure 5-5: Cracker-Degree Algorithm

The following is a detailed description of the two phases:

5.5.1 Seed Identification Phase:

In this phase, we scan the graph looking for connected components identifier ids (seed nodes), while excluding nodes that have no effect on chosen seed nodes and add them to the propagation tree. This reduces the graph size and saves the processing time in later iterations. The main idea is that each node interacts with its set of adjacent neighbours in to choose its potential connected component identifier. When a node is not recognised as a potential CC identifier for any of its neighbours and it is guaranteed that it will not become one in later iterations, this node will be deactivated in the main processing graph and excluded from being computed in later iterations. It will then be added to the propagation tree rooted to a seed node which is the potential CC identifier recognised among its neighbours. This phase finishes when all nodes are added to the tree, and there are no active nodes in the processing graph.

As shown in the Figure 5-4, this phase starts by taking the output graph of the pre-processing stage as its initial input graph and only works on the VertexRDD of the graph using the GraphX operator *graph.vertices*, where each node has a set of adjacent neighbours as its property. In the initial graph, all the nodes are in the active state, and therefore are all nodes are involved in the first computation. The computation in this phase is achieved by iteratively executing two main steps: (i) Max identification step, helps to identify non-seed node in the processing graph. (ii) Pruning step, where identified nodes are excluded from the processing graph and added to propagation tree. The following is a detailed description for the two steps.

i. Max Identification Step

The computation in this step is performed at two levels - local and cluster level. Initially, in local computation, each data partition is processed locally, with no need to shuffling any output data between partitions. In cluster computation, the output result requires data from all partitions and hence requires data shuffling between cluster nodes. Details of these two levels are presented in the following two sub-steps:

a) Local Max Identification Step:

In this step, data is partitioned across the cluster and is processed on the node where it is stored. Each node in the cluster could host more than one partition. Computation is therefore, carried out on each data partition a locally on the cluster node and no information is shuffled across the cluster. Each partition is therefore, considered to be an independent part of the graph, where we seek to identify the connected components in this partial graph. The purpose of this step is to find the local CC identifier for each group of vertices, which are connected in the data partition to which they belong. This will help to reduce the amount of processing required in later operations that could involve shuffling data across the cluster, as well as reducing the amount of shuffled data itself. This concept was recommended by Lin and

Schatzand [118] as in rule (2) in section 5-1, and was implemented in different ways as in [14], [119], [120].

For example Kiveris in his paper [14] used distributed hash table (DHT), where in map phase all nodes are loaded in a hash-table data structure and stored, each node is mapped to a its component identifier, initially the node itself. The process then repeatedly scans the table and updates the component identifier for each node with the minimum component identifier among neighbours until convergence. At this point, all the nodes will be mapped to their component identifier, which is represented by the minimum node id in that component.

In our work, we use the same approach; however, instead of using hash table, our approach is based on the Disjoint-Set data structure (also called union-find data structure) with path-compression so that the sets have a self-adjusting structure. The data is partitioned into non-overlapping dynamic sets with no intersection among them, each set has one of its members as the representative element of this set. This data structure was chosen as in order to increase the speed updates required for the component identifier for each node.

Generally, generating these updates in the hash table require depth-first-search algorithm (DFS) to do the job, DFS has the time complexity of $O\ (V + E)$. On the other hand, Disjoint-Set (DS) has time complexity of $O\ (\alpha(V) + E)$. Optimised DS provides near-constant-time operations with amortised time $O(\alpha(V))$ (bounded by $\alpha(V)$ the inverse Ackermann function where V is the number of nodes, which is a very slowly growing function that is less than 5 for any practical value), and turns out to be just barely more than O(1). Which make the time complexity of DS close to $O\ (V + E)$, which is very much close in performance to DFS. However, DS is preferred for situation where edges are continuously being added, and incremental computation for connected component is required. [122]

In our case, in the disjoint-set data structure, each set represents a connected component with the component id identifier as its representative node. We implement the sets as a rooted tree with each element as a node, and where each node points to its parent in the tree. The root of each tree is the representative node of that tree which in our case is the connected component identifier. To use disjoint-set, we need to define three main functions:

1. a function to initiate any new node as a new set (tree) with a pointer to the root node (parent). Initially, each node is rooted to itself by default.
2. A *root* function, that for each node returns the root of the tree this node belongs to. In this function path compression is recursively applied, which means that at any call for this function all the nodes on the same path between the requested node and the root node will update their parent to point directly to the root node (all nodes will become direct children of the root node), which will make subsequent similar operations much more efficient.
3. A union function, which will unify any two tree sets into one by pointing the root of one of them to the root node of the other. In this way, the two trees will be merged with one of the original root nodes becoming the new root. The new root node is chosen based on the degree of that node on the original graph, the node with the higher degree will become the new root node. If both root nodes of the trees have the same degree, the node with the larger id will be chosen. This operation is very much based on the *root* function defined earlier to check the root node for any given node. It will also use path-compression to adjust the tree structure with each call of the *root* function, and eventually lead to a more efficient performance in later calls.

In the local max identification step, we process each data partition alone on the local machine where it is stored. The aim is to try to accomplish part of the computation locally on small partitions of the data before the need to process the whole data across the cluster.

We use the disjoint-set data structure to help us find the local connected component identifier for each node in the partition it is stored on. Using the *mapPartition* function in Spark, each partition will be processed individually using our *LocalMaxIdentification* function to prepare the data for later processing, see figure 5-6 for the pseudo code of this function.

Input: a partition of the graph $G_p = (V,E)$
Output: G_{update} updated partition

```
1.  u ←(u id ,u degree,u comp, u adjSet) ∀u ∈ Gp
2.  ds = new DisjointSet()
3.  for (u in  Gp){
4.      if (ds not contains(u id) )){
5.          ds+= u
6.     }
7.     if (u adjSet  not empty)  {
8.        for (node  in  u adjSet){
9.          if (ds not contains(node.id))){
10.             ds+= node
11.         }
12.         union(item ,node)
13.       }
14.    }
15. }
16. Gupdate ← Gp
17. for (u in  Gupdate){
18.      ucomp = ds.root(u id)   ∀u ∈ Gupdate
19. }
20. return Gupdate
```

Figure 5-6: Local Max Identification Function

Each data partition G_p will be passed to the *LocalMaxIdentification* function where for each node *u* in that partition, the node will be added to build the disjoint-set *ds* in case it was missing, then add all adjacent neighbours of *u* (in $u_{adjSet)}$ to *ds*. The *union* operation will then unify the node *u* with each of its neighbours to be in the same tree. As consequence, the root of that tree will be updated to be the node with the max degree. After all the nodes in the partition are added to build the *ds* disjoint-Set data structure, we will have in *ds* a forest of

trees, each representing connected components in that partition with the component identifier as the root of each tree. Finally, each node in the data partition G_p will updates its component identifiers according to the root of the tree it belongs to in *ds*. The flowchart for this function is shown in figure 5-7.

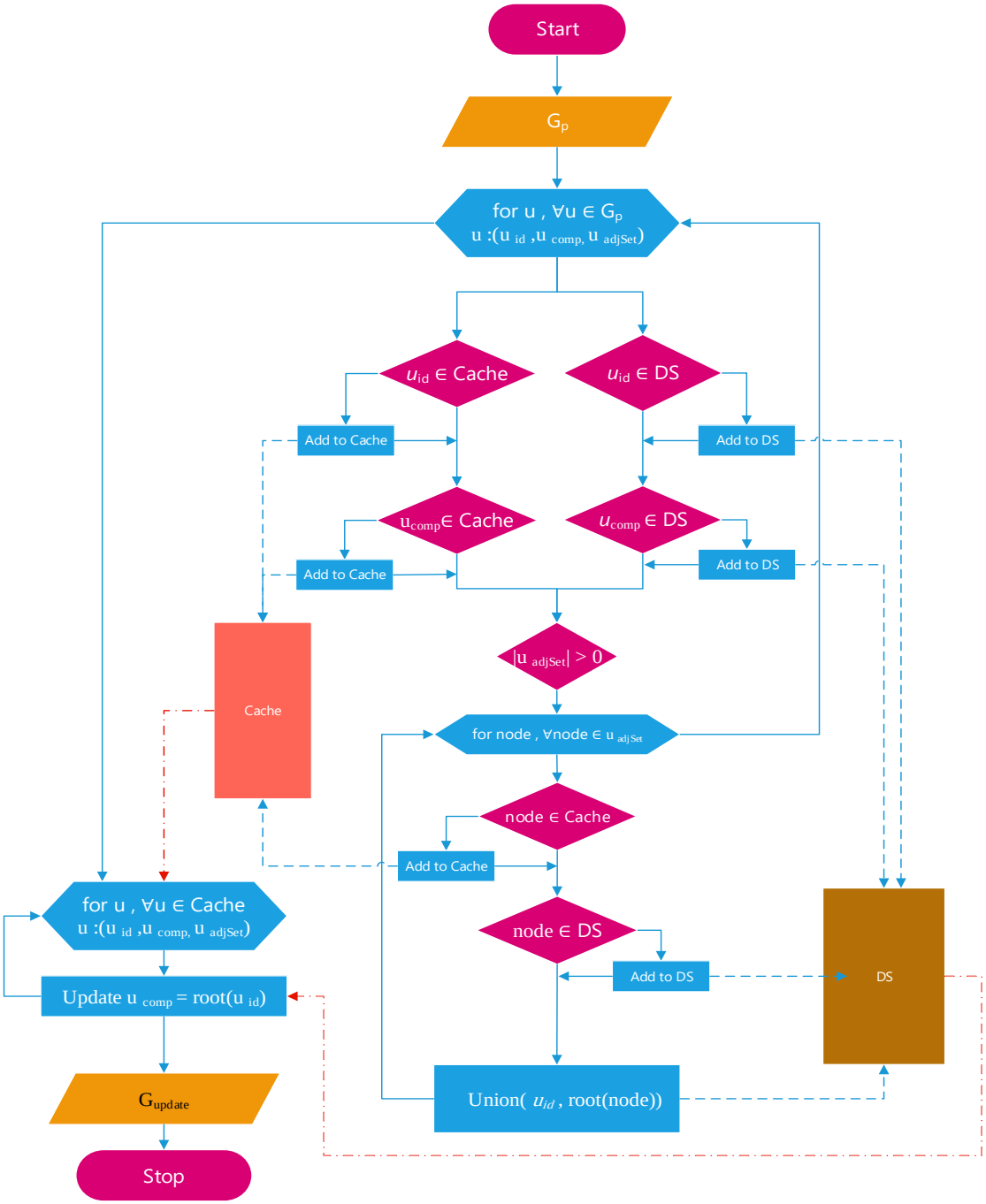


Figure 5-7: LocalMaxIdentification_Map

b) Cluster Max Identification

The main objective to be achieved in this step is to identify seed nodes in the graph. We achieve that by initially identify nodes that have no effect on the process of choosing the connected component identifiers (seed nodes) and exclude those nodes from being processed in later operations. This, in turn, will help to reduce both the size of data processed and the volume of transfer between cluster nodes. The outcome will lead to faster convergence.

Using the data received from the previous step, each node will be processed as follows:

1. For each node $u \in G^t$ that has the adjacent set of neighbours $\Gamma(u)$, we defined $\Gamma^+(u)$ as a set of the adjacent nodes of node u including the node u itself, $\Gamma^+(u) = \Gamma(u) \cup \{u\}$.
2. Compute $v_{max,}$ which is the node that has the maximum degree and largest id in $\Gamma^+(u)$ using the function *findMaxCompInSet,* which will be explained in details in section 6.2.1.
3. For each node v in $\Gamma^+(u)$ add an edge $(v \rightarrow v_{max})$ to HC^t, which is a directed graph that has the output of this step for each node.

These pseudocode for this step is represented in figure 5-8.

Input: a node $u \in G^t = (V,E)$
1. $\Gamma(u) = \{ v: (u \leftrightarrow v) \in G^t \}$
2. $\Gamma^+(u) = \Gamma(u) \cup \{u\}$
3. v_{max} = findMaxCompInSet ($\Gamma^+(u)$)
4. for ($v \in \Gamma^+(u)$){
5. add $(v \rightarrow v_{max})$ to HC^t
6. }

Figure 5-8: ClusterMaxIdentification Function

This step is implemented using one MapReduce job, where the map function processes all nodes in each partition of the graph using the function *map_ClusterMaxIdentification.* This will send v_{max} of $\Gamma^+(u)$ for each node u to all the nodes in $\Gamma^+(u)$. The flowchart for the map function is shown in figure 5-9.

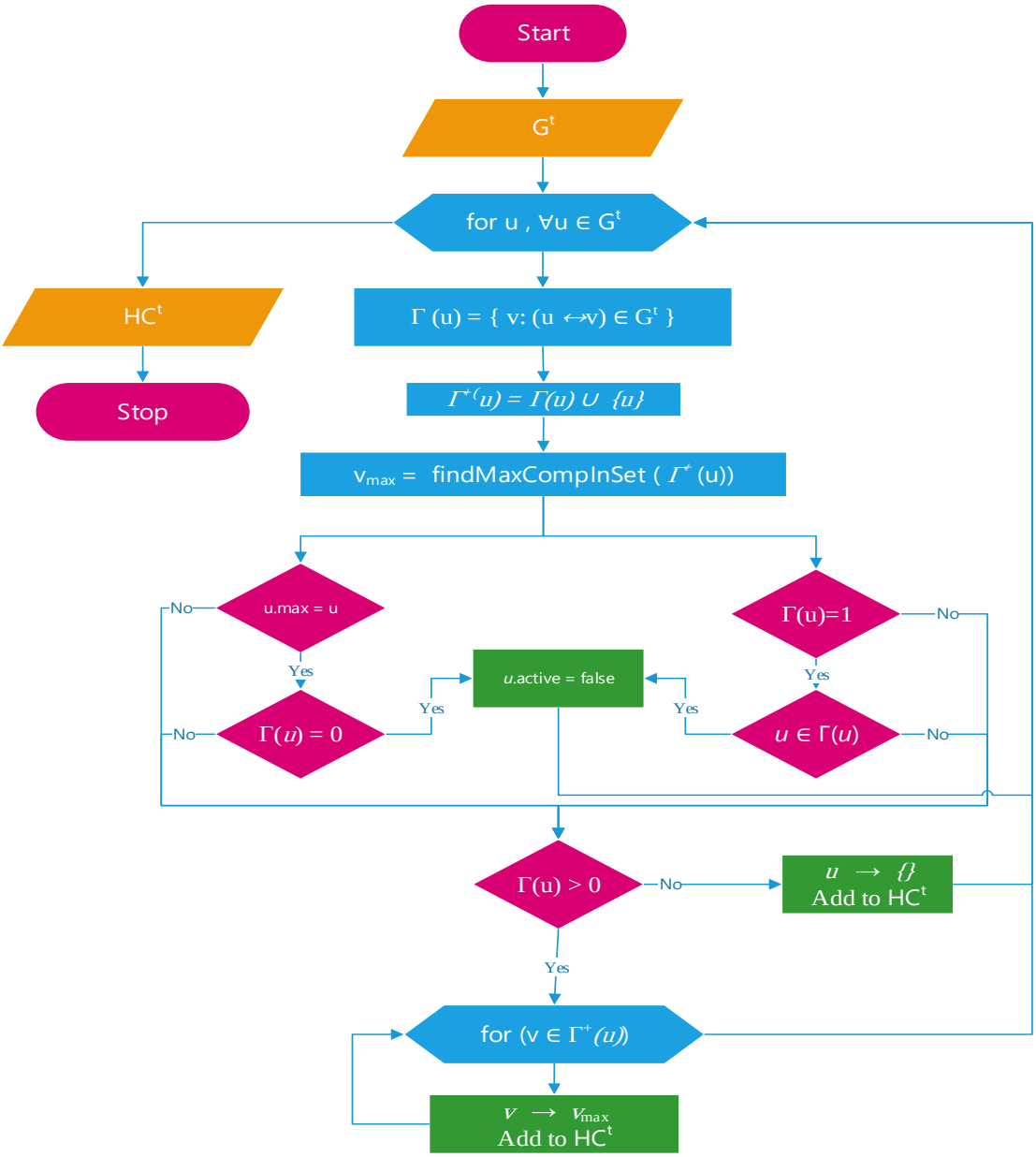


Figure 5-9: ClusterMaxIdentification_Map

The messages from the output of the map function are grouped for each node and then reduced using the function *reduce_ClusterMaxIdentification*. which will process each node as follows:

1. Aggregate the nodes notified by map functions and group them as the new adjacent set of the outgoing edges for the processed node.

2. Identify and assign the node with the max degree as the new component identifier (seed node) for the processed node.

Because non-seed nodes could be characterised by not being the local component identifier for any of their neighbours, they will not have any incoming edge in the output graph of the *Max Identification Step* (HC^t). This will help in identifying those non-seed nodes in next step (*Pruning Step)*. In the following we explain the pruning step, in which non-seed nodes are excluding from computation in this phase and added to the propagation tree to be processed in next phase.

ii. Pruning Step

The output from the *Max Identification Step* is a directed graph with a collection of edges that represents the relationship between each node and its neighbours, we call it HC^t. This output graph (HC^t) is used to isolate nodes that are guaranteed not to be seeds anymore and exclude them from taking part in later computations in this phase. Excluded nodes added to the seed propagation tree T, where each node is rooted to its local component identifier which is represented by the neighbour node with the maximum degree. Detailed explanation of the pruning step is further presented in the following three sub-steps:

a) Node assorting

This step will use HC^t as input and process all the nodes to produce two outputs:

1) A new set of edges that form the new graph G^{t+1} after excluding all the guaranteed non-seed nodes.
2) A set of edges to update the seed propagation tree T. These updates consist of excluded non-seed nodes with each one rooted to its local components identifier (seed node) in HC^t.

Using the data received from the previous step, each node will be processed as the following:

1. For each node $u \in HC^t$ that has the adjacent set of neighbours $\Gamma(u)$, we defined $\Gamma^+(u)$ which is a set of the adjacent nodes of node u including the node u itself $\Gamma^+(u) = \Gamma(u) \cup \{u\}$.
2. Compute $v_{max,}$ which is the node that has the maximum degree and largest id in $\Gamma^+(u)$ using the function *findMaxCompInSet* will be explained in detail in section 6.2.2.
3. For all the neighbours of u in $\Gamma^+(u)$, generate an undirected edge to $v_{max.}$ This will preserve connectivity in the graph in case node u needs to be excluded from G^{t+1}.
4. Check if node u is guaranteed not to be a seed node in order to exclude it from G^{t+1} and add it to T. As mentioned before, nodes which are not potential local component identifiers to any of their neighbours will not have any incoming edge from the others. In the previous step in the Cluster Max Identification, when a node is identified as a potential component identifier v_{max}, a collection of edges will be generated from all the nodes of $\Gamma^+(u)$ to v_{max} (see line 5 in figure 5-11). This will also include an edge from this node to itself ($v_{max} \rightarrow v_{max}$) in HC^t. Self-loop edge for a node u means that this was identified as a potential component identifier in the previous step and accordingly, nodes which have no existence in its neighbours set (self-loop) $u \notin \Gamma_H{}^t(u)$), are guaranteed not to be seed node and therefore could be safely excluded. These nodes are inserted in the seed propagation tree T where each will be rooted to its local component identifier ($v_{max} \rightarrow u$) (see line 11 in Figure 5-11).
5. Seed nodes also identified in this step. The main purpose of this step is to reduce the size of the graph after each iteration by excluding non-seed nodes. Eventually, for each component, the last active node processed is the seed node, which is the component identifier for all neighbour nodes that belong to the same components, which has been

previously excluded. As mentioned before, self-loop edge for a node indicates that it was identified as a local seed node in the previous step. However, when the node has no other edge, other than the one to itself, means it is either the end root node for other nodes in the propagation tree or a single component node where its component identifier is itself. Identified seed nodes are then deactivated and inserted in the seed propagation tree T where each will be rooted to itself (u →u) (see line 15 in Figure 5-11).

The flowchart of node assorting is presented in figure 5-10 and figure 5-11 for pseudocode.

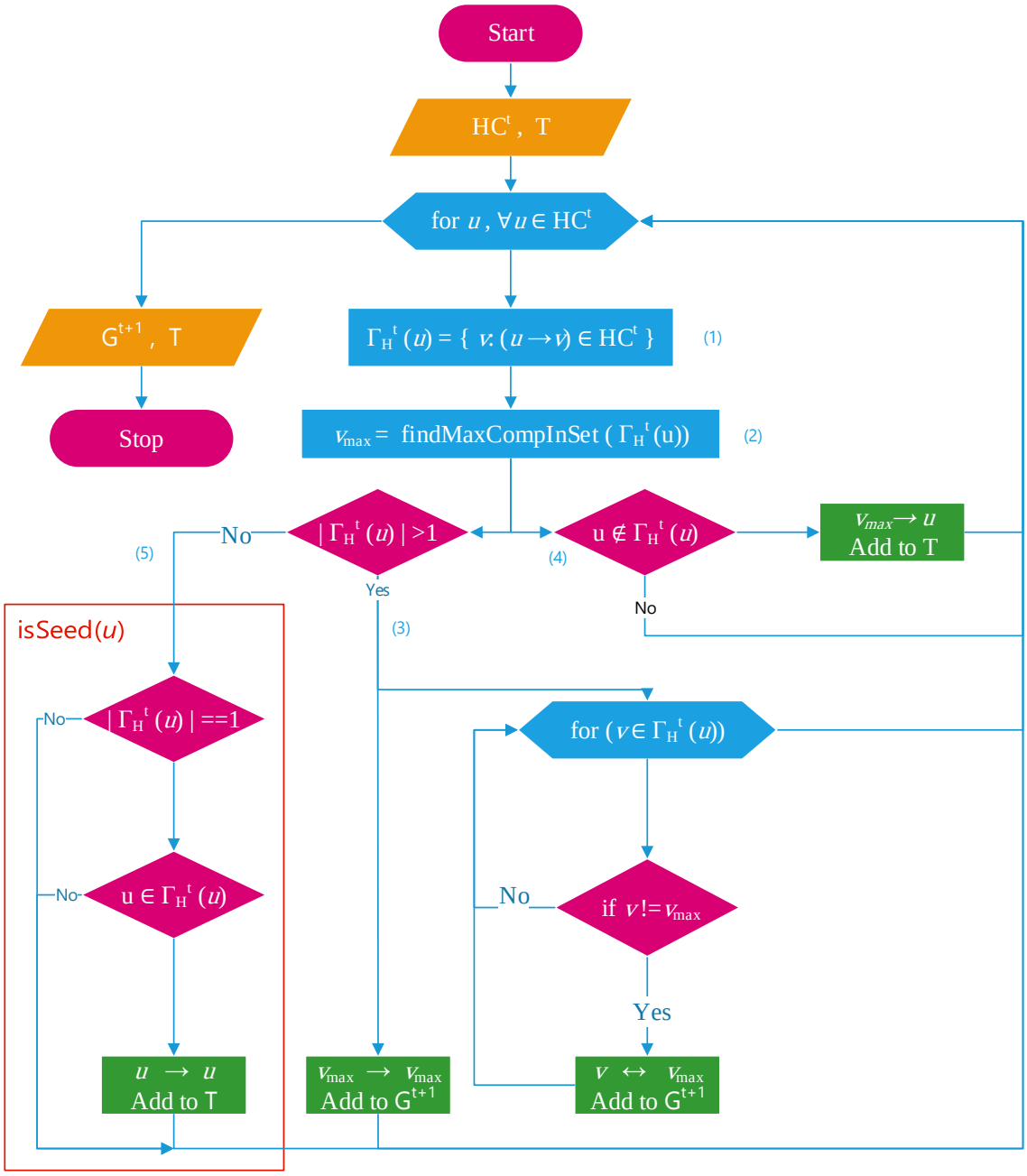


Figure 5-10: Node Assorting Flowchart

Input: node u , $\forall u \in HC^t$
seed propagation tree T
Output: G^{t+1}
update T

1. $\Gamma_H^t(u) = \{ v: (u \rightarrow v) \in HC^t \}$ # neighbour of node u in HC^t
2. v_{max} = findMaxCompInSet (Γ_H^t(u)) # max node in u neighbours Γ_H^t(u)
3. if ($|\Gamma_H^t(u)| > 1$) {
4. for ($v \in \Gamma_H^t(u) \setminus v_{max}$) {
5. Add ($v \rightarrow v_{max}$) to Gt^{+1}
6. Add ($v_{max} \rightarrow v$) to Gt^{+1}
7. }
8. }
9. if ($u \notin \Gamma_H^t(u)$) {
10. u.Active = False
11. Add ($v_{max} \rightarrow u$) to T
12. }
13. If (IsSeed (u)) {
14. u.Active = False
15. Add ($u \rightarrow u$) to T
16. }

Figure 5-11: Node Assorting

b) Degree Update

The main concept in our approach is to use the degree as the primary criteria for selecting the connected component identifier. However, new edges are created, and nodes are excluded in each step, thus the structure of the graph changes with each iteration.

The Graph structure changes as its size shrinks due to nodes both being deactivated and added to the seed propagation tree. Thus, nodes' degrees continuously changes, especially for active nodes that act as potential seeds. This is because non-seed nodes are removed from their neighbourhood or when components merge. The graph degree was evaluated during the first step in the pre-processing stage. However, as the graph structure changes the node degrees need to be updated to reflect the actual graph structure after each iteration. To

address this problem, we add a lightweight MapReduce job to evaluate the new degree values for each node (shown in Figure 5-12). From each node, a message will be sent to each of its neighbours, and count the number of messages received from its adjacent neighbours.

This information gathered will then be used to update the degree for all nodes in G^{t+1}. The updates will be sent to each cluster node, then using the *mapPartition* operator in Spark (which process data locally with no data shuffling required), in each partition the degree for each node will be updated in addition to both its component identifier degree and the degree for each of its neighbours (see Appendix1, Update Degree section).

Map: a node $u \in G^{t+1}$
1. $\Gamma(u) = \{ v: (u \leftrightarrow v) \in G^{t+1} \setminus u \}$
2. for ($v \in \Gamma(u)$){
3. emit ($v \rightarrow 1$)
4. emit ($u \rightarrow 1$)
5. }

Reduce: u, $\Gamma(u)$
6. Emit (u, Γ(u).sum)

Figure 5-12: Degree Update

c) Propagation Tree Update

In the Node Assorting step, the aim is to exclude non-seed nodes which make no difference on the process of choosing the component identifier for any other node. The excluded nodes will be sent as updates to the propagation tree T after each iteration, where they are merged into T. The updated format will be a set of directed edges, each root the identified non-seed node to it local seed node ($v^{max} \rightarrow u$), where u is the non-seed node and v^{max} is the root of that node u. At the beginning of the step, the propagation tree T is an RDD (we call it *PropTreeRDD*) initialized with each node rooted to itself (each node is an independent tree itself with the root node the same as the node itself). However, as updates arrive following each iteration of the *Max Identification Step*, the new updates are added to *PropTreeRDD.*

The *Seed Identification Phase* will repeat as long as there are nodes to process in G^t. Nodes will continue to migrate from G to T at each iteration until there are no more active nodes $G^t = \emptyset$ (see Figure 5-11, Line 10). This is confirmed by counting the output of the *Cluster Max Identification Step*. When it is zero, this phase will terminate (See appendix A).

5.5.2 Seed Propagation Phase:

When all the nodes have been added to the seed propagation tree T, the process of seed propagation starts.

Seed propagation Tree T has the following characteristics:

- Non-seeds nodes are rooted to their local seed nodes.
- Local seed nodes are rooted to other seed nodes with a higher degree.
- The seed node, which is the root of the tree, is rooted to itself.

We could use two methods for the seed propagation process, either by processing the propagation tree after the *seed identification phase* has finished, or by updating the propagation tree during the process of *seed identification phase.* The two methods are described in the following:

i. Seed Propagation.

In this phase, seed nodes start to iteratively propagate their component ids to all their children nodes, and every node that receives a new components id will propagate it to its children. In the end, all nodes belonging to the same component will be propagated with the same identifier, forming a tree rooted to the components id node. Computation is initially performed locally and then across cluster nodes. Further details of the two computation levels in this step are presented in the following:

a) Local Seed Propagation.

The computation in this step is carried out locally on each data partition on the cluster node and no information is shuffled across the cluster. The root nodes will send its component identifier to the child nodes on the same partition using the map function *mapPropagate* (see Appendix A.2). Messages for each node in the same partition are grouped locally. Its component identifier is accordingly updated using the reduce function *reducePropagate*. This operation iterates until no further updates are generated. Each partition is considered as an independent part of the graph, and here we seek to identify the connected components in this partial graph. The purpose of this step is to find the local CC identifier for each group of vertices, which are connected in the data partition they belong to. This will help to reduce the processing required in later operations that could involve shuffling data across the cluster. It will also reduce the number of shuffled messages.

b) Cluster Seed Propagation

After the nodes in each partition have updated its component identifiers according to the other root nodes in the same partition, the same process is repeated across cluster nodes. Here for each connected nodes' tree the updates are generated from the root to the child nodes, and each node notifies its child nodes with its component identifier, which is the id of its root node. This step reaches convergence when the component identifier for the leaf nodes in the tree are the same as the main root node.

In the end, we will have a forest of trees where each tree represents a connected component with the root node as the component identifier.

ii. DisjointSet Seed Propagation

As indicated previously, all steps suggest the implementation of the seed propagation step after the Seed Identification Phase has finished, where all nodes would be added to the

propagation tree T. However, here we suggest building the propagation tree and caching it in the memory of the driver node in the cluster. Because at the end of each iteration of the *Seed Identification Phase* all the information will be gathered on the driver node to generate the output, there is no need to wait until the end to gather propagation tree update and process them. Therefore, such updates could be merged into the propagation tree dynamically as they are generated using the disjoint-Set data structure (explained before). This could significantly increase the load on the driver node, however, usually driver nodes in clusters are chosen with the highest specification compared to other worker nodes. Thus, the overload caused by implementing this step could be handled by increasing the specification of the driver node only.

5.6 Summary

This chapter began by describing the general pattern used in MapReduce algorithm for finding connected components, then identify two kind of properties that could be targeted to apply improvements, which are Graph Structure properties and Processing System properties. Next, in respect to these properties the proposed approach to enhance the performance of CC algorithm was introduced. First, it adopted the degree property in graph as the main criteria for choosing the component identifier. Second, it introduced optimisations based on properties provided by the processing system. It went on to suggest the use of Cracker algorithm to apply the proposed approach, and gave a detailed description of framework design to integrate suggested improvements in the Cracker algorithm. The next chapter describes the procedures and methods used in the design and implementation of the proposed approach.

Chapter 6: Design & Implementation

6.1 Introduction:

Large graph distributed processing is very fast-moving research and development area, in which new systems and new paradigms regularly appear. Hadoop is considered the most common tool used for big data. Thus, the initial plan was to use Hadoop in our algorithm implementation and MapReduce as our programming framework with Python. However, as we investigated the area in more detail and started to develop a better understanding of the field, we found an excellent opportunity to change our plan and use Spark instead. Spark is considered the new alternative for MapReduce. It provides the ability to cache some parts of the data in memory, allowing it to be used it in later iterations. This capability could help to reduce the number of iterations and decrease the intermediate communication load between iterations, leading to potential performance enhancement. This performance enhancement could be achieved without losing the ability to expand the size of the data beyond the size of the memory resources available. If the cache files grow and no longer fit in memory, Spark can distribute the cache files on local disks and recompute them later when needed. As a result, Spark could run programs up to 100 times faster than Hadoop MapReduce in memory, or 10 times faster on disk.[68]

For the graph processing, GraphX on top of Spark was a good candidate, and was chosen for several reasons:

1. Apache Spark is a very active project, more so than Hadoop. More than 100 developers from over 200 companies have contributed to Spark. 19 organisations are committed to the project. This is very promising as Spark has recently started to replace the MapReduce paradigm.[68]

2. GraphX is Spark's API for graphs and graph-parallel computation. It has a growing library of graph algorithms, which provide seamless interfaces and operations with both graphs and collections while its performance competes with the fastest graph processing currently available.
3. Its high-level Scala API provides the ability to efficiently use the wide range of operations and optimisations available for use in our implementation.
4. The storage abstraction Resilient Distributed Datasets (RDDs) capability used in Spark achieves efficient fault tolerance using the notion of lineage, which enables automatic recreation lost data partitions.
5. GraphX can retain graphs or any RDD in memory for later use. This is essential for iterative graph algorithms, and even when RDD does not fit in available memory, it can be stored on disk in a way similar to MapReduce.
6. Using the operators provided in GraphX, can optimise communication in finding connected components algorithm implementation by controlling the partitioning of the RDDs.

Spark 1.0.0 was released in 2014. Since then it has received numerous updates and additions. In our implementation, we have upgraded to the latest stable version when a new update is available. We used Spark 2.0.0 in 2016 for development and testing on a single machine However for the evaluation of our work on a cluster; we used the latest available version as we will describe in next chapter. GraphX is a part of the Apache Spark project, so it is tested and updated with each Spark release. GraphX is built using Scala, running on a Java VM. It is a general-purpose programming language providing support for functional and object-oriented programming.

For local implementation and testing Spark was downloaded to a single Windows PC machine (not a cluster) and built using sbt. We used also IntelliJ IDEA, an open source integrated development environment (IDE) tool.

For our initial implementation and testing, Spark was set up to work in a simple standalone deploy mode with all implementations developed using the Scala language. In the initial local testing, we were able to launch a standalone cluster, which is established, by manually starting a master and workers together on one single machine, then run our program, or by using one of the launch scripts to launch a Spark standalone cluster with each run of the program [123].

6.2 Framework Implementation

The framework design presented in the previous chapter (section 5-4), is based on the work of Lulli and his Cracker algorithm for finding connected components in large graphs[115], [116]. It extends his work to optimises the performance by using a different approach for choosing the component identifier (the degree property in our case), then uses features and operators available in GraphX to optimise and enhance the data flow in the algorithm by processing the data locally when applicable to reduce both the time for convergence and the number of shuffled message between iterations.

This framework starts by taking input data in its raw format, preparing it and transforming it into the format required in the processing stage which is where we apply our approach to finding the connected components. It completes by accessing and arranging the output in a presentable format ready for evaluation.

In the following, we review each stage in the framework and provide a detailed explanation of the implementation process.

6.2.1 Pre-Processing Stage

In this stage, we take the dataset and prepare the data to build the initial graph for later processing. As explained previously, the initial graph for processing should be built using the adjacency list representation, where each node has a list of its neighbours.

$$< u \rightarrow \text{adjSet} > \forall u \in \text{G} = (\text{V}, \text{E})$$

We developed two functions. The first was used to test other algorithms, on the graph with an adjacency list generated, where each node is attached to its list of neighbour nodes. The second function is used to create a graph with an adjacency list similar to the previous one, but with the addition that each node is aware of its degree and the degree of all other nodes with which it interacts.

For the adjacency list graph the following steps are followed:

1. Use raw data it to create a ***graph*** in GraphX using the GraphX operators provided.
2. Compute the adjacency list for each node in the graph and add it to its property, to create the *Adjacent List Graph*.
3. We could assign the component identifier for each node *compID*, from each node's adjacency list take the minimum and compare it with the node ID and assign the minimum to *compID*. At completion, *Initial Graph* is created, in which each row of the VertexRDD has the form *<id, (compID, adjSet))>* with the *compID* set to the minimum ID.

To create the adjacency list graph with degree awareness the following steps are followed:

a. Use the raw data it to create a ***graph*** in GraphX using the GraphX operators provided.

b. Compute the degree of each node and add it as its property.

c. Compute the adjacency list for each node in the graph to create a new graph, the *Adjacent List Graph*, where each node has its degree, and a set of the adjacent nodes attach to it.

d. Output the graph for the next stage with each node having its degree and a set of pairs of adjacent nodes and their degrees *<nodeId, (nodeDegree, Set(adjacent nodes))>*.

In this stage, we process the raw data available on the cluster, usually on a distributed file system like HDFS or Amazon S3. Next, we generate the graph in GraphX using *graph builder* operators. The raw Data could be structured or unstructured. For example, data could be in a text file in a format like *<source_id, destination_id>*, or an XML web content format that needs some cleaning and preparation to be in the right format for processing.

After creating the graph, the adjacency list for each vertex needs to be generated. For this purpose, we created a function called *adjacencyListGenerator()*, which takes a graph as input and returns an output graph with each vertex with its adjacent list. In the VertexRDD of the output graph, the property field for each vertex has a set of adjacent neighbour vertices.

The *adjacencyListGenerator()* function performs the following steps (see figure 6-1): The function starts by initializing each vertex (node) property with its own VertexID.

1) Then we call GraphX operator *collectNeighborIds()* on the graph, which will Collect the neighbour vertex IDs for each vertex and return it in a VertexRDD.

2) We convert the array of vertices into an mutable *HashSet.* A mutable *HashSet* type in Scala is considered here as it is iterable and contains no duplicate

elements. Here operations such as lookup, add, and remove, can take effectively a constant time depending on some assumptions such as the maximum length of a vector or the distribution of hash keys[124].

3) Finally, take the VertexRDD *adjGraph* created and generate a new *graph* using the same EdgeRDD from the original graph, and return the new *graph.*

```
def adjacencyListGeneratorOpt[VD:ClassTag, ED:ClassTag](graph: Graph[VD, ED]): Graph[Set[VertexId], ED] =
{
  val WorkGraph = graph.mapVertices { case (vid, _) => (vid) }
  val nbrs = WorkGraph.collectNeighborIds(EdgeDirection.Either).cache()
  val nbrsVerts: VertexRDD[Set[VertexId]] = nbrs.mapValues ( (vid, nbrs) => Set(nbrs.toSet.toArray: _*))
  val adjGraph: Graph[Set[VertexId], ED] = Graph(nbrsVerts, graph.edges)
  adjGraph
}
```

Figure 6-1: adjacencyListGenerator function

Using the new graph generated in the previous step, we apply *.mapVertices()* operator, which will take the property for each vertex *(compID, adjSet)* and apply the function *findMaCompInSet()*. The function will find the vertex that has the minimum ID in the *adjSet*, compare it to the *compID* and return the minimum (see figure 6-2).

```
def findMincCompInSet(compID: VertexId, set: Set[Long]): VertexId = {
  var setMin = compID
  if (!set.isEmpty) {
    setMin = set.min
    if (setMin > compID) setMin = compID
  }
  setMin
  }
```

Figure 6-2: findMincCompInSet function

The output of this step is a graph (we call it the *Initial Graph*) with the *compID* set to minimum ID. We will use the *Initial Graph* later in testing CC algorithms and comparing it with the proposed approach of using the degree in CC algorithms.

For the proposed approach, we will use a graph which has degree awareness. This means all vertices in the graph, know their degrees and the degree of all the other nodes they directly interact with. To create the adjacency list graph with degree awareness, we created a function called *adjacencyListGeneratorDgOpt()* that takes a graph as input and returns an output graph.

In the VertexRDD of the output graph, the property field for each vertex has the vertex degree and a set of adjacent neighbour vertices with their degrees.

The *adjacencyListGeneratorDgOpt()* function performs the following steps (figure 6-3):

1. The function starts by computing the degree of each vertex using a built-in operators in graphX. This is achieved using a Pregel-like (bulk-synchronous message-passing) implementation inside GraphX. Then the degree is added as the vertex property in the graph.
2. We call the operator *aggregateMessages()* on the graph, to aggregate values from the neighbouring edges and vertices of each vertex and return it in a VertexRDD as the vertex property. As a result, we will have with an array of neighbouring vertices.
3. We convert the array of vertices into a set of neighbouring vertices each stored with its ID and degree.
4. Finally, we take the VertexRDD *neighboursWithDegree* graph created and used it to generate a new *graph* using the same EdgeRDD from the original

graph, and return the new *graph as the initial graph*. This is achieved using *outerJoinVertices* operator.

```
def adjacencyListGeneratorDgOpt[VD:ClassTag, ED:ClassTag](graph:Graph[VD,ED]): Graph[(Int,
Set[(VertexId,Int)]), ED] = {

    val degrees = graph.degrees

    val graphWithDegrees = graph.outerJoinVertices(degrees){(_, _, optDegree) => optDegree.getOrElse(1)}

    val WorkGraph = graphWithDegrees.mapVertices { case (vid, degree) => (vid,degree) }

    val neighboursWithDegree = WorkGraph.aggregateMessages[Set[(VertexId, Int)]](

        sendMsg = triplet => {

            val srcWithDegree = triplet.srcAttr

            val dstWithDegree = triplet.dstAttr

            triplet.sendToDst(Set(srcWithDegree))

            triplet.sendToSrc(Set(dstWithDegree))

        },

        mergeMsg = (x, y) => x ++ y

        ).mapValues(x=>(x.size,x))

    val emptySet:(Int,scala.collection.immutable.Set[(VertexId,Int)])= ( 0, scala.collection.immutable.Set())

    val adjGraph = graph.outerJoinVertices(neighboursWithDegree){(_, _, optDegree) =>

                                                                    optDegree.getOrElse(emptySet) }

   adjGraph

}
```

Figure 6-3: adjacencyListGeneratorDg function

From the new graph generated in the previous step, we apply the *.mapVertices()* operator that will take the property for each vertex *(compID, adjSet)* and apply the function *findMaxCompInSet (compID,adjSetDg)*. The function will find the vertex that has the max degree in the *adjSetDg* using *maxDg()* function, which will identify the node that has the max degree. If both nodes have the same degree, it will choose the one with the higher ID. At completion, it will compare it to the *compID* and return the max using the same function (See figure 6-4).

```
def findMaxCompInSet (compID: (VertexId,Int), setDg: Set[(VertexId, Int)]): (VertexId,Int) = {
  def maxDg(ver1:  (VertexId,Int), ver2:(VertexId,Int)):(VertexId,Int) ={
            if (ver1._2 > ver2._2) { ver1
             } else if (ver1._2 == ver2._2){
               if (ver1._1 > ver2._1) ver1
               else ver2
             } else ver2
  }
 var setMaxDg = compID
 if (!setDg.isEmpty)   setMaxDg = setDg.reduceLeft(maxDg)
 maxDg(setMaxDg , compID)
}
```

Figure 6-4: findMaxCompInSet function

The operation of generating the adjacency list graph with degree awareness is more expensive that one generates the adjacency list without considering the degree. However, this operation is executed only once to prepare the graph for processing in the next stage.

6.2.2 Computing Stage

In this section, we review the implementation process and code structure for the computing stage documented in section 5.5. It is worth noting that in respect with the objectives of this research, we experimentally apply all the proposed improvements on the fastest existing CC algorithm, which is the Cracker algorithm in our case. Hence, the coding structure in many parts of this work is based on the original implementation of the Cracker algorithm[23], with further implementation of our extensions that were proposed in section 5.2.

The output from the pre-processing stage is a vertexRDD, which is representation of each node and its properties. The vertexRDD holds in the property field for each node, the node

[23] https://github.com/hpclab/cracker

degree and all its adjacent nodes with their degrees. Processing is carried out by exchanging messages between nodes and accordingly each node use the receive messages to update its properties such as: its degree, its component identifier, or its adjacent set of neighbours. Therefore, the code structure uses different kinds of message-classes. In each phase, different node properties need to be exchanged and for each we have different class of message (see Appendix A- Classes). For example, *message_Identification* class used for message used in the *max identification & pruning steps.* *message_Tree* class used for generate update messages for seed propagation tree T. *message_Propagation* used in the *Seed Propagation Phase* to hold the updates are generated from the root to the child nodes, where each node notifies its child nodes with its component identifier. Figure 6-6 presents the class diagram of the classes used for exchanging messages between nodes.

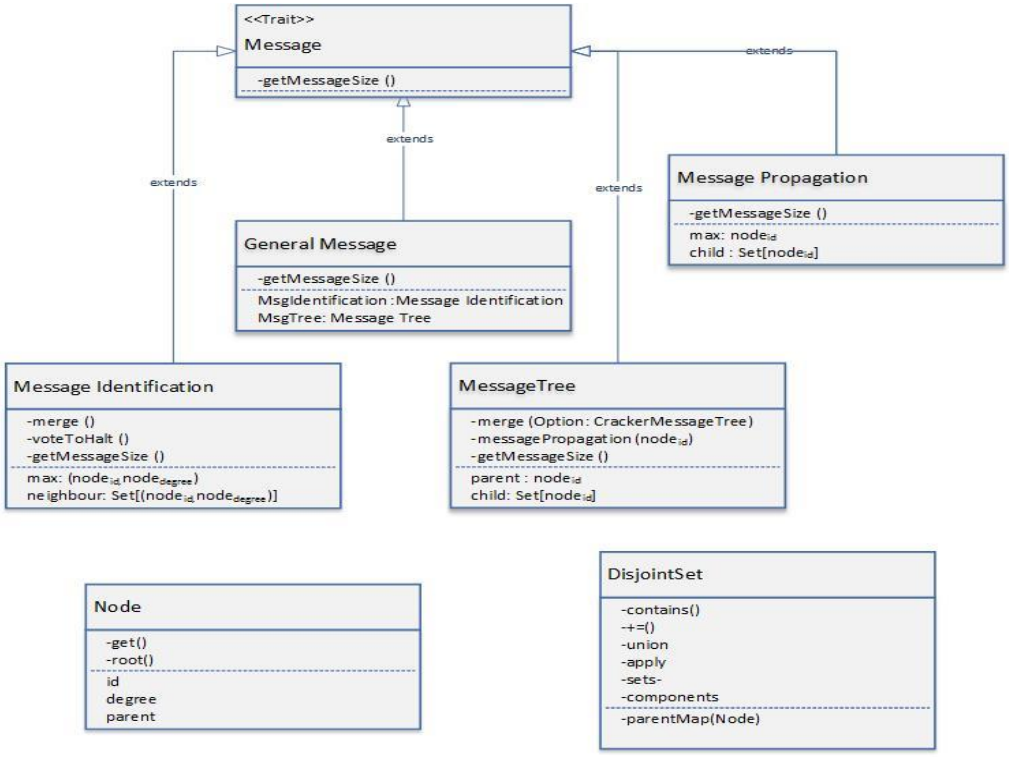


Figure 6-5: Class Diagram

The two main phases of the algorithm (described in the framework model figure 5-3 and explained section 5-5) are seed identification and seed propagation.

i. Seed Identification Phase

In which two main steps are performed:

a) The Max identification Step

The aim of this step is to update the node properties in such a way that it will be possible to exclude non-seed nodes from the computation and add them to the propagation tree in next step. This step is implemented in two levels:

- *Local Max Identification*, in which we try to accomplish part of the computation locally on small partitions of the data before the need to process the whole data. Therefore, we developed the function *LocalMaxIdentification* described in section 5.5.i.a (data flowchart in figure 5-7). We apply the function using the *mapPartition* operator in GraphX.The function will use the disjoint-set data structure to find the local component identifier for all nodes in the same partition and update their component identifiers accordingly. (appendix A.2.i.a and Appendix A.Classes for code)
- *Cluster Max Identification*, in which we try to identify non-seed nodes. We achieve this using one MapReduce job. The map function used to generate the messages is called *map_ClusterMaxIdentification* (described in section 5.4.b). The output is reduced by using the function *reduce_ClusterMaxIdentification*, which will aggregate the nodes notified by the map functions and group them as the new adjacent set of outgoing edges for the processed node, then identify and assign the node with the max degree as the new component identifier. As a result, any node that is smaller than all its neighbour nodes in term of degree and id (not the local component identifier for any adjacent nodes), this node will not receive any incoming edges. (appendix A.2.i.b for code)

b) The Pruning Step

In which three further operations are required:

- *Node assorting*, where identified non-seed nodes are separated from the rest of the nodes. These will be added to the propagation tree later and the rest of nodes will be processed again using *max identification step*. Node assorting requires one MapReduce job. For the map phase, nodes are assorted by into two groups of nodes, which are (a) potential seed nodes and (b) non-seed nodes.

  A function called *map_Pruning* was created for purpose (described in section 5.5.1.i.a, data flowchart in figure 5-10, code in appendix A.2.i.c). The *map_Pruning* will process each node and it adjacent nodes and generate new messages, as either *IdentificationMessages* or *TreeMessages*. The former are messages that identified as potential seed nodes and they will be processed again in the next iteration of the *seed identification phase*. The latter are the non-seed nodes and they will be added to the propagation tree T and processed later in the *seed propagation phase*.

  For the reduce phase, a reduce function called *reduce_Pruning* function was created to merge and reduce messages for each node. Next, these messages are filtered into separate groups of nodes; nodes that need to be added to the propagation tree and the rest of nodes. Code structure is shown in figure 6-6.

```
val tmp = ret.flatMap(item => map_Pruning(item))                    \\ apply the map function

val tmpReduced = tmp.reduceByKey(reduce_Pruning).cache              \\ apply the reduce function

val tmpReduced_MsgIdentification= tmpReduced.filter(t => t._2._2.MsgIdentification.isDefined)
.map(t => (t._1, (t._2._1,t._2._2.MsgIdentification.get)))          \\ filter the potential seed
nodes

val tmpReduced_MsgTree  =tmpReduced.filter(t => t._2._2.MsgTree.isDefined)
.map(t => (t._1, t._2._2.MsgTree.get))                              \\ filter the non-seed nodes
```

Figure 6-6: Node Assorting Code

- *Update Propagation Tree*

  Initially, all nodes in the propagation tree RDD all are rooted to themselves, in another word, the component identifier refers to the node itself. In the implementation the component identifier is set to (-1) to denote this.

```
propagationTreeRDD = Option.apply(ret.map(t => (t._1, new DgCracker_MsgTree(-1, Set()))))
```

Nodes gathered in the *tmpReduced_MsgTree* (which represent the non-seed nodes), are used to update the propagation tree T. This is achieved by simply merging the two RDDs (*propagationTreeRDD, tmpReduced_MsgTree)* using the *union* operator in GraphX (detailed code is presented in appendix A.2.i.c).

```
propagationTreeRDD = propagationTreeRDD.get.union( tmpReduced_MsgTree)
```

- *Update Degree*

The graph structure changes as its size shrinks due to some nodes being added to the seed propagation tree T and some excluded from the processing graph G. Thus, the nodes' degree continuously changes. To reflect those changes, we add the MapReduce job to evaluate the new values of degree for each node. Each node will send a message for each of its neighbours and count the number of messages received from its adjacent neighbours, which will return the new degree. These values are collected in a HashMap structure and then broadcasted to each node in the cluster.

A *mapPartition* operation will then run to update the degree for each node within each partition from the hash map which exist on the same machine without the need to shuffle the whole data between cluster nodes (see appendix A.2.i.c for code).

ii. Seed Propagation Phase

When all the nodes has been added to the propagation tree T, and there is no more nodes for further processing in G, the *seed propagation phase* starts.

Initially, the data is an RDD with a collection of entries made by the union operation between the propagationTree RDD and the updates generated after each iteration in the *seed identification phase*. Each entry is a node connected to its root ($v_{max} \rightarrow u$) (section 5.4.c). First, we need to prepare the data for processing by applying a reduce operation using the *reducePrepareDataForPropagation* function that will aggregate for each root node all its child nodes. Afterwards, we run an iterative MapReduce job to propagate the component identifier from each node to all its children. This operation runs on two levels:

Local Seed Propagation, using the *mapPartition* operator, the root nodes will send its component identifier to the child nodes on the same partition by applying the map function *mapPropagate,* and then the output will be grouped and reduced inside the same partition. Each partition is considered as an independent part of the graph, and the function will try to identify the connected components in this partial graph. (appendix A.2.ii (a) for code).

- *Cluster Seed Propagation,* a MapReduce job runs on all the data in the cluster, in which update are generated from the root to the child nodes, and each node notifies its child nodes with its component identifier, which is the id of its root node. The map function *mapPropagate* will generate update from the root node to its child node, and the reduce function *reducePropagate* will update the component identifier for each node (see appendix A.2.ii (b) for code).

6.2.3 Post-Processing Stage

This stage is only to verify the work and show the results to check their correctness. We do not include its runtime in the evaluations.

The input of this stage is the updated *propagationTreeRDD*, where each node is rooted to the max components identifier (*VertexID→CompID*)

The input for this stage is the output graph from the *computing phase*. Where in the VertexRDD, each node and its components identifier are in the form *<VertexID,CompID>,* (CompID refers to the ID of the connected components they belong to).

To show the largest 10 components, we run a basic MapReduce job, which is the same as the word count example (explained in 2.1.7):

- In the map phase, for each node generate an output to the component identifier

  <VertexID, CompID > => < CompID, 1>.

- In the reduce phase we sum all the (1)s for each CompID.

Then we use *.sortBy()* to sort the output, and *.take(10)* to only take the first 10, and print the largest 10 components with number of nodes in each one (see appendix A.3 for code). All the information about the number of components and size for each one are known for each dataset used. Therefore, this step will help in verifying the accuracy of the algorithm on dataset when results are compared with previously known information about the dataset.

Chapter 7: Experimental Evaluation & Results

To validate and fairly evaluate the proposed enhancements suggested in our approach with respect to other existing approaches we applied the same methodology and used matching technologies running in an identical environment.

Spark was built using the Scala programming language. It supports many languages to enable parallel applications to be developed and interactively used. However, its graph processing library, GraphX, only uses the Scala language. Therefore, all the coding for developing the proposed approach and the implementation of other approaches was carried out using the Scala language on Spark. All our tests used the same installation of Spark with the same configuration and the same computational resources. Details about the datasets used and the experimental setup are presented in the following sections.

7.1 Dataset description

To test the performance of proposed approach different types and sizes of datasets were selected, in order to generalise the result of the evaluation in the experiment. All tests ran on two categories of graphs:

- Real-world Datasets: a collection of commonly used datasets for large graph testing such as.
    - Web-google: It was released in 2002 by Google, with edges represented by hyperlinks between pages.
    - Patent citation: The graph includes all citations made by patents over 37 years.

- KGS: It is a real-time server that enables two online players to simultaneously play against each other in real time. The graph represents edges between players.
- Dota-league: The graph represents friendship between players in the DotA gaming platform in Europe.
- LiveJournal: It is a free online community. Members maintain journals, individual and group blogs, and friendship between each other.

- Synthetic datasets (Graph500 graphs[24]). A collection of benchmark graphs which focus on graph analysis. They were developed t to help evaluate systems for data intensive applications[125]. Graph500 graphs are generated using the Kronecker generator, which will produce power-law graphs (also called scale-free graphs) and are similar to Recursive MATrix (R-MAT) scale-free graphs [126].

	Dataset name	**# of nodes**	**# of edges**
Real-World Datasets	Google web	875713	5105039
	Patent citation	3774768	16518948
	KGS	832247	17891698
	Dota-league	61170	50870313
	LiveJournal	3997962	34681189
Synthetic Datasets	graph500-22	2396657	64155735
	graph500-23	4610222	129333677
	graph500-24	8870942	260379520
	graph500-25	17062472	523602831
	graph500-26	32804978	1051922853

Table 7-1: Datasets used in the evaluation

All the selected datasets contain a large connected component that includes most nodes or in some all the nodes. They are publicly available to give a fair opportunity for re-evaluation

[24] http://graph500.org/

of this work[25] [26]. Datasets used are shown in table 7.1 with information about the number of nodes and edges in each one.

7.2 Experimental Setup:

For testing jobs, we ran all our experiments on a shared production cluster using spot Amazon EC2 (Amazon Elastic Compute Cloud) instances. The driver node (Type: r4.4xlarge) has 16 Cores CPU and 122 GB of memory, each of the 8 worker nodes (Type: r4.2xlarge) has 8 Cores CPU and 61 GB of memory. All nodes run Spark version 2.1 (built with Scala 2.10).

Spark 2.1 (Scala 2.10)	EC2 instance	Nodes	Processor	Memory
Driver:	r4.2xlarge	8	8CPU	61G
Workers:	r4.4xlarge	1	16CPU	122G

Table 7-2: Amazon EC2 instances used for the cluster in the evaluation

For cluster management, we used the Databricks Cloud Platform powered by Apache Spark[27], which makes it straightforward to manage big data complex infrastructures, systems and tools. Databricks also provides its collaborative workspace Notebooks to run interactive queries with Spark-powered dashboards. We used Notebooks to run our Scala code and publish the Notebook for public availability (appendix A). All our datasets were hosted on Amazon Simple Storage Service (Amazon S3) and connected to the Databricks platform to allow data transfer to EC2 instances for processing.

It should be noted that, Cloud Computing Environments such as the Amazon EC2, have some overheads posed on communication and computation because of virtualization on

[25] https://atlarge.ewi.tudelft.nl/graphalytics/#
[26] https://snap.stanford.edu/data/index.html
[27] https://databricks.com/unified-analytics-platform

Amazon's instances. These affect the performance evaluation[127]. Furthermore, using the Databricks services could also introduce some extra overheads with their configuration of the cluster and the cluster load at testing time. However, due to the limited resources and non-availability of an in-house cluster to evaluate our work in an optimal configurable environment, we used all services previously mentioned. To overcome any unexpected lags in performance, we ran all our tests three times on the cluster and report the median value of the results after removing any anomalies.

7.3 Measuring Metrics:

To evaluate our approach in comparison with other algorithms, several metric measures can be used to indicate the algorithm's performance. These include running time, communication cost, number of iterations, evolution of the graph size, scalability, sensitivity to diameter, memory usage, and other resource usage. However, in this study, we only consider the following three measures:

i. Running time

The performance of algorithms could be indicated by reporting the running time to show the improved processing speed of each algorithm compared to others. In this study, for running times we reported only the actual times by omitting the time needed for transferring data to the cluster. We also omitted pre-processing stage timings, which include building and preparing the graph

ii. Evolution of the graph size

This indicates how the structure of graph is changing and whether it is growing or shrinking. Many algorithms change the graph structure during the process, as in the graph contraction scheme, and this change could have a significant impact on how the algorithm performs.

Several algorithms try to reduce to size of the graph after each iteration by removing or deactivating nodes. This can help to reduce the amount of computation, the cost of memory access and disk I/O. This could also reduce the amount of network transferred information due to fewer messages being generated.

iii. Number of Iterations

The total number of iterations the algorithm takes until it finishes, could be a useful indicator of how fast the algorithm achieves convergence. However, it is not a precise measure of how well the algorithm performs.

7.4 Testing & Results

This section reports the results following testing of each of our algorithm extensions. Each extension is tested in one experiment and its results are compared with the results from the original implementation without the extension. Table 7-3 shows the extensions evaluated, the algorithms used in the evaluation, which part of the algorithm was measured, and the criteria measures used in the evaluation.

Extension Evaluated	Algorithm	Algorithm part evaluated			Criteria measured		
					Runtime	**Number of Iterations**	**Evolution of graph Size**
Degree approach	Pregel-Original	All			Yes	Yes	No
	Pregel-Degree	All			Yes	Yes	No
	Alternating -Original	All			Yes	Yes	No
	Alternating - Degree	All			Yes	Yes	No
	Cracker-Original	Min-Selection step	Pruning step	Propagation step	Yes	Yes	Yes
	Cracker-Degree	Max Identification step	Pruning step	Propagation step	Yes	Yes	Yes
Local-Max Identification	Cracker-Degree	Seed Identification phase			Yes	Yes	Yes
	Cracker-Degree-Opt				Yes	Yes	Yes
Local-Seed propagation	Cracker-Degree	Propagation phase			Yes		
	Cracker-Degree-Opt				Yes		
DS-Pruning	Cracker-Degree	All			Yes		
	Cracker-Degree-DS				Yes		

Table 7-3: Evaluation Table

7.4.1 Effect of using the Degree Approach to find connected components

Here, we evaluate the approach of using the vertex degree in finding connected components. Usually, algorithms used for that purpose, selects the identifier for each component based on the lexical ordering of nodes and ignores the existing graph's structure. However, in this experiment, both the general approach of selecting the component identifier based on the minimum id node, and the approach of selecting the identifier based on the degree of node. Where node that has the highest degree is selected, and in case more than one node share the same highest degree, the node with the maximum ID is selected.

i. Performance of the Degree Approach in BSP Paradigm

In the first experiment, we compared the performance of two implementations of Pregel to find connected components in the datasets presented in Table 7-1. Pregel, described previously in the literature review, is considered a more efficient framework for distributed graph processing than MapReduce. However, it does not provide fault tolerance and does not perform well for very large datasets with a skewed degree distribution[14].

The first implementation used the traditional way that identified the node with the minimum ID. The second is our approach of using the node degree. Our results are presented in figure 7-1 for runtime and figure 7-2 for number of iterations until convergence.

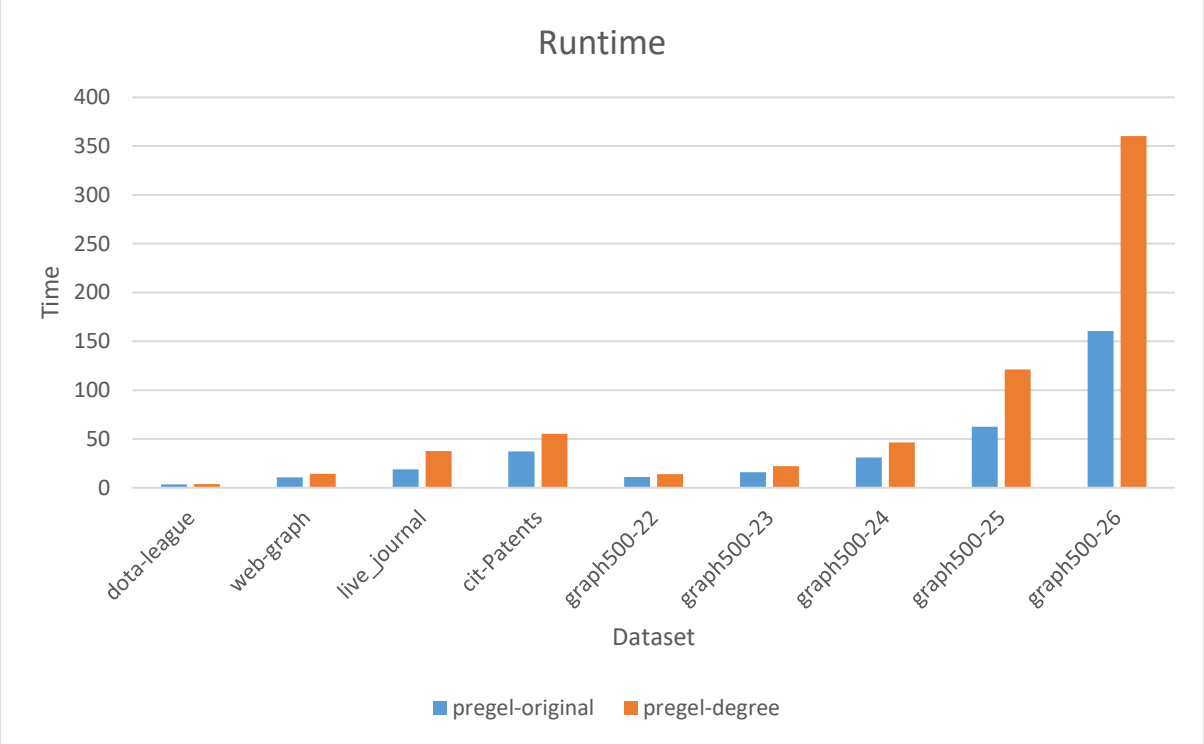


Figure 7-1: Run-Time for Pregel-Original vs Pregel-Degree

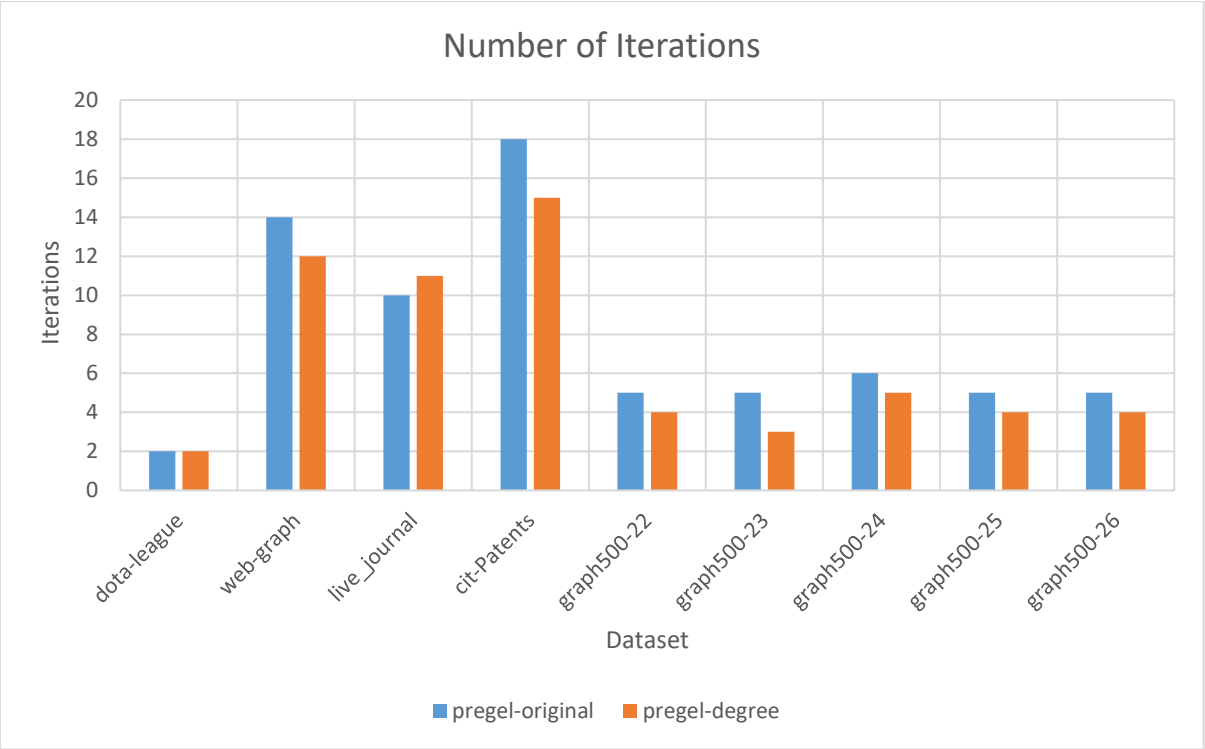


Figure 7-2: Number of Iterations for Pregel-Original vs Pregel-Degree

These results show an increase in runtime for the algorithm to finish. This because of the increase in message size between nodes and the need for more computation to choose component identifiers in each iteration. However, we also noticed a decrease in the number of iterations needed to converge in most tests.

Figure 7.3 shows that, in the initial few iterations, processing takes more time when using the degree approach compared to the original approach. However, this margin rapidly decreases and in some cases, runtime is less than the original approach for the corresponding iteration, and in almost every case there are fewer iterations.

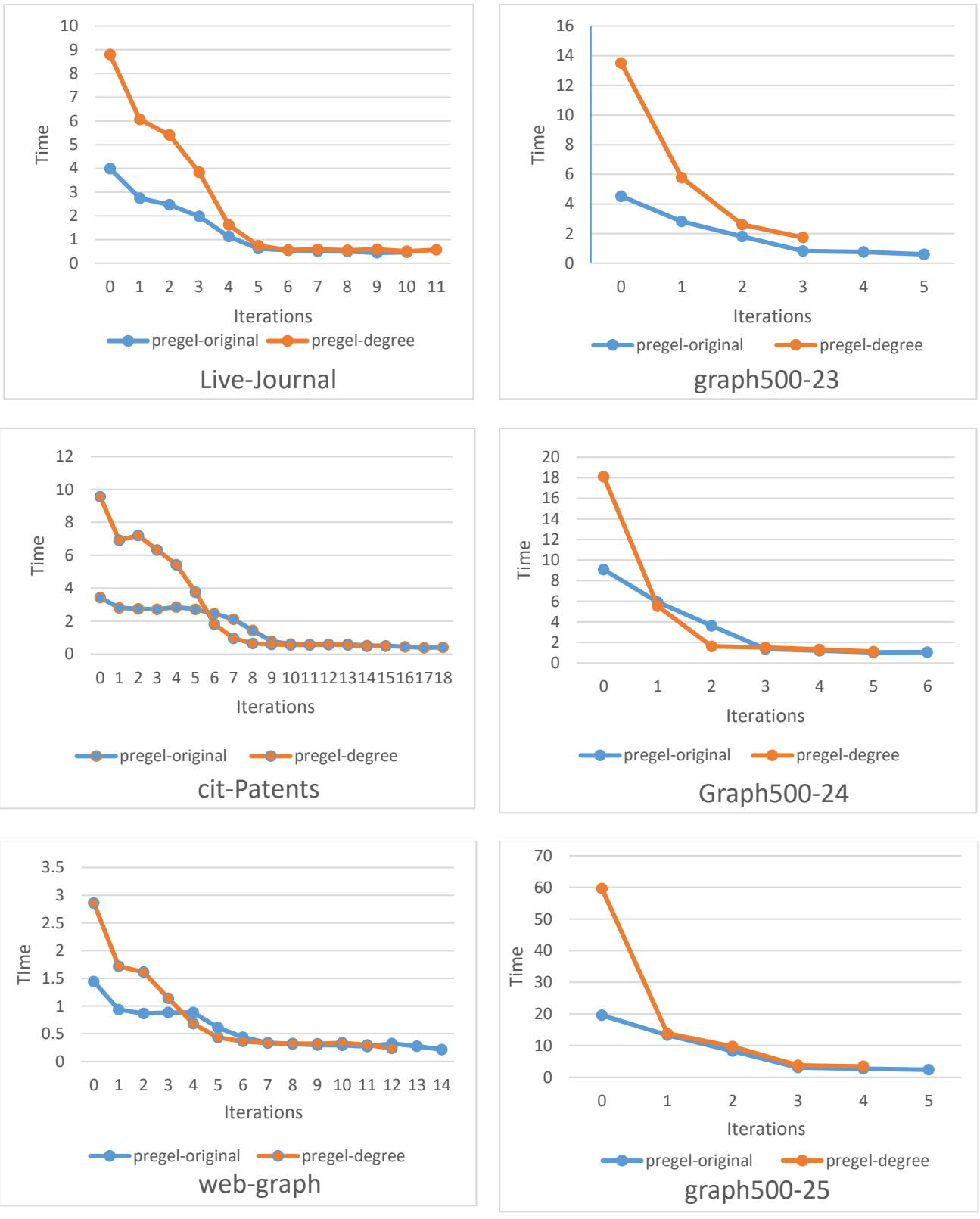


Figure 7-3: Iteration vs Reducer Time for Pregel-Original vs Pregel-Degree

ii. Performance of the Degree Approach in Alternating algorithm

The second experiment is very similar to the previous one. However, here we compare the performance of two implementations of the Alternating algorithm[14] to finding connected components in some of the datasets presented in Table 7.1. The first implementation is based on the original algorithm using the minimum ID as the component identifier. The second implementation uses the node degree approach. The results presented in Figure 7-4 and 7-5 show similar performance, where the runtime increases when the degree is used, and the number of iterations decreases in most of cases.

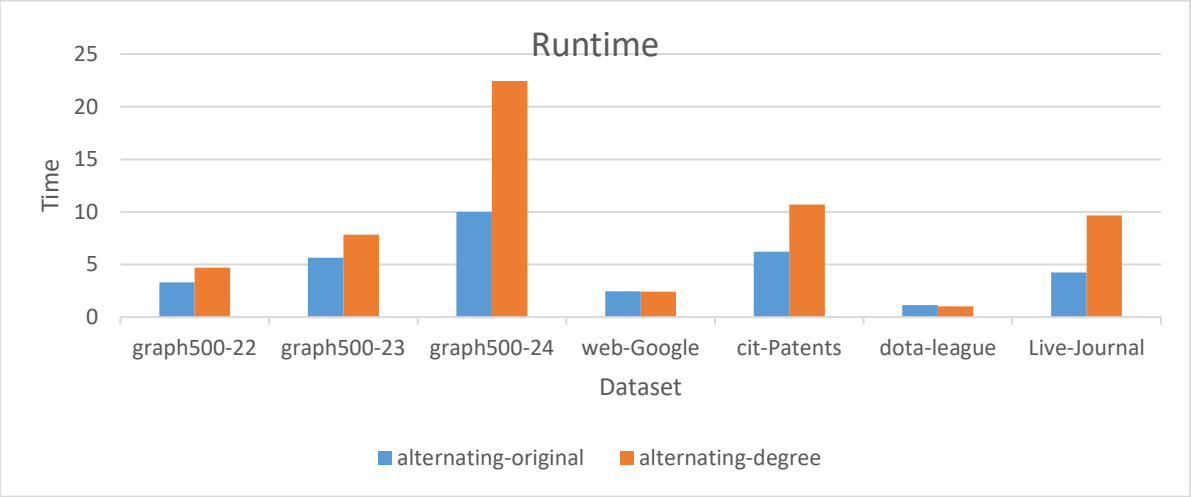


Figure 7-4: Run-Time for Alternating-Original vs Alternating -Degree

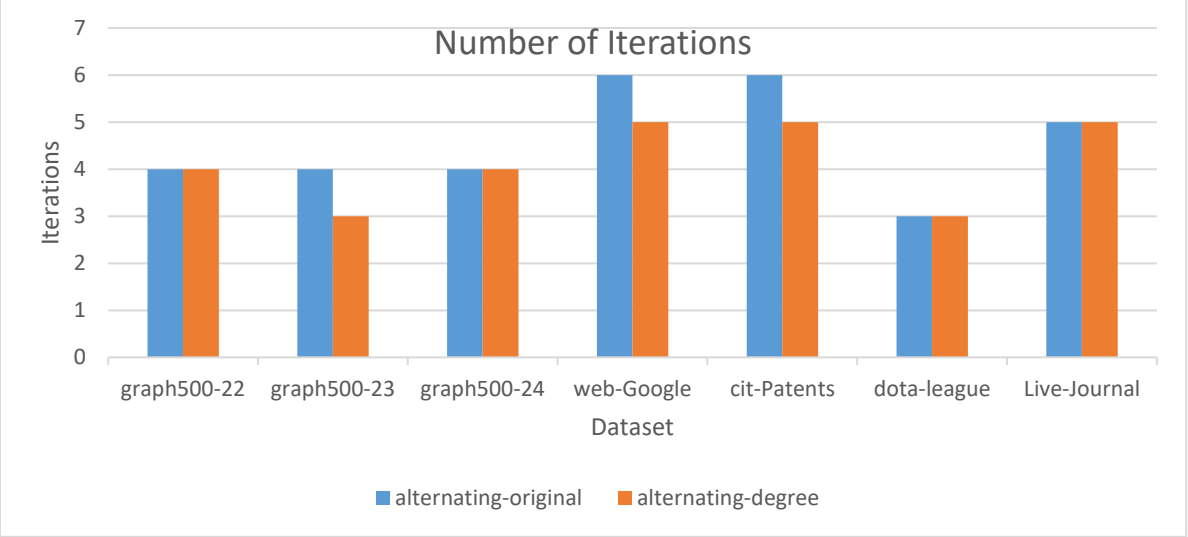


Figure 7-5: Number of Iterations for Alternating-Original vs alternating -Degree

iii. Performance of the Degree Approach in Cracker algorithm

In this experiment, our objective was to determine the node degree approach for finding connected components on the performance of the Cracker algorithm, which was explained in section 4.7.8. According to Lulli[115] the Cracker algorithm outperforms its competitor algorithms for finding connected components in large graphs in terms of both time and volume of messages between iterations.

The first implementation used the original source code[28] provided by the developer of the algorithm. In the second implementation, we integrated the degree in the Cracker algorithm and modified it to choose the component identifier based on node degree instead of the node ID. Our source code is shown in Appendix A. We did not introduce any optimisation to the modified algorithm other than using the degree.

Figure 7.6 presents the results for the runtime taken by both the original Cracker algorithm and our modified version, which we call Cracker-Degree to indicate our implementation using the degree approach discussed previously. In figure 7-7, runtime is shown separately for each of the main three steps in the Cracker algorithm to evaluate the effect of our modifications in each step.

The graph shows that there has been an insignificant difference with some slight increase for runtime in some real-world datasets. This can be result from the extra node degree information included in the messages transferred between cluster nodes. However, for the synthetic datasets (Graph500~) the graphs show a marked decrease in runtime of almost every step when using the degree approach. This could be explained from Tables 7-4 and 7-5 that show the number of active nodes after each iteration. Table 7-4 indicates that there has been a sharp drop in the number of active nodes after the first iteration in synthetic

[28] https://github.com/hpclab/cracker

datasets. This results from using the degree approach in identifying non-seed nodes in the seed identification step and excluding them for the computation process in later iterations. Figures 7-8 and 7-9 show the results where more non-seed nodes identified in Cracker-Degree were compared to the original algorithm. This also helped to reduce the number of iterations. Therefore, it appears that using the degree approach results in faster convergence and can lead to a significant performance improvement.

	Graph500-22		**Graph500-23**		**Graph500-24**		**Graph500-25**	
Iter	*Cracker_Org*	*Cracker_Degree*	*Cracker_Org*	*Cracker_Degree*	*Cracker_Org*	*Cracker_Degree*	*Cracker_Org*	*Cracker_Degree*
1	2396657	2396657	4610222	4610222	8870942	8870942	17062472	17062472
2	748509	220282	1440366	412988	2797920	777933	5412675	1457768
3	878	167	1443	252	2661	427	4670	786
4	2	0	2	0	2	2	3	0
5	0		0		0	0	0	

Table 7-4: The number of active nodes at each iteration for synthetic datasets

Iter	**dota-league**		**kgs**		**cit-Patents**		**web-Google**		**soc-LiveJournal**	
1	61170	61170	832247	832247	3774768	3774768	875713	875713	4846609	4847571
2	4302	1945	146900	136740	2023363	940837	280757	107648	1228095	1125190
3	0	0	3561	1920	229472	90203	17305	8237	79398	61598
4			50	41	11850	5330	1010	472	3903	1825
5			2	0	395	230	91	70	214	29
6			0		18	4	6	2	9	2
7					0	0	0	0	0	0

Table 7-5: The number of active nodes at each iteration for real-world datasets

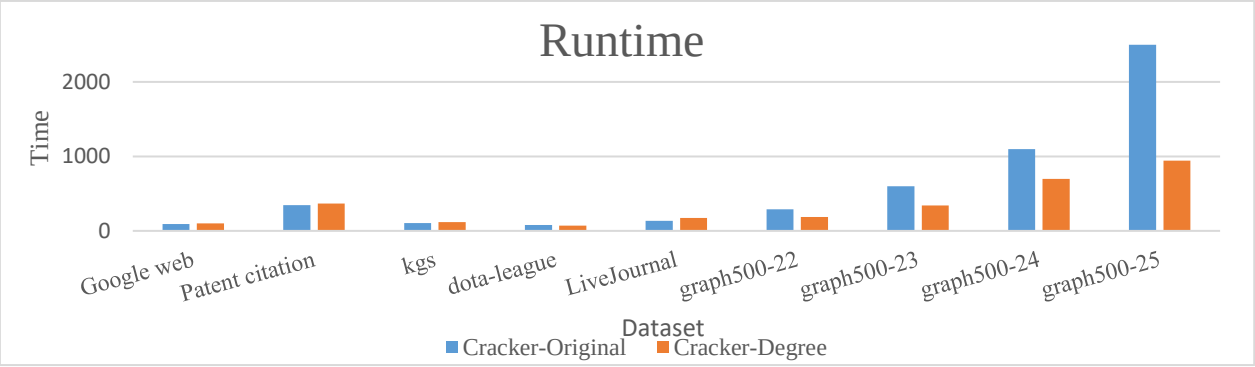


Figure 7-6: Runtime for the Cracker-Original and Cracker-Degree

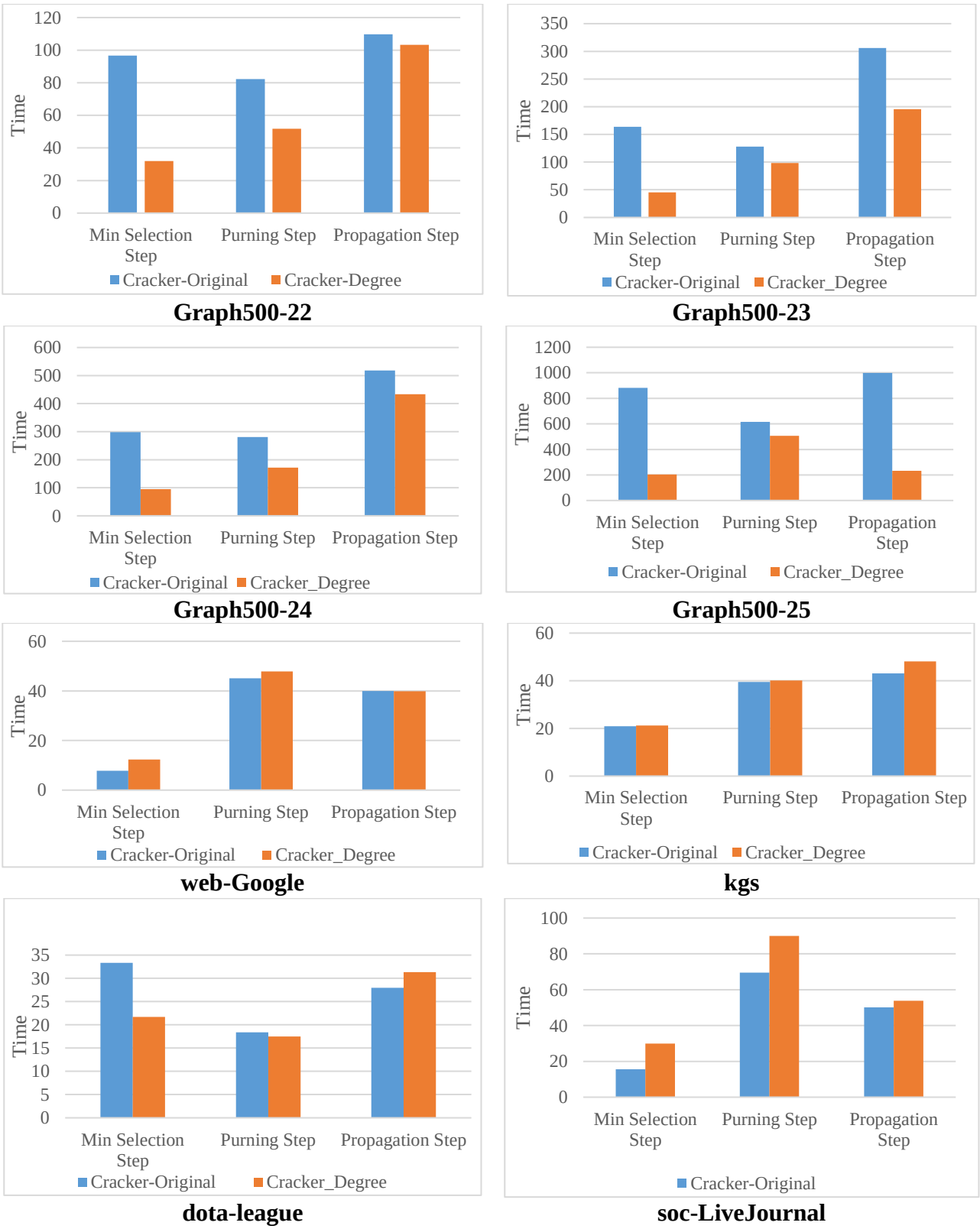


Figure 7-7: Runtime for the Cracker-Original and Cracker-Degree at each step

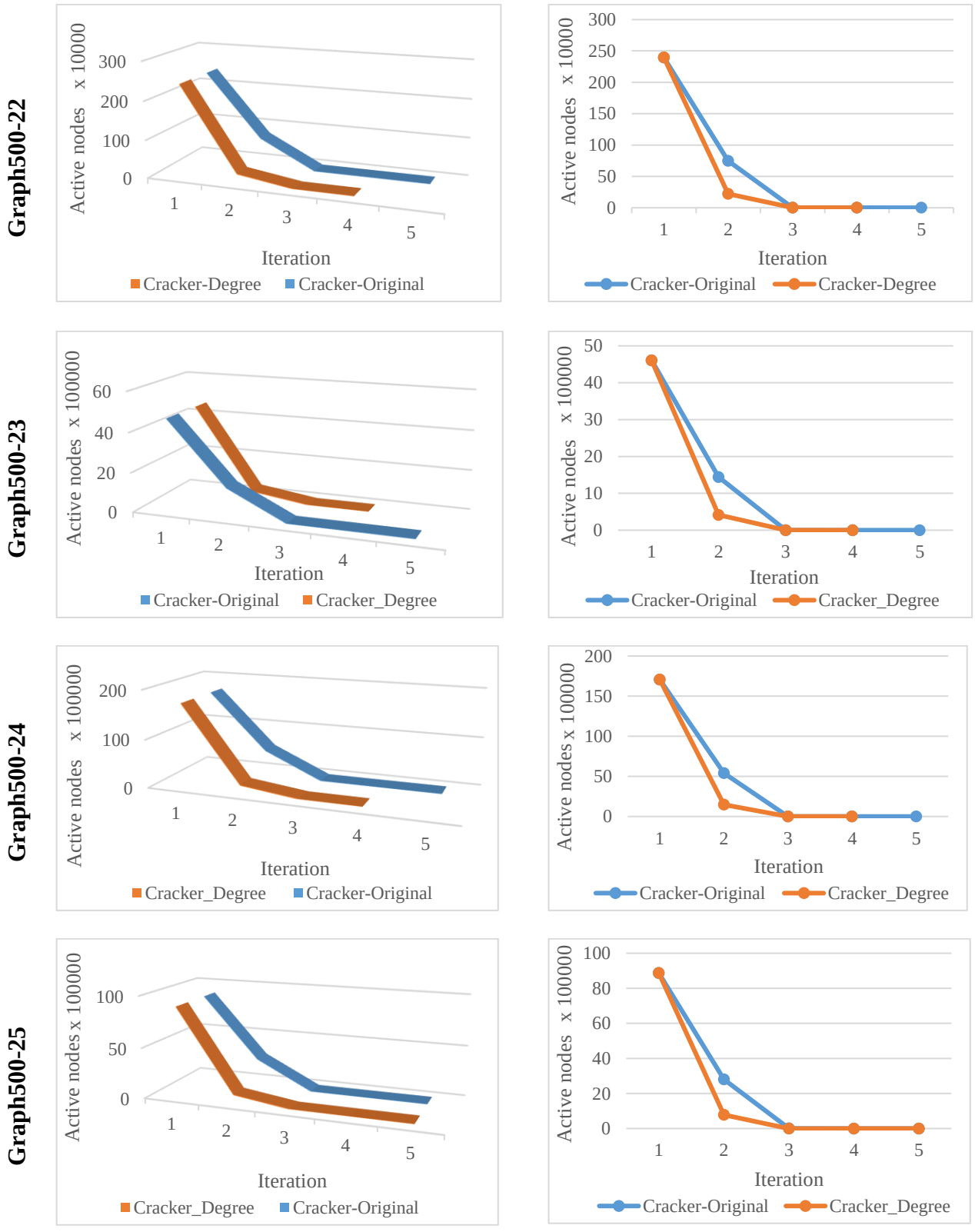


Figure 7-8: The number of active nodes at each iteration

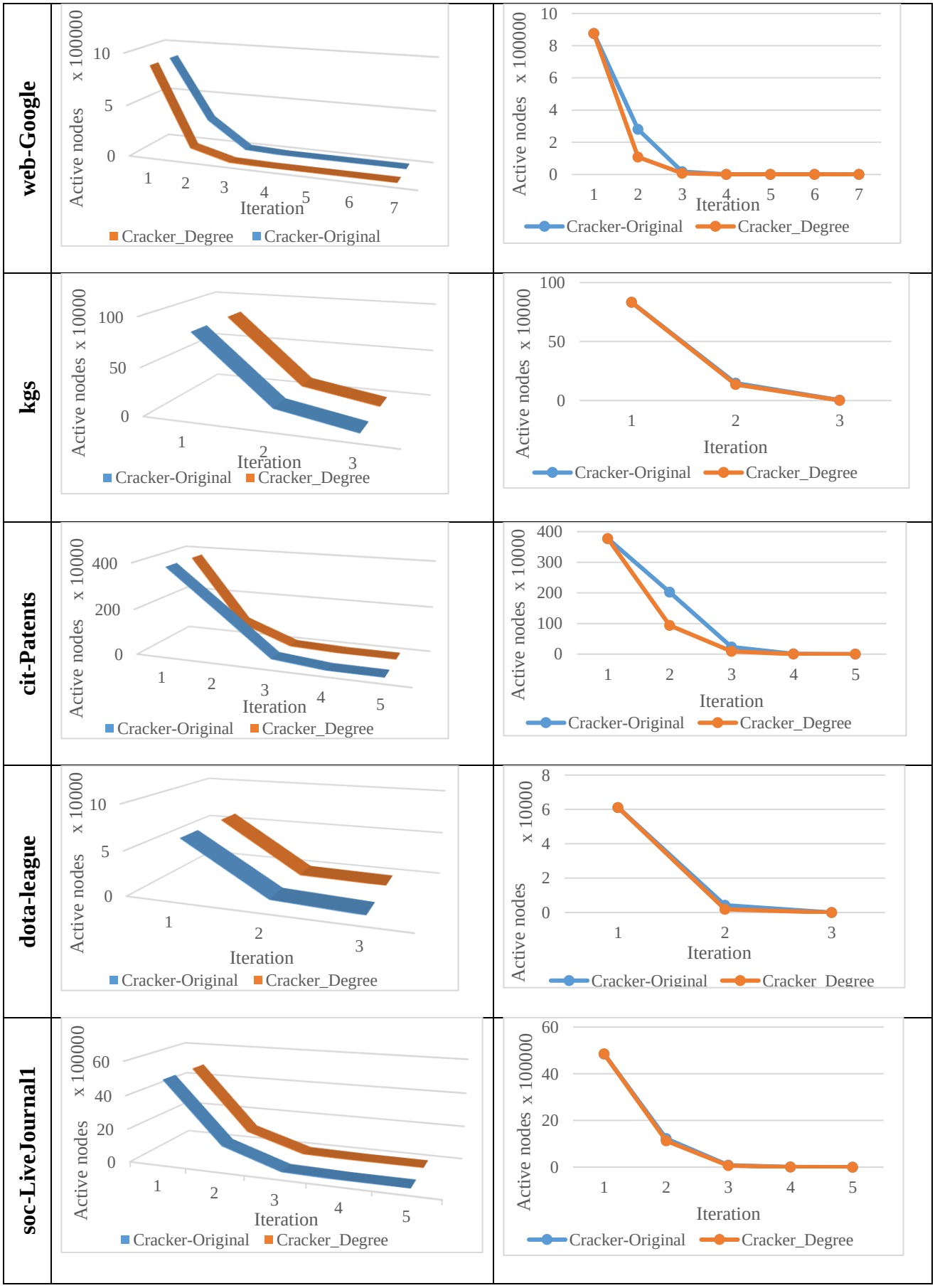


Figure 7-9: The number of active nodes at each iteration

7.4.2 Effect of local Max Identification

In the Seed Identification phase, the main objective is to exclude non-seed nodes from the computation and add them to the propagation tree. In the original Cracker algorithm, it is implemented in two steps, max identification and pruning step. The first step will identify those nodes which will be excluded in the pruning step. However, we try to break the first step in two further steps: local max identification and cluster max identification. In the local max identification step, which we described in section 5.5.1, we process each partition separately to find the local component identifier for all nodes in that partition and for each node update its component identifier accordingly. A disjoint data structure is used to implement this step as described in section 5.5.1. The results from the experiment of evaluating the efficiency of this step are presented in Table 7-6. The number of active nodes after each iteration is reported for both the modified implementation (where the local max identification is added), and the original algorithm implementation (where only the cluster max identification is performed).

Iter	cit-Patents		kgs		dota-league	
	Seed-Iden-Org	Seed-Iden- Opt	Seed-Iden- Org	Seed-Iden- Opt	Seed-Iden- Org	Seed-Iden- Opt
1	2396657	2396657	832247	832247	61170	61170
2	220282	17757	136740	15062	1945	0
3	2	14	1920	52	0	
4	0	0	41	3		
5			0	2		
6				0		

Iter	Graph500-22		Graph500-23		Graph500-24		Graph500-25	
	Seed-Iden-Org	Seed-Iden-Opt	Seed-Iden-Org	Seed-Iden-Opt	Seed-Iden-Org	Seed-Iden-Opt	Seed-Iden-Org	Seed-Iden-Opt
1	2396657	2396657	4610222	4610222	8870942	8870942	17062472	17062472
2	220282	17757	412988	61811	777933	202671	1457768	610993
3	2	14	3	24	427	61	786	120
4	0	0	0	0	2	0	0	0
5					0			

Table 7-6: The number of active nodes after each iteration of seed identification phase. Optimised Seed ident (Seed-Iden-Opt) vs Original Seed ident (Seed-Iden- Org)

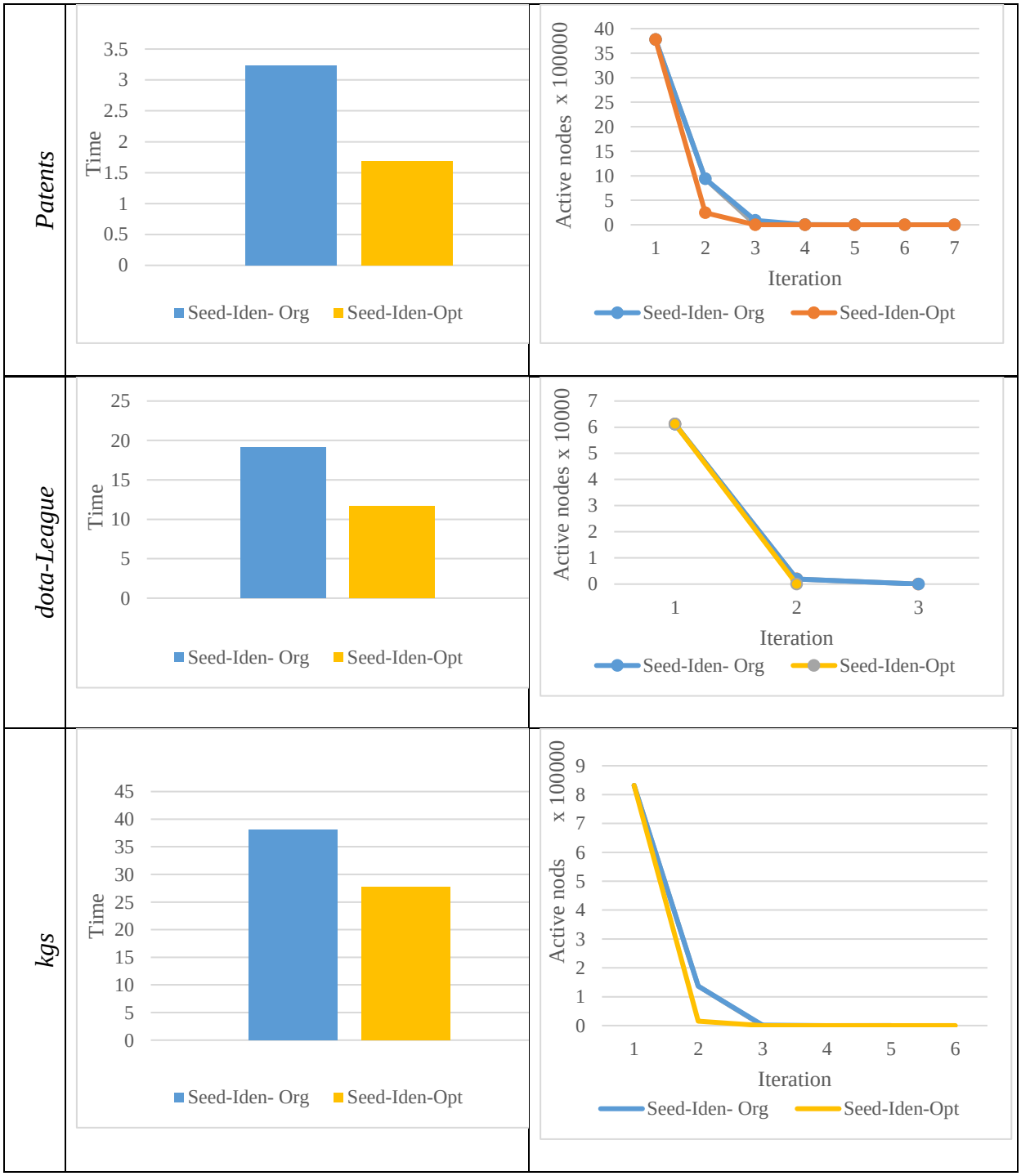


Figure 7-10: Runtime for the Seed Identification Phase

Optimised Seed ident (Seed-Iden-Opt) vs Original Seed ident (Seed-Iden- Org)

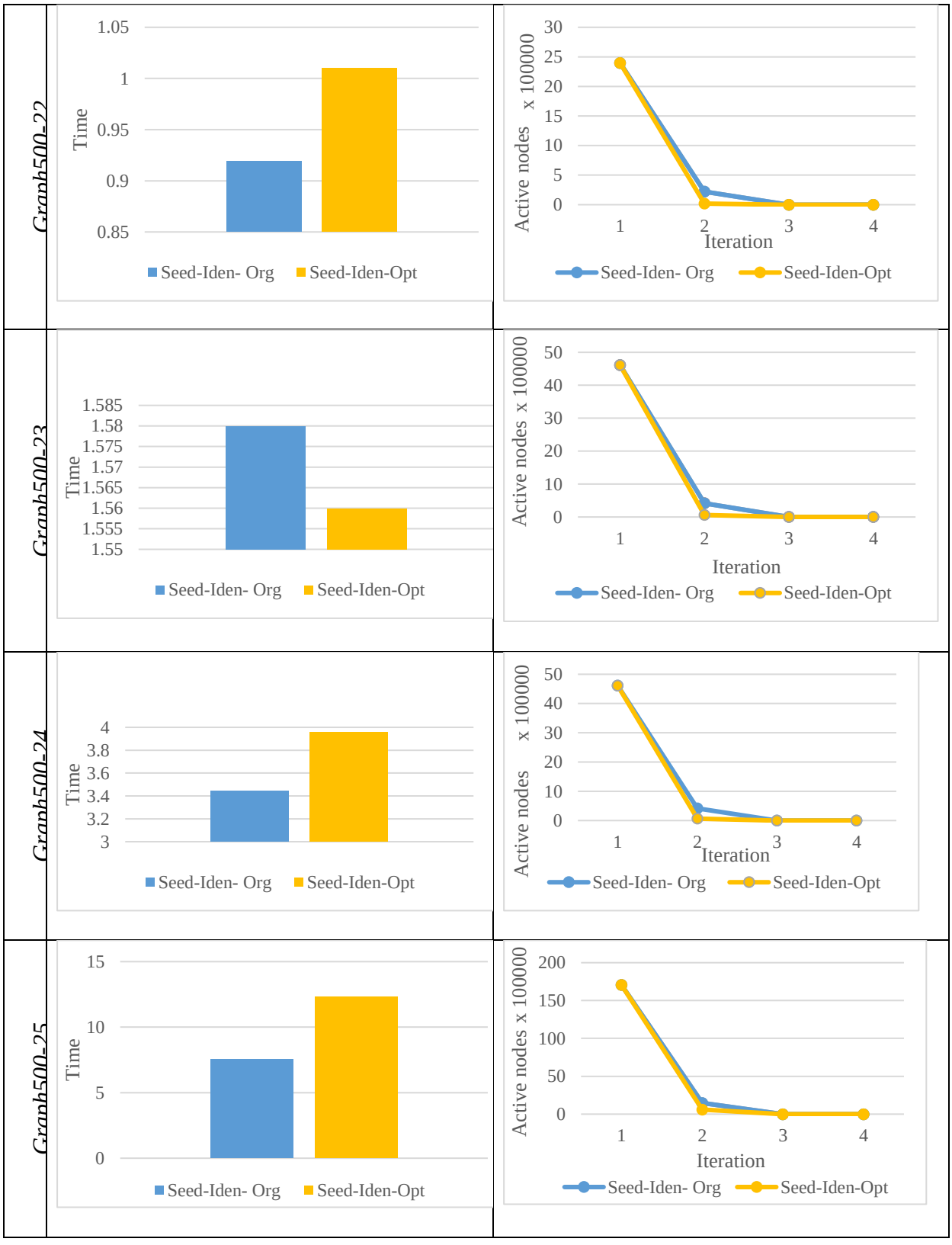


Figure 7-11: Runtime for the Seed Identification Phase

Optimised Seed ident (Seed-Iden-Opt) vs Original Seed ident (Seed-Iden- Org)

The results of this experiment show a further drop in the number of active nodes and the number of iterations when local max identification is used. These results agree with and support the objectives introduced from the implementation of this step. However, contrary to expectations, the runtime results shown in figures 7-10, 7-11, and 7-12 indicate some inconsistencies. Although for some real-world datasets the runtime was promising as it decreased, for the synthetic datasets the runtime increased. This inconsistency may be due to the difference in the type of graph structure, as real-world graphs tend to be more sparse. Furthermore, not all tests on real-world datasets have finished in our implementation, as in case of the web-google and soc-liveJournal datasets. This indicates the need for further optimisation of our implementation. In addition, further research should be undertaken to investigate the cause of some contradicting results.

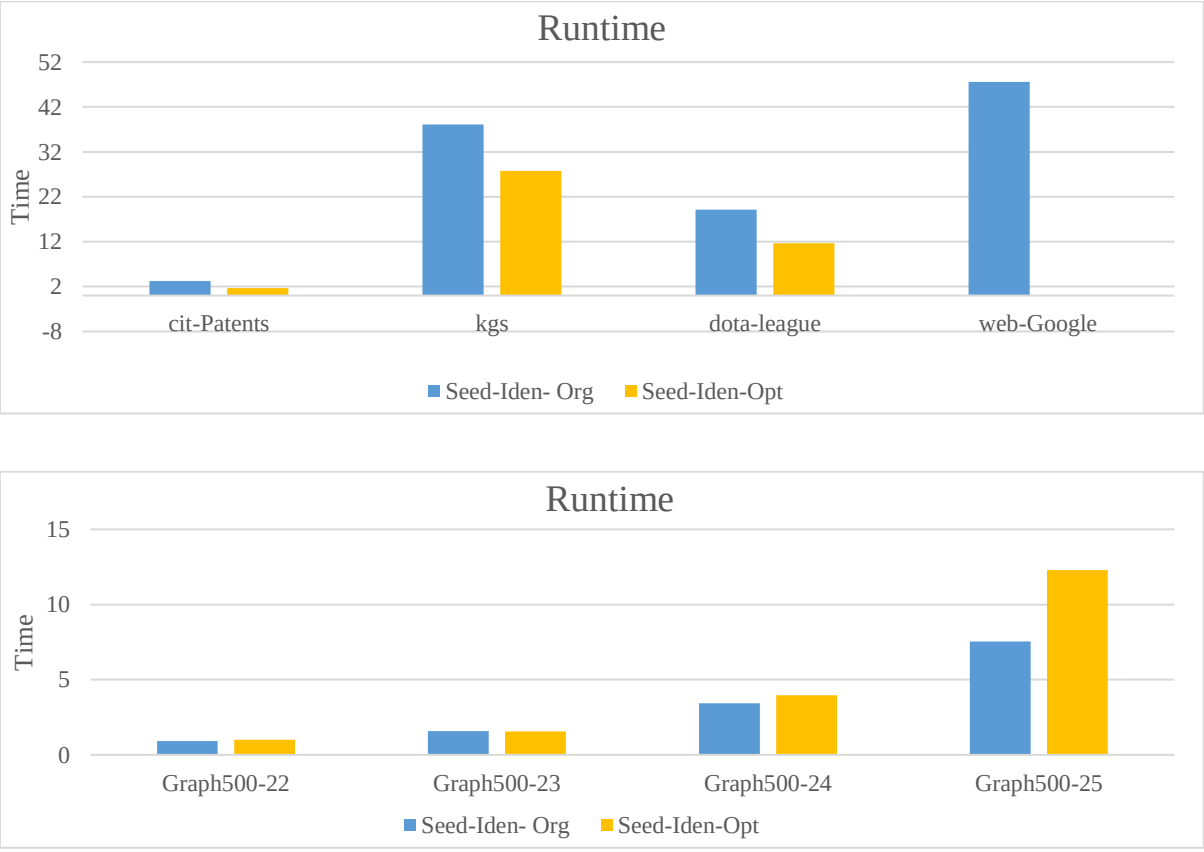


Figure 7-12: Runtime for the Seed Identification Phase on different datasets Optimised Seed ident (Seed-Iden-Opt) vs Original Seed ident (Seed-Iden- Org)

7.4.3 Effect of local Seed Propagation

Nodes are added to the propagation tree T in the Seed Identification phase, where each node is rooted to its local component identifier. In the Seed Propagation Phase, each node then propagates its component identifier to its children until all nodes belonging to the same component are propagated with the same identifier. To enhance this operation a Local Seed Propagation Step was proposed as described in section 5.5.2.

In the experiment presented in figure 7-13, the runtime results for the *Seed Propagation Phase*, for both the enhanced implementation (where the local seed propagation is added), and the original algorithm implementation (where only the cluster seed propagation is performed) are shown. For the real-world datasets, no significant differences are noted. However, when the size of the synthetic dataset increase there is a marked decrease in the runtime.

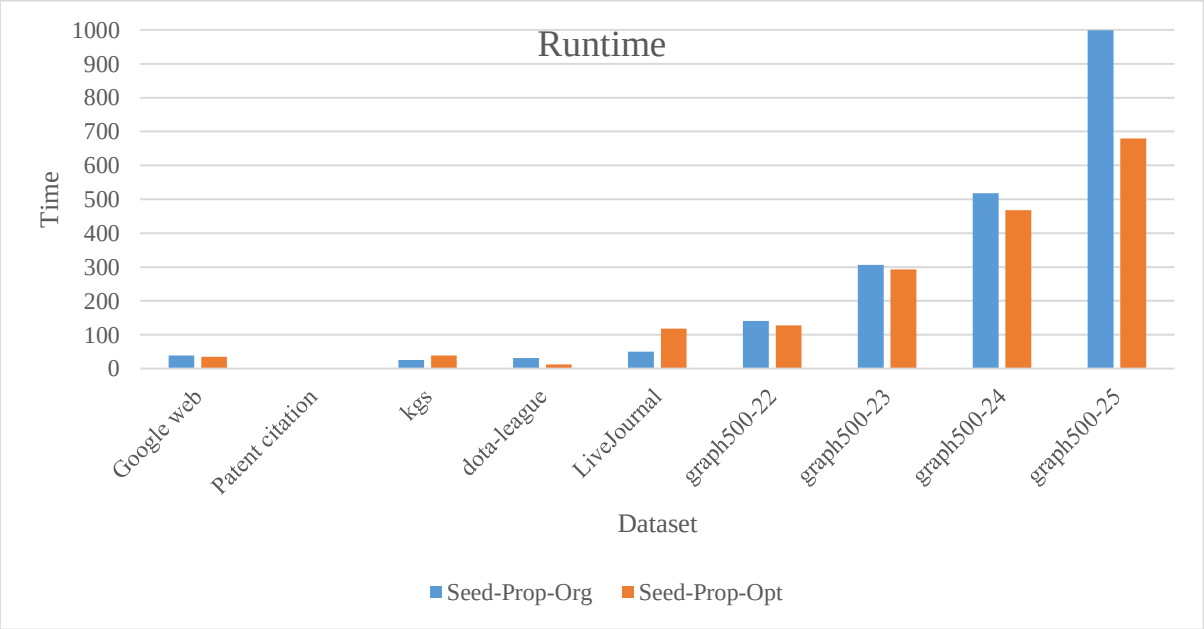


Figure 7-13: Runtime for the Seed Propagation Phase on different datasets Optimised Seed Propagation (Seed-Prop-Opt) vs Original Seed Propagation (Seed-Prop-Org)

The results of this experiment indicate that there is an interesting runtime improvement on the synthetic datasets tests, where the runtime improves as the size of the tested dataset becomes larger.

7.4.4 Performance of the DS-Pruning

In this experiment, we tested the implementation of building the propagation tree and caching it in the memory of the driver node in the cluster. Updates at the end of each iteration of the Seed Identification Phase are gathered and merged into the propagation tree dynamically as they are generated using the disjoint-Set data structure described in section 5.4.ii.

Although, Spark does not support such operations where data could be cached only in the driver node memory, we tried implementing this on Spark using its available operators. we collected the propagation updates and forced a dynamic merge operation between the updates and the propagation tree. However, we couldn't cache the propagation tree in the memory of the driver node, which caused to some overhead on the system (our code is shown in Appendix A).

Results are presented in figure 7-14. These results were not very encouraging as the runtime increased significantly compared to the original algorithm implementation. This is most likely caused by the unexpected issue of the vastly increased amount of data shuffling. Nonetheless, we managed to get accurate results and the algorithm coverage as expected.

In the results of this experiment, only the runtime has been reported because the number of iterations and the size of graph evolution are the same in both the original and optimised algorithms. Nonetheless, the need to proceed into the *Seed Propagation Phase* was avoided,

and consequently this could help to improve the performance in case the issue of the enormous amount of data shuffling was resolved. There is significant room for further progress if this could be optimised in Spark, or implemented using a different distributed graph processing framework.

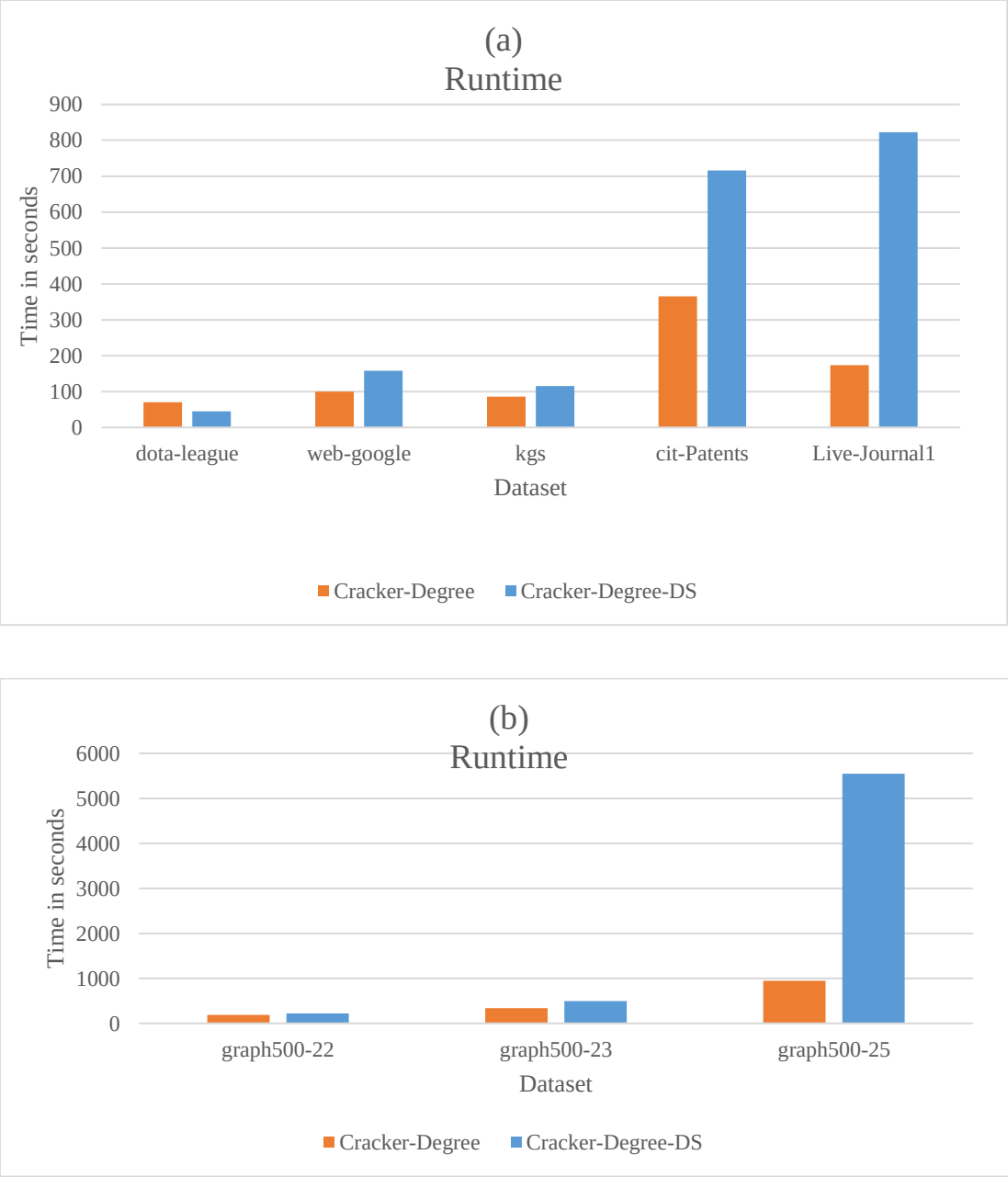


Figure 7-14: Runtime for the seed propagation (a) & (b) Cracker-Degree vs Cracker-Degree using disjoint-set for Propagation (Cracker-Degree-DS)

7.5 Summary

The results in this chapter with respect to the proposed improvements indicate the following:

(i). **Using the node degree approach**:

In finding connected components algorithms using the node-degree for choosing the component identifier can significantly affect the performance of the algorithm.

- In all case, it helps to reach converge faster by reducing the number of iterations. However, it increases the size of the graph as the node degree is attached to each node, in addition this increases the communication load due to the increased size in messages. Therefore, this approach can increase the runtime of the algorithm in spite of decreasing the number of iterations needed.
- For algorithms which provide *vertex pruning***,** where the algorithm has the ability of excluding unusual vertices for the computation from the process, which shrink the size of the graph after each iteration. In this case, using the node degree approach appears to be very efficient and results in faster convergence which lead to a significant performance improvement. Both the runtime and number of iteration has decreased, especially after the first iteration where there has been a sharp drop in the number of active nodes in compare to the approach where the node ID is used.

(ii). **Using the local computation for connected components approach:** by moving more computation toward where the data is stored on the worker nodes and finding connected component in the data partition

- In case of local max identification

- The runtime on the synthetic dataset has increased, while for the real-world dataset runtime decreased with more effect on reducing the number of node after each iteration when this approach is used.
- This inconsistency may be due to the difference in the type of graph structure, as real-world graphs tend to be more sparse. Also, the increase of the runtime could be explained due to the overhead caused by the computation for each partition is done by only CPU core (spark configuration).

- In case of local propagation
  - The runtime for the real-world datasets showed no significant differences are noted. However, when the size of the synthetic dataset increases there is a marked decrease in the runtime, which indicates that, when the graph size increases, and consequently the number of partitions increases, performance of local seed identification becomes more efficient.
- Performance of the DS-Pruning

  These results were not very encouraging as the runtime significantly increased most likely due to the vastly increased amount of data shuffling. However, final results were accurate and the algorithm coverage and the need to proceed into the *Seed Propagation Phase* was avoided, which could consequently improve the performance in case the issue of the enormous amount of data shuffling could be resolved.

In summary, it has been shown from the results that these experiments have confirmed that using the degree approach resulted in faster convergence and can lead to significant performance improvement. In many cases, optimising the design of the algorithm with local pre-processing of the data using processing system features can also result in performance enhancement.

Chapter 8: Conclusions

8.1 Introduction:

The aim of this research has been to examine the processing of large-scale graphs and more specifically, enhance the performance of finding connected components algorithms in large graphs. Finding connected components is an essential pre-processing step to extract knowledge about the graph. It is also a fundamental operation for some graph computations such as pattern recognition, reachability, graph compression, graph partition, and random walk[12]. The MapReduce[18] framework dominates the processing of large-scale data on Hadoop, and it is commonly used for mining big graphs[128]. However, iterative processing is not directly supported in MapReduce. Nonetheless, some recent works[21][22] show that it is possible to outperform other models for finding connected components using MapReduce. Yet, only a few studies have investigated this problem in big data distributed system using MapReduce[14].

Current big data processing systems have become more advanced with features beyond MapReduce, such as Spark[23], which supports iterative processing. In addition, current MapReduce algorithm for finding connected component only use the traditional approach to selecting the component identifier for each component based on the lexical ordering of the node ID value. These issues have been addressed by implementing a new algorithm for finding connected components following best practices and design patterns recommended when using MapReduce paradigm. In the new algorithm, graph structure property is considered. More specifically, the node degree has been used as the main criteria for choosing the component identifier. In addition, features beyond MapReduce provided by the processing system have also been considered, such as the ability to move more computation

toward where the data is stored. For the local computation for the connected components, disjoint-set data structure has been used.

8.2 Summary

This thesis has reviewed current big data processing systems, focusing on large-scale graph processing systems. The study has started by defining what big data is, the technologies used, and how MapReduce operates. It has discussed graphs and big graph processing systems and the approach used in the developments of big graph processing systems with a brief overview of the most common distributed graph programming models. The main focus of this study has been on the algorithm of finding Connected Components in an undirected graph, which is one of the main concepts that have been studied in Graph Theory[6]. Most of the known algorithms for CC in MapReduce have been reviewed in depth, and a few improvements in our approach has been introduced. The improvements implemented have included graph contraction based on node degree, dynamic evaluation of the degree after each iteration, and computing local CC in the map phase based on Disjoint-Set data structures. The study has applied the proposed improvements on the latest algorithm for finding connected components, carrying out extensive experimental evaluations of the implementations using large real world and synthetic graph datasets on computing clusters.

One of the key issues in the design of current CC algorithms for large graphs is that they do not consider the structure of the graph processed. Moreover, they do not benefit from new features available in the current advance distributed processing systems. Instead, they follow the traditional MapReduce programming model using its original Hadoop system implementation. This study has addressed these issues in the proposed approach. It has initially experimentally investigated using the node degree for choosing the component identifier instead of just using the node ID. Furthermore, it has proposed moving more

computation to where the data is stored on the worker nodes and used a disjoint-set data structure to help find connected components locally on one node.

The results of these experiments confirm that using the degree approach resulted in faster convergence and can lead to significant performance improvement. In many cases, optimising the design of the algorithm with local pre-processing of the data can also result in performance enhancement. However, this step should be considered wisely as it could cause some system overhead and a drop-in performance. Many factors can affect this, such as graph size, structure, and density.

8.3 Contributions

This study contributes to the field of knowledge as follows:

(v). **Using the node degree approach in finding connected components algorithm**:

using the degree approach in choosing the connected component identifier will always result in less number of iteration until convergence, however it adds some overload on the system due to the extra work required to calculate the degree for each node and the increased size of messages due to the attachment of the degree to the node. Nonetheless, this approach showed significant performance improvement when applied to algorithms which apply vertex pruning; where unuseful nodes for the computation are excluded from the process after each iteration. In this kind of algorithms (Cracker in our case) the number of iterations decreases and the size of graph shrinks faster when this approach is applied, leading to better runtime.

(vi). **Using the local computation for connected components approach:**

Moving more computation towards where the data is stored, and trying to apply computation on a data partition before the need to do computation on the cluster can

effectively improve the performance of the algorithm. In the case with the Cracker algorithm, despite the inconsistency in results, in general there is a noticeable performance improvement especially in the *seed propagation phase* for the larger datasets. This approach should to be wisely considered and implemented as it could increase the load on the system and lead to performance degradation.

(vii). **Considering different level of computation in the design of the algorithm.**

In big data processing system operations are applied at different level, by identifying the level of processing, and integrating them in the process of the algorithm design can help to increase the efficiency of the algorithm. For example, start by processing the data partition, then process the collective data of partitions inside a cluster worker node, and finally process all the data at the cluster driver node. Customising operation in the algorithm for each level could increase the performance of the algorithm. In this study, processing has been customised and applied on the data partitions in the cluster driver nodes. However, additional operations could be added to process the data inside a cluster worker node using multi-core structure of the cluster nodes.

(viii). **Guidelines to be implemented in different context**

It is worth noting that one of the major contribution of this work is to encourage active researcher in the field to consider features provided by the current new processing systems in the design of their algorithms. This could be considered as useful guidelines to be implemented in different context.

8.4 Limitation

During the process of conducting this research, we faced many issues. The most important limitation being that of resources and the non-availability of a suitably configured in-house processing cluster to evaluate our work in an optimal configurable environment. This study was therefore, carried out using the Amazon Cloud Computing Environment. However, this also had some limitations, it is not obvious wither there is other application running on the same cluster nodes at the same time and what is the impact of their jobs. So not all the allocated resources could be available at that time. In addition, some overheads could impact communication and computation caused by the virtualization used by Amazon, which affects the performance evaluation. During the testing, we encountered inconsistency in readings for the algorithm runtime. We attempted to overcome any unexpected lags in performance by running all our tests three times on the cluster and reporting the median value of the results after removing any anomalies. However, this did not guarantee the right results. We restarted the cluster before each test run to ensure there would be no impact from running the previous tests, as we noticed that Spark keeps some unnecessary RDD files in memory even after asking it to un-cache them.. However, we had no control on communications and other processes that could be running on other shared resources in the virtual system. An additional uncontrolled factor was controlling the configuration of the cluster. We used the Databricks cloud platform to manage the cluster and its Spark-powered dashboards to run our code. This also could introduce some extra overheads with their configuration of the cluster and the cluster load at the testing time. Again, this also limited our ability to change the configuration of cluster, and therefore restricted the options in trying larger datasets and using larger clusters.

An issue that was not addressed in this study was the fast-moving pace of technology in this area, as most of the algorithms and technologies mentioned were created during the process of conducting this study.

8.5 Future Works

Findings from this research offer the following insights and recommendations for future research:

- Future research should concentrate on the investigation of the previous issues addressed in the limitations section 8.4. For example, using a dedicated cluster with a controlled configuration could achieve very reliable results and help to understand them.
- A further study could assess the scalability of the improvements implemented to determine how they perform on larger datasets or on different type of graphs.
- A more straightforward future evaluation would be to implement the new algorithm using serverless service in Databricks, which could automatically allocate the optimal resources as needed. However, this service is not available yet.
- More research is also required to determine the efficacy of using the degree of node approach on another algorithm. This could prove to be particularly valuable especially for algorithms that use the graph contraction scheme.
- Another possible area of future research would be to investigate the use of parallel disjoint-set for finding connected component locally. This could benefit from the multi-core structure available in the processing node.
- Further studies regarding designing the algorithm to use local pre-processing is strongly recommended, as most new big data processing systems support it.

- Spark is a very active developing project. Therefore, it is always recommended to revisit this study and apply updated features and best practices to help tune and troubleshoot Spark implementation of the algorithm.
- Apply the approach of using the degree in finding connected components in dynamic graphs. In addition, test the approach of local computation of connected components using disjoint-Set data structure, as it is more efficient in situation where edges are continuously being added and incremental computation of connected components is required.

References:

[1] T. Kraska, “Finding the Needle in the Big Data Systems Haystack,” *IEEE Internet Comput.*, vol. 17, no. 1, pp. 84–86, Jan. 2013.

[2] D. Cearley, “The Top 10 Strategic Technology Trends for 2015,” in *Gartner Symposium/ITxpo*, 2015.

[3] P. Zikopoulos and C. Eaton, *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill Osborne Media, 2011.

[4] S. Madden, “From Databases to Big Data,” *IEEE Internet Comput.*, vol. 16, no. 3, pp. 4–6, May 2012.

[5] E. Dumbill, “Making Sense of Big Data,” *Big Data*, vol. 1, no. 1, pp. 1–2, Mar. 2013.

[6] D. Easley and J. Kleinberg, *Networks, crowds, and markets: Reasoning about a highly connected world.* Cambridge University Press, 2010.

[7] G. D. F. Morales, “Big Data and the Web: Algorithms for Data Intensive Scalable Computing,” PhD.Thesis, IMT Institute for Advanced Studies, Lucca,Italy, 2012.

[8] R. Bordawekar, B. Blainey, C. Apte, M. Mcroberts, and Y. Heights, “Analyzing Analytics Part 1 : A Survey of Business Analytics Models and Algorithms,” *IBM Syst. J.*, vol. 42, no. 4, pp. 1–82, Feb. 2014.

[9] H. Kardes and S. Agrawal, “CCF: Fast and scalable connected component computation in MapReduce,” *2014 Int. Conf. Comput. Netw. Commun.*, pp. 994–998, Feb. 2014.

[10] J. Cohen, “Graph Twiddling in a MapReduce World,” *Comput. Sci. Eng.*, vol. 11, no. 4, pp. 29–41, 2009.

[11] S. Skhiri and S. Jouili, “Large Graph Mining: Recent Developments, Challenges and Potential Solutions,” *Bus. Intell. SE - 5*, vol. 138, pp. 103–124, 2013.

[12] H.-M. Park, N. Park, S.-H. Myaeng, and U. Kang, “Partition Aware Connected Component Computation in Distributed Systems,” in *IEEE International Conference on Data Mining series (ICDM)*, 2016.

[13] T. Seidl, B. Boden, and S. Fries, “CC-MR – Finding Connected Components in Huge Graphs with MapReduce,” in *Machine Learning and Knowledge Discovery in Databases*, 2012, pp. 458–473.

[14] R. Kiveris, S. Lattanzi, V. Mirrokni, V. Rastogi, and S. Vassilvitskii, “Connected Components in MapReduce and Beyond,” in *Proceedings of the ACM Symposium on Cloud Computing - SOCC ’14*, 2014, pp. 1–13.

[15] C. Jain, P. Flick, T. Pan, O. Green, and S. Aluru, “An Adaptive Parallel Algorithm for Computing Connected Components,” *IEEE Trans. Parallel Distrib. Syst.*, vol. XX, no. c, pp. 1–1, 2017.

[16] A. B. Patel, M. Birla, and U. Nair, “Addressing big data problem using Hadoop and

Map Reduce," *Eng. (NUiCONE), 2012 Nirma Univ. Int. Conf.*, pp. 1–5, 2012.

[17] "Apache™ Hadoop®," *Apache Hadoop*. [Online]. Available: http://hadoop.apache.org/. [Accessed: 20-Jun-2017].

[18] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, 2004, p. 10.

[19] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, p. 107, Jan. 2008.

[20] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations," in *2009 Ninth IEEE International Conference on Data Mining*, 2009, pp. 229–238.

[21] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[22] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. Das Sarma, "Finding connected components in map-reduce in logarithmic rounds," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 2013, pp. 50–61.

[23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *2nd {USENIX} Workshop on Hot Topics in Cloud Computing, HotCloud'10*, 2010, p. 10.

[24] M. Zaharia *et al.*, "Apache Spark: A Unified Engine for Big Data Processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016.

[25] J. J. E. Gonzalez, R. R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph Processing in a Distributed Dataflow Framework," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 599–613.

[26] F. Bajaber, R. Elshawi, O. Batarfi, A. Altalhi, A. Barnawi, and S. Sakr, *Big Data 2.0 Processing Systems: Taxonomy and Open Challenges*, vol. 14, no. 3. 2016.

[27] M. Romao, "A Vision for Big Data," *Policy@Intel*. 2013.

[28] Planning Guide, "Getting Started with Big Data," *Intel IT Center*, 2013.

[29] S. Sagiroglu and D. Sinanc, "Big data: A review," *2013 Int. Conf. Collab. Technol. Syst.*, pp. 42–47, May 2013.

[30] "STATS | YouTube Company Statistics – Statistic Brain." [Online]. Available: http://www.statisticbrain.com/youtube-statistics/. [Accessed: 28-Jul-2017].

[31] F. Shull, "Getting an Intuition for Big Data," *IEEE Softw.*, vol. 30, no. 4, pp. 3–6, Jul. 2013.

[32] T. Davenport, P. Barth, and R. Bean, "How 'Big Data'is Different," *MIT Sloan Manag. Rev.*, vol. 54, no. 1, 2012.

[33] A. Katal, M. Wazid, and R. H. Goudar, "Big data: Issues, challenges, tools and Good practices," in *2013 Sixth International Conference on Contemporary*

Computing (IC3), 2013, pp. 404–409.

[34] P. Hunter, “Journey to the Centre of Big Data,” *Eng. Technol.*, vol. 8, no. 3, pp. 56–59, Apr. 2013.

[35] S. Chaudhuri, “How Different is Big Data?,” in *2012 IEEE 28th International Conference on Data Engineering*, 2012, pp. 5–5.

[36] F. J. Alexander, A. Hoisie, and A. Szalay, “Big Data,” *Comput. Sci. Eng.*, vol. 13, no. 6, pp. 10–13, Nov. 2011.

[37] K. Venkatram and M. A. Geetha, “Review on Big Data & Analytics – Concepts, Philosophy, Process and Applications,” *Cybern. Inf. Technol.*, vol. 17, no. 2, Jan. 2017.

[38] D. Laney, “3-D Data Management: Controlling Data Volume, Velocity and Variety.,” *META Gr. Res. Note*, vol. 6, p. 70, 2001.

[39] A. Bhatia and G. Vaswani, “Big data: A review,” *IEEE Int. J. Eng. Sci. Res. Technol. IJESRT*, vol. 2, no. 8, pp. 2102–2106, Aug. 2013.

[40] S. Kaisler, F. Armour, J. A. Espinosa, and W. Money, “Big Data: Issues and Challenges Moving Forward,” *2013 46th Hawaii Int. Conf. Syst. Sci.*, pp. 995–1004, Jan. 2013.

[41] S. Yin and O. Kaynak, “Big Data for Modern Industry: Challenges and Trends,” *Proc. IEEE*, vol. 103, no. 2, pp. 143–146, Feb. 2015.

[42] LAVASTORM Analytics, “The Top Challenges in Big Data and Analytics,” 2013.

[43] R. Mateosian, “Ethics of Big Data,” *IEEE Micro*, vol. 33, no. 2, pp. 60–61, Mar. 2013.

[44] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, “Naiad,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP ’13*, 2013, pp. 439–455.

[45] “HPCC Systems: High performance Computing Cluster.” [Online]. Available: http://hpccsystems.com/.

[46] G. Malewicz *et al.*, “Pregel,” in *Proceedings of the 2010 international conference on Management of data - SIGMOD ’10*, 2010, p. 135.

[47] “Storm, distributed and fault-tolerant realtime computation.” [Online]. Available: https://storm.apache.org/. [Accessed: 20-Jun-2017].

[48] “S4: Distributed Stream Computubg Platform.” [Online]. Available: http://incubator.apache.org/s4/.

[49] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google file system,” *ACM SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, p. 29, Dec. 2003.

[50] U. Kang and C. Faloutsos, “Big graph mining,” *ACM SIGKDD Explor. Newsl.*, vol. 14, no. 2, p. 29, Apr. 2013.

[51] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of Massive Datasets*. Cambridge: Cambridge University Press, 2014.

[52] U. Kang, D. H. Chau, and C. Faloutsos, “Pegasus: Mining billion-scale graphs in the cloud,” in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2012, pp. 5341–5344.

[53] S. N. Srirama, P. Jakovits, and E. Vainikko, “Adapting scientific computing problems to clouds using MapReduce,” *Futur. Gener. Comput. Syst.*, vol. 28, no. 1, pp. 184–192, Jan. 2012.

[54] V. K. Vavilapalli *et al.*, “Apache Hadoop YARN,” in *Proceedings of the 4th annual Symposium on Cloud Computing - SOCC ’13*, 2013, pp. 1–16.

[55] K. Grolinger, M. Hayes, W. a. Higashino, A. L’Heureux, D. S. Allison, and M. A. M. Capretz, “Challenges for MapReduce in Big Data,” in *2014 IEEE World Congress on Services*, 2014, no. Services, pp. 182–189.

[56] G. Shroff, “Web Intelligence and Big Data,” *coursera.org*, 2013. [Online]. Available: https://class.coursera.org/bigdata-002/. [Accessed: 10-Dec-2013].

[57] Xindong Wu, Xingquan Zhu, Gong-Qing Wu, and Wei Ding, “Data mining with big data,” *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 1, pp. 97–107, Jan. 2014.

[58] J. Ekanayake *et al.*, “Twister,” in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing - HPDC ’10*, 2010, p. 810.

[59] U. Gupta and L. Fegaras, “Map-based graph analysis on MapReduce,” in *2013 IEEE International Conference on Big Data*, 2013, pp. 24–30.

[60] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, “HaLoop,” *Proc. VLDB Endow.*, vol. 3, no. 1–2, pp. 285–296, Sep. 2010.

[61] “Apache Tez.” [Online]. Available: http://hortonworks.com/hadoop/tez/. [Accessed: 20-Jun-2017].

[62] J. Ekanayake *et al.*, “DryadLINQ for Scientific Analyses,” in *2009 Fifth IEEE International Conference on e-Science*, 2009, pp. 329–336.

[63] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica, “Hyracks: A flexible and extensible foundation for data-intensive computing,” in *2011 IEEE 27th International Conference on Data Engineering*, 2011, pp. 1151–1162.

[64] D. J. Dewitt and A. Krioukov, “Clustera : An Integrated Computation And Data Management System,” vol. 1, no. 212, pp. 28–41, 2008.

[65] A. Alexandrov *et al.*, “The Stratosphere platform for big data analytics,” *VLDB J.*, vol. 23, no. 6, pp. 939–964, Dec. 2014.

[66] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas, “Apache Flink: Unified Stream and Batch Processing in a Single Engine,” *Data Eng.*, pp. 28–38, 2015.

[67] “Apache Giraph: Iterative Graph Processing System.” [Online]. Available: http://giraph.apache.org/. [Accessed: 20-Jun-2017].

[68] “Apache Spark™ - Lightning-Fast Cluster Computing.” [Online]. Available: http://spark.apache.org/. [Accessed: 20-Jun-2017].

[69] M. Zaharia *et al.*, “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*, 2012, no. UCB/EECS-2011-82, p. 2.

[70] M. Armbrust *et al.*, "Scaling spark in the real world," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1840–1843, Aug. 2015.

[71] J. Lin and C. Dyer, "Data-Intensive Text Processing with MapReduce," *Synth. Lect. Hum. Lang. Technol.*, vol. 3, pp. 1–177, 2010.

[72] V. Nicosia, J. Tang, C. Mascolo, M. Musolesi, G. Russo, and V. Latora, "Graph metrics for temporal networks," *Underst. Complex Syst.*, pp. 15–40, 2013.

[73] V. Kostakos, "Temporal graphs," *Phys. A Stat. Mech. its Appl.*, vol. 388, no. 6, pp. 1007–1023, 2009.

[74] J. Han, M. Kamber, and J. Pei, "Graph Mining, Social Network Analysis, and Multirelational Data Mining," in *Data Mining: Concepts and Techniques*, Second Edi., Morgan Kaufmann, 2005, pp. 535–589.

[75] C. C. E. Tsourakakis, "Data Mining with MAPREDUCE: Graph and Tensor Algorithms with Applications," Master thesis, Carnegie Mellon University, 2010.

[76] M. Junghanns, A. Petermann, M. Neumann, and E. Rahm, *Handbook of Big Data Technologies*. Cham: Springer International Publishing, 2017.

[77] A. Ghrab, S. Skhiri, S. Jouili, and E. Zimányi, "An Analytics-Aware Conceptual Model for Evolving Graphs," in *Data Warehousing and Knowledge …*, 2013, pp. 1–12.

[78] A. Guerrieri, "Distributed Computing for Large-scale Graphs," PhD thesis, University of Trento, 2015.

[79] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*. Elsevier, 2013.

[80] R. Angles, "A comparison of current graph database models," *Proc. - 2012 IEEE 28th Int. Conf. Data Eng. Work. ICDEW 2012*, pp. 171–177, 2012.

[81] S. Sakr, "Large-Scale Graph Processing Systems," in *Big Data 2.0 Processing Systems: A Survey*, Cham: Springer International Publishing, 2016, pp. 53–73.

[82] V. Kalavri, V. Vlassov, and S. Haridi, "High-Level Programming Abstractions for Distributed Graph Processing," pp. 1–19, 2016.

[83] D. Yan, Y. Bu, Y. Tian, A. Deshpande, and J. Cheng, "Big Graph Analytics Systems," *Proc. 2016 Int. Conf. Manag. Data*, vol. 7, no. 1–2, pp. 2241–2243, 2016.

[84] V. Kalavri, "Performance Optimization Techniques and Tools for Distributed Graph Processing School of Information and Communication Technology," Doctoral dissertatio, KTH Royal Institute of Technology, 2016.

[85] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[86] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From 'think like a vertex' to 'think like a graph,'" *Proc. VLDB Endow.*, vol. 7, no. 3, pp. 193–204, Nov. 2013.

[87] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, “SYNC or ASYNC: time to fuse for distributed graph-parallel computation,” in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP 2015*, 2015, pp. 194–204.

[88] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie, “Pregelix: Big(ger) graph analytics on a dataflow engine,” *Proc. VLDB Endow.*, vol. 8, no. 2, pp. 161–172, Oct. 2014.

[89] D. Yan, Y. Huang, J. Cheng, and H. Wu, “Efficient Processing of Very Large Graphs in a Small Cluster,” Jan. 2016.

[90] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, and X. Lin, “Scalable big graph processing in MapReduce,” in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD ’14*, 2014, pp. 827–838.

[91] R. R. McCune, T. Weninger, and G. Madey, “Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing,” *ACM Comput. Surv.*, vol. 48, no. 2, 2015.

[92] E. Carlini, P. Dazzi, A. Lulli, and L. Ricci, “Layered Thinking in Vertex Centric Computations,” pp. 1–3.

[93] E. Carlini, P. Dazzi, A. Lulli, and L. Ricci, “Distributed Graph Processing: An Approach based on Overlay Composition,” *Sac 2016*, pp. 1912–1917, 2016.

[94] R. Lämmel, “Google’s MapReduce programming model — Revisited,” *Sci. Comput. Program.*, vol. 70, no. 1, pp. 1–30, Jan. 2008.

[95] P. Stutz, A. Bernstein, and W. Cohen, “Signal/Collect: Graph Algorithms for the (Semantic) Web,” Springer, Berlin, Heidelberg, 2010, pp. 764–780.

[96] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. 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)*, 2012, pp. 17–30.

[97] Y. Lu, J. Cheng, D. Yan, and H. Wu, “Large-scale distributed graph computing systems,” *Proc. VLDB Endow.*, vol. 8, no. 3, pp. 281–292, Nov. 2014.

[98] D. Yan, J. Cheng, Y. Lu, and W. Ng, “Blogel - a block-centric framework for distributed computation on real-world graphs,” *Proc. VLDB Endow.*, vol. 7, no. 14, pp. 1981–1992, 2014.

[99] A. Quamar, A. Deshpande, and J. Lin, “NScale: Neighborhood-centric Analytics on Large Graphs,” *Proc. VLDB Endow.*, vol. 7, no. 13, pp. 1673–1676, 2014.

[100] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, “Arabesque,” in *Proceedings of the 25th Symposium on Operating Systems Principles - SOSP ’15*, 2015, pp. 425–440.

[101] M. A. Rodriguez and M. A., “The Gremlin graph traversal machine and language (invited talk),” in *Proceedings of the 15th Symposium on Database Programming Languages - DBPL 2015*, 2015, pp. 1–10.

[102] M. Junghanns, A. Petermann, N. Teichmann, K. Gómez, and E. Rahm, “Analyzing

extended property graphs with Apache Flink," in *Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics - NDA '16*, 2016, pp. 1–8.

[103] H.-C. Lai, C.-T. Li, Y.-C. Lo, and Shou-De Lin, "Exploiting and Evaluating MapReduce for Large-Scale Graph Mining," *2012 IEEE/ACM Int. Conf. Adv. Soc. Networks Anal. Min.*, pp. 434–441, Aug. 2012.

[104] R. R. S. Xin, J. J. E. Gonzalez, M. J. M. Franklin, and I. Stoica, "GraphX: A Resilient Distributed Graph System on Spark," in *First International Workshop on Graph Data Management Experiences and Systems*, 2013, p. 2:1--2:6.

[105] R. S. Xin, D. Crankshaw, A. Dave, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: Unifying Data-Parallel and Graph-Parallel Analytics," *arXiv Prepr. arXiv1402.2394*, Feb. 2014.

[106] "GraphX Programming Guide - Spark 1.2.0 Documentation." [Online]. Available: https://spark.apache.org/docs/latest/graphx-programming-guide.html. [Accessed: 20-Jun-2017].

[107] P. Carbone Asterios Katsifodimos, S. Ewen Volker Markl, and S. Haridi Kostas Tzoumas, "Apache Flink™: Stream and Batch Processing in a Single Engine," *Bull. IEEE Comput. Soc. Tech. Comm. Data Eng.*, vol. 36, no. 4, 2015.

[108] R. Mccoll, O. Green, and D. A. Bader, "A New Parallel Algorithm for Connected Components in Dynamic Graphs," in *The 20th Annual IEEE International Conference on High Performance Computing (HiPC)*, 2013, pp. 246–255.

[109] A. Sharma and R. Misra, "Computing Large Connected Components Using Map Reduce in Logarithmic Rounds," in *2016 IEEE 2nd International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS)*, 2016, pp. 1–6.

[110] M. M. A. Patwary, P. Refsnes, and F. Manne, "Multi-core Spanning Forest Algorithms using the Disjoint-set Data Structure," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012, pp. 827–835.

[111] M.-S. Kim, S. Lee, W.-S. Han, H. Park, and J.-H. Lee, "DSP-CC-: I/O Efficient Parallel Computation of Connected Components in Billion-Scale Networks," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 10, pp. 2658–2671, Oct. 2015.

[112] R. McColl, O. Green, and D. A. Bader, "A new parallel algorithm for connected components in dynamic graphs," in *20th Annual International Conference on High Performance Computing*, 2013, pp. 246–255.

[113] F. N. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. D. Ullman, "Map-reduce extensions and recursive queries," in *Proceedings of the 14th International Conference on Extending Database Technology - EDBT/ICDT '11*, 2011, p. 1.

[114] B. Wu and Y. Du, "Cloud-based Connected Component Algorithm," in *2010 International Conference on Artificial Intelligence and Computational Intelligence*, 2010, pp. 122–126.

[115] A. Lulli, L. Ricci, E. Carlini, P. Dazzi, and C. Lucchese, "Cracker: Crumbling large graphs into connected components," in *2015 IEEE Symposium on Computers and Communication (ISCC)*, 2015, pp. 574–581.

[116] A. Lulli, E. Carlini, P. Dazzi, C. Lucchese, and L. Ricci, “Fast Connected Components Computation in Large Graphs by Vertex Pruning,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 3, pp. 760–773, Mar. 2017.

[117] R. Elshawi, O. Batarfi, A. Fayoumi, A. Barnawi, and S. Sakr, “Big Graph Processing Systems: State-of-the-Art and Open Challenges,” in *2015 IEEE First International Conference on Big Data Computing Service and Applications*, 2015, pp. 24–33.

[118] J. Lin and M. Schatz, “Design patterns for efficient graph algorithms in MapReduce,” in *Proceedings of the Eighth Workshop on Mining and Learning with Graphs - MLG ’10*, 2010, pp. 78–85.

[119] A. Varamesh, “Fast Detection of Connected Components in Large Scale Graphs Using MapReduce,” *IOSR J. Eng.*, vol. 4, no. 2, pp. 35–42, Feb. 2014.

[120] L. Kolb, Z. Sehili, and E. Rahm, “Iterative Computation of Connected Graph Components with MapReduce,” *Datenbank-Spektrum*, vol. 14, no. 2, pp. 107–117, Jul. 2014.

[121] Y. Deng, J. Wu, and Y. Tan, “A fast connected component algorithm based on hub contraction,” in *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2016, pp. 000066–000069.

[122] S. Dasgupta, C. H. Papadimitriou, and U. Vazirani, *Algorithms*, 1 edition. McGraw-Hill, Inc., 2006.

[123] “Spark Standalone Mode - Spark 1.2.0 Documentation.” [Online]. Available: http://spark.apache.org/docs/latest/spark-standalone.html#cluster-launch-scripts. [Accessed: 20-Jun-2017].

[124] “Collections - Sets - Scala Documentation.” [Online]. Available: http://docs.scala-lang.org/overviews/collections/sets.html. [Accessed: 20-Jun-2017].

[125] T. Rabl, N. Raghunath, M. Poess, M. Bhandarkar, H.-A. Jacobsen, and C. Baru, Eds., *Advancing Big Data Benchmarks*, vol. 8585. Cham: Springer International Publishing, 2014.

[126] A. Iosup *et al.*, “LDBC graphalytics,” *Proc. VLDB Endow.*, vol. 9, no. 13, pp. 1317–1328, Sep. 2016.

[127] P. Mehrotra *et al.*, “Performance evaluation of Amazon Elastic Compute Cloud for NASA high-performance computing applications,” *Concurr. Comput. Pract. Exp.*, vol. 28, no. 4, pp. 1041–1055, Mar. 2016.

[128] U. Kang, D. H. Chau, and C. Faloutsos, “Mining large graphs: Algorithms, inference, and discoveries,” in *2011 IEEE 27th International Conference on Data Engineering*, 2011, pp. 243–254.

Appendix A: Code

// import the library needed for the execution

```
import org.apache.spark.rdd.RDD

import org.apache.spark.SparkContext

import org.apache.spark.graphx._

import scala.reflect.ClassTag

import scala.collection.immutable.HashMap

import scala.collection.mutable.HashSet

import scala.collection.mutable.ListBuffer

import scala.collection.mutable.Set
```

8.6 Pre-Processing Stage

// defined a function to restructure the graph in adjacency list representation:

```
def adjacencyListGeneratorOpt[VD:ClassTag, ED:ClassTag](graph: Graph[VD, ED]): Graph[Set[VertexId], ED] = {

   val WorkGraph = graph.mapVertices { case (vid, _) => (vid) }

   val nbrs = WorkGraph.collectNeighborIds(EdgeDirection.Either).cache()

   val nbrsVerts: VertexRDD[Set[VertexId]] = nbrs.mapValues ( (vid, nbrs) => Set(nbrs.toSet.toArray: _*))

   val adjGraph: Graph[Set[VertexId], ED] = Graph(nbrsVerts, graph.edges)

   adjGraph

}
```

// defined a function to restructure the graph in adjacency list representation:

```
def adjacencyListGeneratorDgOpt[VD: ClassTag, ED: ClassTag](graph: Graph[VD, ED]): Graph[(Int,scala.collection.immutable.Set[(VertexId,Int)]), ED] = {

   val degrees = graph.degrees

   val graphWithDegrees = graph.outerJoinVertices(degrees) { (_, _, optDegree) =>

      optDegree.getOrElse(1)

   }

   val WorkGraph = graphWithDegrees.mapVertices { case (vid, degree) => (vid,degree) }
```

```
   val neighboursWithDegree = WorkGraph.aggregateMessages[scala.collection.immutable.Set[(VertexId, Int)]](

       sendMsg = triplet => {

          val srcWithDegree = triplet.srcAttr

          val dstWithDegree = triplet.dstAttr

          triplet.sendToDst(scala.collection.immutable.Set(srcWithDegree))

          triplet.sendToSrc(scala.collection.immutable.Set(dstWithDegree))

       },

       mergeMsg = (x, y) => x ++ y

   ).mapValues(x=>(x.size,x))

   val emptySet:(Int,scala.collection.immutable.Set[(VertexId,Int)])= ( 0, scala.collection.immutable.Set())

   val adjGraph = graph.outerJoinVertices(neighboursWithDegree) { (_, _, optDegree) =>

          optDegree.getOrElse(emptySet)

       }

   adjGraph

}
```

// apply the adjacencylistGenerator function on the graph

```
val adjGraphDg = adjacencyListGeneratorDgOpt(graph)
```

// create the initial graph and add the component identifier for each node using the *findMaxCompInSet* function:

```
val ret_Initial = adjGraphDg.vertices.map(t =>

                        (t._1, (t._2._1,new DgCracker_MsgIdentification(

                        findMaxCompInSet_Dg((t._1,t._2._1),t._2._2),t._2._2 ))))

                        .cache

var ret= ret_Initial
```

8.7 Processing Stage

Chapter 9: Seed Identification Phase

a) Local Max Identification Step

// Defined the map function for the Local Max Identification Step

```
def map_LocalMaxIdentification_DisjointSet(items :  Iterator[(VertexId,(Int,
DgCracker_MsgIdentification))], forceLoadBalancing : Boolean) = {

 val ds =  new DisjointSetModified()

 var cache =scala.collection.mutable.HashMap.empty[VertexId,(Int,DgCracker_MsgIdentification)]

 while (items.hasNext) {

  val cur = items.next

  val id = cur._1

  val dg = cur._2._1

  val idDg= (id,dg)

  val max = cur._2._2.max

  val adjSet = cur._2._2.neigh

   if (!(cache contains id)){

    cache.put(id, (dg,new DgCracker_MsgIdentification(max, adjSet)))

  }

  if (!(cache contains max._1)){

    cache.put(max._1, (max._2,new DgCracker_MsgIdentification(max,  scala.collection.immutable.Set())))

  }

    if (!(ds contains(id))){

    ds+= idDg

  }

  if (!(ds contains(max._1))){

    ds+= max

  }

  ds union(idDg,max)
```

```
    if (adjSet.size >0)  {

      for (node<- adjSet){

        if (!(cache contains node._1)){

//          cache.put(node._1, (node._2,new DgCracker_MsgIdentification(max,
scala.collection.immutable.Set())))

        }

        if (!(ds contains(node._1))){

          ds+= node

        }

        ds union(idDg,(ds(node),node._2))

      }

    }

  }

    cache.foreach{ cur =>

        val id = cur._1

        val dg = cur._2._1

        val idDg= (id,dg)

        val max = cur._2._2.max

        val adjSet = cur._2._2.neigh

        val maxNew= ds(idDg)

        val maxNewDg= (maxNew,cache(maxNew)._1)

        cache(id)= (dg,new DgCracker_MsgIdentification(maxNewDg, adjSet))

  }

  cache.iterator

  }
```

// Apply using the mapPartition function

```
val ret_LocalMax = ret.mapPartitions { ItrInp => {
map_LocalMaxIdentification_DisjointSetModified(ItrInp)}}

ret =ret_LocalMax
```

b) Cluster Max Identification Step

// defined the map function for the Cluster Max Identification Step

```
def map_ClusterMaxIdentification(item : (VertexId,(Int, DgCracker_MsgIdentification)),
forceLoadBalancing : Boolean, edgePruning : Boolean = true) : Iterable[(VertexId, (Int,
DgCracker_MsgIdentification))] =    {

      var outputList : ListBuffer[(VertexId, (Int, DgCracker_MsgIdentification))] = new ListBuffer

      if (item._2._2.max._1 == item._1 && (item._2._2.neigh.isEmpty || (item._2._2.neigh.size == 1 &&
item._2._2.neigh.contains((item._1,item._2._1))))) {  //this will deactivate  single nodes or root nodes

      } else {

         val max = item._2._2.max

         if (item._2._2.neigh.isEmpty) {

            outputList.prepend((item._1,(item._2._1, new DgCracker_MsgIdentification(max,
scala.collection.immutable.Set()))))

         } else {

            outputList.prepend((item._1,(item._2._1, new DgCracker_MsgIdentification(max,
scala.collection.immutable.Set(max)))))

         }

         if (max._2 > item._2._1 || !forceLoadBalancing || !edgePruning) {

            val it = item._2._2.neigh.iterator

            while (it.hasNext) {

               val next = it.next

               outputList.prepend((next._1,(next._2, new DgCracker_MsgIdentification(max,
scala.collection.immutable.Set(max)))))

            }

         }

      }

      outputList.toIterable

   }
```

// Apply the map function

```
val map_ClusterMax= ret.flatMap(item => map_ClusterMaxIdentification(item, false))
```

// defined the Reduce function for the Cluster Max Identification Step

```
def reduce_ClusterMaxIdentification(item1 : (Int, DgCracker_MsgIdentification), item2 : (Int,
DgCracker_MsgIdentification)) : (Int, DgCracker_MsgIdentification) = {
```

```
    val ret = item1._2.neigh ++ item2._2.neigh

    val max = maxDg(item1._2.max,item2._2.max)

    val dg  = if(item1._1> item2._1) item1._1 else  item2._1

    (dg,new DgCracker_MsgIdentification(max, ret))

}
```

// Apply the reduce function

```
val reduce_ClusterMax =  map_ClusterMax.reduceByKey(reduce_ClusterMaxIdentification)//.dependencies
```

c) Pruning Step

c) Node Assorting

//mapper for the node assorting step

```
def map_Pruning(item : (VertexId,(Int, DgCracker_MsgIdentification))) : Iterable[(VertexId,(Int,
DgCracker_GeneralMsg))] = {

    map_Pruning(item, false)

}

def map_Pruning(item : (VertexId,(Int, DgCracker_MsgIdentification)), forceLoadBalancing : Boolean,
obliviousSeed : Boolean = true) : Iterable[(VertexId,(Int, DgCracker_GeneralMsg))] = {

    var outputList : ListBuffer[(VertexId,(Int, DgCracker_GeneralMsg))] = new ListBuffer

    val maxSet : scala.collection.immutable.Set[(VertexId,Int)] = item._2._2.neigh

      if (maxSet.size > 1) {

        if( forceLoadBalancing || obliviousSeed)   {

            outputList.prepend((item._2._2.max._1,(item._2._2.max._2, DgCracker_GeneralMsg.apply(new
DgCracker_MsgIdentification(item._2._2.max, scala.collection.immutable.Set(item._2._2.max))))))

        }

        else {

            outputList.prepend((item._2._2.max._1,(item._2._2.max._2, DgCracker_GeneralMsg.apply(new
DgCracker_MsgIdentification(item._2._2.max, maxSet)))))

        }

        var it = maxSet.iterator

        while (it.hasNext) {

            val value : (VertexId,Int) = it.next

            if (value != item._2._2.max)
```

```
            outputList.prepend((value._1,(value._2, DgCracker_GeneralMsg.apply(new
DgCracker_MsgIdentification(item._2._2.max, scala.collection.immutable.Set(item._2._2.max))))))

        }

    } else if (maxSet.size == 1 && maxSet.contains((item._1,item._2._1))) {

        outputList.prepend((item._1, (item._2._1,DgCracker_GeneralMsg.apply(new
DgCracker_MsgIdentification((item._1,item._2._1), scala.collection.immutable.Set())))))

    }

    if (!item._2._2.neigh.contains((item._1,item._2._1))) {

        outputList.prepend((item._2._2.max._1,(item._2._2.max._2, DgCracker_GeneralMsg.apply(new
DgCracker_MsgTree(-1, scala.collection.immutable.Set(item._1))))))

        outputList.prepend((item._1,(item._2._1, DgCracker_GeneralMsg.apply(new
DgCracker_MsgTree(item._2._2.max._1, scala.collection.immutable.Set())))))

    }

                    outputList.toIterable

}
```

// the merge function for message that update the Graph G^{t+1}

```
def merge_MsgTree(first : Option[DgCracker_MsgTree], second : Option[DgCracker_MsgTree]) :
Option[DgCracker_MsgTree] = {

    if (first.isDefined) {

        first.get.merge(second)

    } else {

        second

    }

}
```

//The merge function for messages that update the propagation tree T

```
def merge_MsgIdentification(first : Option[DgCracker_MsgIdentification], second :
Option[DgCracker_MsgIdentification]) : Option[DgCracker_MsgIdentification] = {

    if (first.isDefined) {

        first.get.merge(second)

    } else {

        second

    }

}
```

// reducer for the node assorting step

```
def reduce_Pruning(item1 : (Int, DgCracker_GeneralMsg), item2 : (Int, DgCracker_GeneralMsg)) : (Int, DgCracker_GeneralMsg)= {

  val dg = if(item1._1< item2._1) item1._1  else item2._1

   (dg, new DgCracker_GeneralMsg(merge_MsgIdentification(item1._2.MsgIdentification, item2._2.MsgIdentification), merge_MsgTree(item1._2.MsgTree, item2._2.MsgTree)))

}
```

// Initialise variables to check active nodes after Seed node assorting step

```
val active = ret.count

val control = active == 0
```

d) Update Degree

// the mapper function to generate the updates

```
def DgMap(item: (VertexId,(Int, scala.collection.immutable.Set[(VertexId, Int)]))) : Iterable[(VertexId,Int)] ={

    var outputList : ListBuffer[(VertexId,Int)] = new ListBuffer

    val it = item._2._2.iterator

    while(it.hasNext)    {

      val next = it.next

      if (next._1!= item._1) {

        outputList.prepend((next._1, 1))

        outputList.prepend((item._1, 1))

      }

    }

    outputList.toIterable

}
```

//calculate the new degree using Mapreduce

```
val degreePair=tmpReduced_MsgIdentification

                              .flatMap(x=>DgMap((x._1,(x._2._1,(x._2._2.neigh)))) )

                              .reduceByKey(_+_)

// save the updates in HashMap table

val degreeMap= degreePair.collectAsMap
```

```
val degreeBroadcast= sc.broadcast(degreeMap)
```

// the update function

```
def updateDegree(item: (VertexId, (Int, DgCracker_MsgIdentification)))  = {
  val id = item._1
  val dg_old=item._2._1
  val max = item._2._2.max
  val neigh = item._2._2.neigh
  var dg_new= degreeBroadcast.value.get(id).getOrElse(dg_old)
  if (dg_new== -1) dg_new=dg_old
  val it = neigh.iterator
  var max_new= (max._1,degreeBroadcast.value.get(max._1).getOrElse(max._2))
  var neigh_new = scala.collection.immutable.Set[(VertexId, Int)]()
    while(it.hasNext)
    {
      val next = it.next
      neigh_new +=((next._1,degreeBroadcast.value.get(next._1).getOrElse(next._2)))
      }
 (id,(dg_new, new DgCracker_MsgIdentification(max_new,neigh_new)))
}
```

// merge the updates

```
val retUpdated = tmpReduced_MsgIdentification.mapPartitions((it =>
                                  it.map{case (id, attr)=> updateDegree(id,attr)}),
                                  preservesPartitioning = true)
```

e) Update Propagation Tree

```
//Initialise the propagationTree as empty
var propagationTreeRDD : Option[RDD[(VertexId, DgCracker_MsgTree)]] = Option.empty

// add each node and initialize the component identifier as -1
// If not done, CC of size 1 are not recognized
propagationTreeRDD = Option.apply(ret.map(t => (t._1, new DgCracker_MsgTree(-1,
scala.collection.immutable.Set()))))

// merging function
def mergePropagationTree(start : Option[RDD[(VertexId, DgCracker_MsgTree)]], add : RDD[(VertexId,
DgCracker_MsgTree)]) : Option[RDD[(VertexId, DgCracker_MsgTree)]] =

 {

    if (start.isDefined) {

          Option.apply(start.get.union(add))

    } else {

       Option.apply(add)

    }

 }

// merge the tree
propagationTreeRDD = mergePropagationTree(propagationTreeRDD, tmpReduced_MsgTree)
```

Implementation

// implement Seed Identification Phase iteratively in while loop

// configuration initialising

```
cracker_Skip_LocalMaxIdentification  = true

cracker_Skip_PruningUpdateDegree = false

cracker_Skip_PruningUpdateTree   = false

var numIter = 0

ret= ret_Initial

propagationTreeRDD = Option.apply(ret.map(t => (t._1, new DgCracker_MsgTree(-1,
scala.collection.immutable.Set()))))

val active =ret.count

control= active==0
```

// Run everything in while loop (iterate until convergance)

```
while (!control) {

 // 2. Seed Identification

   numIter+=1



 // 2.1. Local Max Identification

   if (!cracker_Skip_LocalMaxIdentification){

   val ret_LocalMax = ret.mapPartitions { ItrInp => {
map_LocalMaxIdentification_DisjointSetModified(ItrInp,false)} }

   ret =ret_LocalMax

   }



 // 2.2 Cluster Max Identification

   val map_ClusterMax= ret.flatMap(item => map_ClusterMaxIdentification(item, false))

   val reduce_ClusterMax = map_ClusterMax.reduceByKey(reduce_ClusterMaxIdentification)

   val active = ret.count

   control = active == 0
```

```
    // 2.3 Purning Step

    if (!control) {

      // Node Assorting

      val tmp = ret.flatMap(item => map_Pruning(item))

      val tmpReduced = tmp.reduceByKey(reduce_Pruning).cache

      val tmpReduced_MsgIdentification=tmpReduced.filter(t => t._2._2.MsgIdentification.isDefined).map(t
=> (t._1, (t._2._1,t._2._2.MsgIdentification.get)))

      val tmpReduced_MsgTree =tmpReduced.filter(t => t._2._2.MsgTree.isDefined).map(t => (t._1,
t._2._2.MsgTree.get))

         // Update Degree

      if(!cracker_Skip_PruningUpdateDegree){

        val degreePair=tmpReduced_MsgIdentification

                              .flatMap(x=>DgMap((x._1,(x._2._1,(x._2._2.neigh)))) )

                              .reduceByKey(_+_)

        val degreeMap= degreePair.collectAsMap

        val degreeBroadcast= sc.broadcast(degreeMap)

        val retUpdated = tmpReduced_MsgIdentification.mapPartitions((it =>

                                 it.map{case (id, attr)=> updateDegree(id,attr)}), preservesPartitioning = true)
//         degreeBroadcast.destroy

        ret = retUpdated

      }else{

        ret= tmpReduced_MsgIdentification

      }

       //update Propagation Tree

      if(!cracker_Skip_PruningUpdateTree){

       propagationTreeRDD = mergePropagationTree(propagationTreeRDD, tmpReduced_MsgTree)

      }else{

        propagationTreeRDD= propagationTreeRDD

      }

    }

}
```

Chapter 10: Seed Propagation Phase:

// get the data from the previous phase

```
var propagationTreeRDD_tmp = propagationTreeRDD.get
```

// reduce function to prepare the data for processing

```
def reducePrepareDataForPropagation(a : DgCracker_MsgTree, b : DgCracker_MsgTree) :
DgCracker_MsgTree = {

   var parent = a.parent

   if (parent == -1) parent = b.parent

   new DgCracker_MsgTree(parent, a.child ++ b.child)

}
```

// prepare the data for processing

```
var propagationTreeRDD_clean = propagationTreeRDD_tmp

                        .reduceByKey(reducePrepareDataForPropagation)

                        .map(t => (t._1, t._2.getMessagePropagation(t._1))).cache
```

// flag to skip running local seed propagation

```
val cracker_Skip_PartitionMapPropagate= false
```

a) Local Seed Propagation.

```
def mapPropagatePart_par(items :  Iterator[(Long, DgCracker_MsgPropagation)]) = //: Iterable[(Long,
CrackerTreeMessagePropagation)] = {

        var outputList : ListBuffer[(Long, DgCracker_MsgPropagation)] = new ListBuffer

          var update: Boolean = true

          control = false

          numIter = 0

           var  treeRDDPropagationPar=   items.flatMap(mapPropagate(_)).toList.groupBy(t => t._1).map {
case (group, traversable) => (group, traversable.map(t=> t._2).reduce(reducePropagate)) }

        control =   treeRDDPropagationPar.map(t => t._2.max != -1).reduce {  (a:Boolean, b:Boolean) => a
&& b }

          while (!control && numIter<20) {

            treeRDDPropagationPar=   treeRDDPropagationPar.toList.flatMap(mapPropagate(_)).groupBy(t =>
t._1)

                                                .map { case (group, traversable) => (group, traversable.map(t=>
t._2).reduce(reducePropagate)) }
```

```
            control =   treeRDDPropagationPar.map(t => t._2.max != -1).reduce {  (a:Boolean, b:Boolean) => a
&& b }

            numIter+=1

          }

        treeRDDPropagationPar.toIterator

}
```

b) Cluster Seed Propagation

// Map function to propagate the component identifier

```
def mapPropagate(item : (VertexId, DgCracker_MsgPropagation)) : Iterable[(VertexId,
DgCracker_MsgPropagation)] = {

            var outputList : ListBuffer[(Long, DgCracker_MsgPropagation)] = new ListBuffer

            if (item._2.max != -1) {

                        outputList.prepend((item._1, new DgCracker_MsgPropagation(item._2.max, Set())))

                        val it = item._2.child.iterator

                        while (it.hasNext) {

                                    val next = it.next

                                    outputList.prepend((next, new DgCracker_MsgPropagation(item._2.max, Set())))

                        }

            } else { outputList.prepend(item) }

            outputList

}
```

// Reduce function to propagate the component identifier

```
def reducePropagate(item1 : DgCracker_MsgPropagation, item2 : DgCracker_MsgPropagation) :
DgCracker_MsgPropagation =    {

      var maxEnd = item1.max

      if (maxEnd == -1) maxEnd = item2.max

      new DgCracker_MsgPropagation(maxEnd, item1.child ++ item2.child)

   }
```

// Run everything in while loop (iterate until convergance)

```
while (!control) {
```

```
  if(cracker_Skip_PartitionMapPropagate){
    propagationTreeRDD_clean = propagationTreeRDD_clean
//                    .mapPartitionsWithIndex { (idx, ItrInp) => { mapPropagatePart3(ItrInp)} }
                    .mapPartitionsWithIndex { (idx, ItrInp) => { mapPropagatePart_par(ItrInp)} }
  }
    propagationTreeRDD_clean = propagationTreeRDD_clean.flatMap(item => mapPropagate(item))
    propagationTreeRDD_clean = propagationTreeRDD_clean.reduceByKey(reducePropagate).cache
    control = propagationTreeRDD_clean.map(t => t._2.max != -1).reduce { case (a, b) => a && b }
//     step = step + 1
  numIter+=1
 }
```

10.1Post-Processing Stage

// found out the number of nodes in each component

```
val Final =propagationTreeRDD_clean.map(t => (t._2.max, 1)).reduceByKey { case (a, b) => a + b }
```

// take the top 10 components with the highest number of nodes

```
Final.sortBy(x => x._2, false).take(10).foreach(println(_))
```

10.2Implementation Using Disjoint-Set Pruning

// initialise the Disjoint-Set Tree

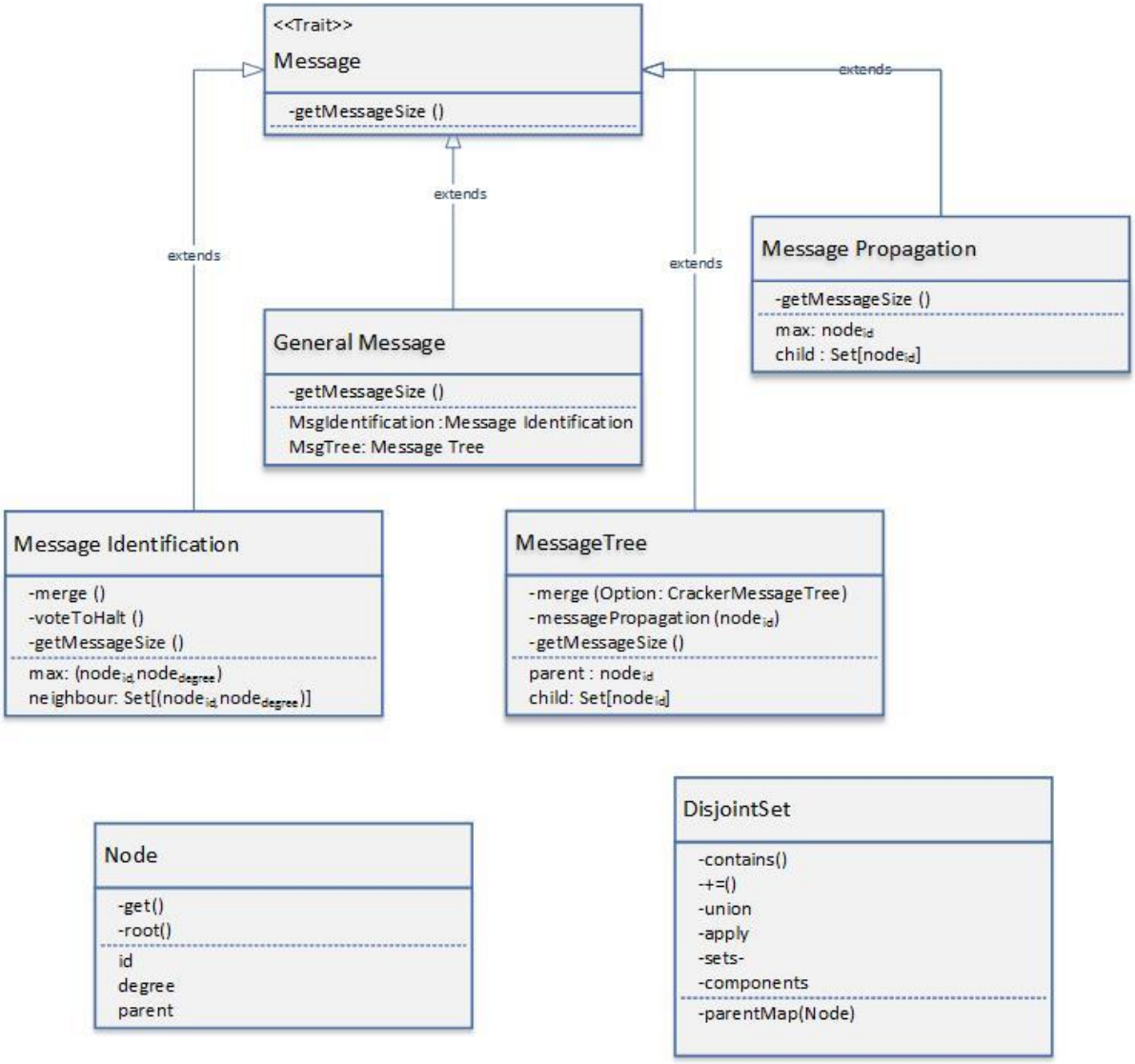
```
val treeDS =  new DisjointSetModified()
ret.map(t => (t._1,t._2)).foreach(t => if (!(treeDS contains(t._1))){treeDS+= (t._1,t._2._1)})
```

// update the treeDS in each iteration of the Seed Identification phase

```
while (!control) {
   numIter1srPhase+=1
   // simplification step
   ret = ret.flatMap(item => emitBlue(item, false))

   ret = ret.reduceByKey(reduceBlue).cache

   val active = ret.count
   control = active == 0
   stat_1srPhase+=((numIter1srPhase,active))

   if (!control) {
      // reduction step
      val tmp = ret.flatMap(item => emitRed(item))

      val tmpReduced = tmp.reduceByKey(reduceRed)

      val tmpReduced_MsgIdentification= tmpReduced.filter(t => t._2._2.first.isDefined).map(t => (t._1,
(t._2._1,t._2._2.first.get)))
      val tmpReduced_MsgTree =tmpReduced.filter(t => t._2._2.second.isDefined).map(t => (t._1,
(t._2._1,t._2._2.second.get)))

      treeRDD = mergeTree(treeRDD,tmpReduced_MsgTree , crackerUseUnionInsteadOfJoin,
crackerForceEvaluation)

//      tmpReduced_MsgIdentification.map(t => t).collect.map(node=> {
      tmpReduced_MsgTree.map(t => t).collect.map(node=> {
        val nodeDg= (node._1,node._2._1)
//          val max = node._2._2.max
        val max = node._2._2.parent
        if (!(treeDS contains(node._1))){
          treeDS+= nodeDg
        }
        if (!(treeDS contains(max._1))){
          treeDS+= max
        }
        treeDS union(nodeDg,(treeDS(max),max._2))
      })

      ret= tmpReduced_MsgIdentification
      step = step + 2
   } else {
      step = step + 1
   }
}
```

Classes

// Class Diagram structure



// DisjointSet Class

```
class DisjointSetModified extends Serializable  {


//https://github.com/pathikrit/scalgos/blob/master/src/main/scala/com/github/pathikrit/scalgos/DisjointSet.sca
la
  import scala.collection.mutable
  import DisjointSetModified.Node
  val parentMap = mutable.Map.empty[VertexId, Node]
  private var numComponents  =0
  private[this] implicit def toNode(x: (VertexId,Int)) = {
    if(contains(x._1)){
          }
    parentMap(x._1)
  }
```

```
//  @return true iff x is known
  def contains(x: VertexId) = parentMap contains x

// Add a new singleton set with only x in it (assuming x is not already known)
  def +=(x: (VertexId,Int)) = {
    if(!contains(x._1)){
      parentMap(x._1) = new Node(x._1,x._2)
      numComponents+=1
    }
  }
  override def toString = parentMap.toString

//  Union the sets containing x and y
   def union(x: (VertexId,Int), y: (VertexId,Int)) = {

    val (xRoot, yRoot) = (x.root, y.root)
    if (xRoot != yRoot) {
      if (xRoot.dg < yRoot.dg) {
        xRoot.parent = yRoot
      } else if (xRoot.dg > yRoot.dg) {
        yRoot.parent = xRoot
      } else {
        if(xRoot.id > yRoot.id){
          yRoot.parent = xRoot
        }else{
          xRoot.parent = yRoot
        }
      }
      numComponents-=1
    }
  }
  //   @return the root (or the canonical element that contains x)
      def apply(x: (VertexId,Int)) = x.root.id
  //   @return Iterator over groups of items in same set
    def sets = parentMap groupBy {_._2.root.id} values
   //   @return number of components
     def components: Int = numComponents
  }
object DisjointSetModified {
// Each internal node in DisjointSet
  class Node(val id: VertexId,val dg:Int=0){
    /**
     * parent - the pointer to root node (by default itself)
     * rank - depth if we did not do path compression in find - else its upper bound on the distance from node
to parent
     */
    var parent= this
    def get={
      (id,dg)
    }
    var rank  = 0
    def root: Node = {
      if (parent != this) {
        parent = parent.root     // path compression
      }
      parent
    }
    override def toString = "{("+id.toString+","+dg.toString+")-> parent:("+parent.id+","+parent.dg+")}"

  }
```

```
//  @return empty disjoint set
  def empty[VertexId,Int] = new DisjointSetModified

//  @return a disjoint set with each element in its own set
  def apply[A,T](elements: (VertexId,Int)*) = {
    val d = empty[VertexId,Int]
    elements foreach {e => d += e}
    d
  }
}
```

// defind function used in Classes

// function to find the the max between two nodes based on the degree then the ID:

```
def maxDg(ver1:  (VertexId,Int), ver2:(VertexId,Int)):(VertexId,Int) ={
  if (ver1._2 > ver2._2) ver1
  else if (ver1._2 == ver2._2){
    if (ver1._1 > ver2._1) ver1
    else ver2
  }
  else ver2
}
```

// function to find the the max in a set of nodes based on the degree then the ID:

```
def findMaxCompInSet_Dg(compID: (VertexId,Int), setDg:scala.collection.immutable.Set[(VertexId, Int)]):
(VertexId,Int) = {

  var setMaxDg = compID

  if (!setDg.isEmpty) {

    setMaxDg = setDg.reduceLeft(maxDg)

  }

  maxDg(setMaxDg , compID)

}
```

// a function to give Boolean if first node is less than the second

```
def lessThan(left:(VertexId,Int),right:(VertexId,Int)):Boolean ={

    if (left._2<right._2) true

    else if (left._2==right._2){

      if(left._1< right._1) true else false

    }else false

}
```

// Message abstract class

```
trait CrackerMessageSize extends Serializable{

def getMessageSize : Long

}
```

// Message Identification Class

```
class DgCracker_MsgIdentification (val max: (VertexId,Int), val neigh:
scala.collection.immutable.Set[(VertexId,Int)]) extends CrackerMessageSize {

        def voteToHalt = neigh.isEmpty

        def getMessageSize = neigh.size + 1

        def merge(other : Option[DgCracker_MsgIdentification]) : Option[DgCracker_MsgIdentification] =

        {

                if(other.isDefined)

                {

                        Option.apply(new DgCracker_MsgIdentification(maxDg(max, other.get.max),

                                neigh ++ other.get.neigh))

                } else

                {

                        Option.apply(DgCracker_MsgIdentification.this)

                }

        }

        override def toString = "(max:"+max.toString+",neigh:"+neigh.toString+")"

}

object DgCracker_MsgIdentification {

        def empty = new DgCracker_MsgIdentification((-1,0), scala.collection.immutable.Set())

}
```

// Message Propagation Class

```
class DgCracker_MsgPropagation (val max : VertexId, val child : scala.collection.immutable.Set[VertexId])
extends CrackerMessageSize {

          def getMessageSize = child.size + 1

   override def toString = "{"+max.toString+","+child.toString+"}"

}
```

// Message Tree Class

```
class DgCracker_MsgTree (val parent : VertexId, val child : scala.collection.immutable.Set[VertexId])
extends CrackerMessageSize
{
	def getMessageSize = child.size + 1

	def merge(other : Option[DgCracker_MsgTree]) : Option[DgCracker_MsgTree] =
	{
		if(other.isDefined)
		{
			var parentNew = parent

			if(parentNew == -1)
			{
				parentNew = other.get.parent
			}

			Option.apply(new DgCracker_MsgTree(parentNew, child ++ other.get.child))
		} else
		{
			Option.apply(DgCracker_MsgTree.this)
		}
	}

	def merge(other : DgCracker_MsgTree) : DgCracker_MsgTree =
	{
		var parentNew = parent

		if(parentNew == -1)
		{
			parentNew = other.parent
		}

		new DgCracker_MsgTree(parentNew, child ++ other.child)
	}
	def getMessagePropagation(id : VertexId) =
	{
		if(parent == -1)
		{
			new DgCracker_MsgPropagation(id, child)
```

```
                } else
                {
                        new DgCracker_MsgPropagation(-1, child)
                }
        }
        override def toString = "{"+parent.toString+","+child.toString+"}"
}
object DgCracker_MsgTree
{
        def empty = new DgCracker_MsgTree(-1, scala.collection.immutable.Set())
}
```

// General Message Class

```
class DgCracker_GeneralMsg (val MsgIdentification : Option[DgCracker_MsgIdentification], val MsgTree :
Option[DgCracker_MsgTree]) extends CrackerMessageSize

{

        def getMessageSize =
MsgIdentification.getOrElse(DgCracker_MsgIdentification.empty).getMessageSize +
MsgTree.getOrElse(DgCracker_MsgTree.empty).getMessageSize

    override def toString =
"MsgIdentification:("+MsgIdentification.getOrElse("").toString+"),(MsgTree:("+MsgTree.getOrElse("").toSt
ring+")"

}



object DgCracker_GeneralMsg

{

        def apply(MsgIdentification : DgCracker_MsgIdentification) = new
DgCracker_GeneralMsg(Option.apply(MsgIdentification), Option.empty)

        def apply(MsgTree : DgCracker_MsgTree) = new DgCracker_GeneralMsg(Option.empty,
Option.apply(MsgTree))

}
```