LudoGraph: a Sampling Capable Cloud-Based System for Large-Scale Graph Processing
Based on the Pregel programming model

Marcin Biczak
LudoGraph: a Sampling Capable Cloud-Based System for Large-Scale Graph Processing
Based on the Pregel programming model

MASTER OF SCIENCE THESIS

For the degree of Master of Science in Computer Science at Delft University of Technology

Marcin Biczak

July 03 2013
Abstract

Growing data size in recent years pose new challenges in efficient processing of these datasets. Small and medium enterprises can not afford to use currently available large-scale solutions such as: supercomputers and large cluster sites. Graph datasets also grow in size and require special type of processing to extract analytics related data.

In this thesis we introduce our evaluation of large-scale graph processing platforms. We investigate each of the platforms pros and cons and we try to determine what makes a good large-scale graph processing platform.

We also design and implement prototype of LudoGraph, which is a large-scale graph processing platform with cloud capabilities and sampling features. LudoGraph also implements Pregel programming model. We believe that our platform with cloud capabilities will be able to provide required functionality for small and medium enterprises.

Furthermore, we perform graph sampling algorithms evaluation. In the algorithm evaluation we investigate the algorithms performance and the accuracy of the collected samples. This work is used as a foundation of a runtime sampling feature in our new LudoGraph platform.
Table of Contents

Acknowledgements xi

1 Introduction 1
   1-1 Motivation ..................................................... 2
   1-2 Research Questions .......................................... 3
   1-3 Approach ...................................................... 3
   1-4 Research and Technical achievements ...................... 4
   1-5 Thesis Structure ............................................. 5

2 Background and Related Work 7
   2-1 Graph .......................................................... 7
   2-2 The Pregel Programming Model ............................... 8
   2-3 Evaluated Large Scale Graph Processing Platforms .... 9
       2-3-1 Neo4j .................................................. 10
       2-3-2 Hadoop ................................................ 11
       2-3-3 Giraph .................................................. 13
   2-4 The YARN Large-Scale Data Processing Platform ....... 14
   2-5 Large-Scale Graph Processing Algorithms ............... 15
       2-5-1 Graph Properties Algorithm ......................... 15
       2-5-2 Breadth-first Search algorithm ...................... 16
       2-5-3 Connected Components Algorithm .................... 17
       2-5-4 Community Detection Algorithm ..................... 18
       2-5-5 Graph Evolution Algorithm .......................... 19
3 A Performance Study of Graph Processing Platforms 21
3-1 Experimental Setup ........................................ 21
3-1-1 Experimental Hardware .................................. 22
3-1-2 Experimental Datasets .................................. 22
3-1-3 Performance Metrics ................................... 23
3-1-4 Experiment Execution .................................. 24
3-2 Experimental Results ....................................... 24
3-2-1 Graph Properties Algorithm ......................... 25
3-2-2 Breadth-first Search Algorithm ...................... 26
3-2-3 Connected Components Algorithm .................. 28
3-2-4 Community Detection Algorithm .................... 29
3-2-5 Graph Evolution Algorithm ......................... 30
3-2-6 Data Ingestion .......................................... 32
3-2-7 Platform Overhead ...................................... 34
3-2-8 Resource Utilization ................................... 35
3-2-9 Scalability ............................................... 40
3-3 Summary .................................................... 43

4 LudoGraph Design & Validation 47
4-1 Goals of LudoGraph ....................................... 47
4-2 Overview of novelties in LudoGraph .................. 48
4-2-1 Cloud Capabilities ..................................... 48
4-2-2 Implementation of Pregel in LudoGraph ............ 48
4-2-3 Runtime Sampling and Custom Traversal .......... 49
4-3 LudoGraph Design ......................................... 49
4-3-1 LudoGraph and YARN .................................. 49
4-3-2 LudoGraph Components ................................ 50
4-3-3 Platform Bootstrapping ................................ 53
4-3-4 Job Execution ........................................... 53
4-3-5 Cloud Capabilities ..................................... 54
4-3-6 Pregel Programming Model Implementation Design 54
4-3-7 Runtime Sampling and Custom Graph Traversal Design 56
4-4 LudoGraph’s Prototype Validation .................... 56
4-4-1 Validation of Cloud Capabilities ...................... 56
4-4-2 Validation of Implementation of Pregel in LudoGraph 57
4-4-3 Validation of Custom Graph Traversal ............... 57
4-5 Summary .................................................... 59
# Table of Contents

5 **Graph Sampling Functionality in LudoGraph**

5-1 Sampling Algorithms ........................................ 61
  5-1-1 Random Walk ........................................ 62
  5-1-2 Metropolized Random Walk ........................... 62
  5-1-3 Unique Metropolized Random Walk .................. 63

5-2 Performance Evaluation of Sampling Algorithms .... 65
  5-2-1 Random Walk ........................................ 65
  5-2-2 Metropolized Random Walk ........................... 66
  5-2-3 Unique Metropolized Random Walk .................. 66
  5-2-4 Sampling Accuracy .................................... 68

5-3 Summary .................................................. 70

6 **Conclusion** ............................................. 73

7 **Future Work** ............................................ 77

A **Hadoop’s limitations for the Graph Properties algorithm execution** ........ 81
  A-1 Graph Properties MapReduce Implementation .......... 82
  A-2 Problem Definition ...................................... 82
  A-3 Hadoop Tuning ......................................... 83
  A-4 Nodelerator++ and Nodelerator+2 Algorithms .......... 85
  A-5 New Algorithm .......................................... 86
  A-6 Conclusion ................................................ 87

**Bibliography** ............................................... 89
Table of Contents
List of Figures

2-1 A directed graph example. ........................................ 8
2-2 Pregel superstep execution. ................................ 9
2-3 Graph representation in Neo4j ............................... 10
2-4 Neo4j graph traversal. ......................................... 11
2-5 Overview of MapReduce execution. ......................... 12
2-6 Overview of Giraph design. ................................ 14
2-7 YARN Architecture. ........................................... 15
2-8 Modularity Function. .......................................... 18

3-1 Graph properties performance. ................................. 26
3-2 Breadth-first search performance. ............................ 28
3-3 Connected Components performance. ....................... 29
3-4 Community Detection performance. ......................... 31
3-5 Forest Fire Model performance. ............................... 33
3-6 Data Ingestion. ................................................ 33
3-7 Platform execution analysis. .................................. 34
3-8 CPU utilization by platform’s Master. ....................... 36
3-9 Memory utilization by platform’s Master. ................. 37
3-10 Network utilization by platform’s Master. ................ 37
3-11 CPU utilization by platform’s Worker. ..................... 38
3-12 Memory utilization by platform’s Worker. ............... 39
3-13 Network utilization by platform’s Worker. ............. 39
3-14 Horizontal scalability of evaluated platforms. ........... 41
3-15 Vertical scalability of evaluated platforms with the use of Citation_FCF dataset. 42
3-16 Vertical scalability of evaluated platforms with the use of Friendster dataset. . . . 43
4-1 LudoGraph Components. YARN is the basic resource management block. . . . . . 50
4-2 Platform bootstrapping. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
4-3 High-level LudoGraph’s Pregel implementation. . . . . . . . . . . . . . . . . . . 55
4-4 LudoGraph’s (with cloud capabilities) and Giraph’s execution. . . . . . . . . . 57
4-5 LudoGraph’s and Giraph’s Pregel implementation execution. . . . . . . . . . . 58
4-6 LudoGraph’s (with Custom Traversal) and Giraph’s execution. . . . . . . . . . 58
5-1 Random Walk performance. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
5-2 Metropolized Random Walk performance. . . . . . . . . . . . . . . . . . . . . . . 67
5-3 Unique Metropolized Random Walk performance. . . . . . . . . . . . . . . . . . . 68
5-4 Average sample uniqueness. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
5-5 Average vertex degree. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
5-6 Average local clustering coefficient. . . . . . . . . . . . . . . . . . . . . . . . . . . 70
5-7 Sample Degree Distribution. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
A-1 Size of data to be transmitted by a vertex. . . . . . . . . . . . . . . . . . . . . . . 83
A-2 Tuning configurations results. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
List of Tables

3-1 DAS4 sites hardware configuration .................................................. 22
3-2 Experimental datasets ................................................................. 23
3-3 Graph Properties Algorithm output of experimental datasets ............... 25
3-4 Longest path from a single source ................................................... 27
3-5 Communities results, based on Giraph results ..................................... 30
3-6 Graph Evolution results, based on Giraph results ................................. 32
3-7 Horizontal scalability configuration .................................................. 40
A-1 Number of vertices with degree value within different ranges ............... 83
I would like to thank my wife for her support and encouragement during my work on this thesis. Also I would like to thank my thesis supervisor Prof. Alexandru Iosup for giving me the opportunity to pursue my goals during research for this thesis. I would also like to thank Prof. Alexandru Iosup for his patient and support without which this thesis would never come to being. I also thank DAS4 system administrators Mr Paulo Anita and Mr Kees Verstoep for their help in the time of need and support. I thank Otto Visser and Yong Guo for their help in putting the thesis together, their comments and remarks were valuable and allowed me to improve my thesis. I would also like to thank Bogdan Ghit, Jianbin Fang, and Jie Shen for their advices. Furthermore I would like to thank members of PDS and Graphitti group for countless interesting discussions.

Delft, University of Technology

Marcin Biczak

July 03 2013
“Ubi tu ibi ego.”

For my Wife
The size of datasets continuously grows [2] and this trend has lead to the creation of a field of computer science known as *Big Data Analytics*. In Big Data Analytics, dataset sizes are measured in terabytes or even petabytes. Such a large volume data can not be processed by a single desktop machine in a feasible time e.g., minutes or few hours. These datasets can be processed only by *supercomputers* or by *clusters* of commodity computers. Although supercomputers have been used for many years, they are still very expensive and not all companies/organizations can afford buying such hardware. Thus, cloud/cluster computing has gained significant attention in the recent years. Cloud computing offers customers the possibility of leasing a number of computers for a predefined period of time and perform computation on them. Such a solution can be much cheaper short-term than buying a supercomputer. Clusters are similarly designed to clouds; however, a cluster is usually owned by a single entity or shared amongst many closely connected organizations (for example universities), and may thus remain underutilized and relatively expensive. Growing demand for processing Big Data in a feasible time has given rise to many distributed platforms which are capable of processing large volumes of data in parallel.

In Big Data Analytics the volume of the data is not the only concern. Another important aspect is the data representation. Not all information can be represented as disjoint units of information (records, arrays); we define disjoint unit of information as a unit of information which has no knowledge regarding the existence of other units of information. For example, while a data record can represent a single unit of information, the set of records can not fully represent existing connections between these records. Thus, many datasets are stored as a *Graph* structure. The graph structure gives the flexibility of storing any information unit as well as a good representation of relations between the information units. Many companies use graphs to represent their data collection, for example *Facebook* [3] uses graph data structures to represent its users’ interaction as well as preferences. Many other companies also store part of their data in a graph structure; among them are *Google* [4] and *LinkedIn* [5].

This thesis focuses on graph processing as an example of Big Data Analytics, using cloud computing.
1-1 Motivation

Many solutions have been created to process large volume data either in clouds or clusters. However very few of them have addressed graph data as many widely used distributed platforms do not take under consideration relations between information units. One of the most popular platforms currently used in processing Big Data, the *Hadoop* platform is based on a *MapReduce*\[6\] programming model created by *Google*. The *MapReduce* programming model works very well for processing disjoint units of information. However, processing graph structures with the use of this model is not intuitive and requires additional work, as it was not designed specifically for graph processing purposes.

The lack of a programming model specifically designed for graph processing was addressed by *Google* with the creation of the *Pregel*\[7\] programming model. However, although the *Pregel* programming model was created, there was no publicly available platform which could execute the *Pregel* model. With time, several open-source projects have dedicated themselves to create a distributed platform based on the *Pregel* programming model; some of these projects are *Giraph*\[8\], *Phoebus*\[9\] and *Bagel*\[10\]. Other programming models and implemented graph processing platforms also exists \[Trinity\[11\], *Hama*\[12\], *Signal/Collect*\[13\]\]. Although many data processing platforms can be used to process graphs, no in-depth performance evaluation has been performed to determine which platform provides best performance. Being able to determine which platform performs better would help potential users choose the platform that best suites their needs.

None of the publicly available implementations of the *Pregel* model supports cloud computing. This might be an issue for certain users especially the efficiency-oriented research labs and small-an-medium-enterprises, as it is hard to determine how many resources will be required to perform the computation in a reasonable amount of time. Delivering a computational platform based on the *Pregel* model and capable of utilizing clouds could potentially help many companies in their data processing effort. A key element in supporting cloud-based graph processing is the ability to provision (lease/release) computational resources on demand. To our knowledge no platform exists which supports both the *Pregel* model and cloud provisioning solutions.

In some cases the graph dataset size is too large making the computation unfeasible with available resources. Either program execution is too long or the available memory is insufficient which can lead to errors in platforms which perform computation in-memory only. *Graph sampling* algorithms could potentially solve this problem by providing a representative sample of the whole graph. We define the *representative sample* of a graph as a sample which holds all properties of the original graph. In some cases unique sample might be required rather than non-unique sample to perform graph analysis; we define a unique sample as a sample which holds only unique vertices. Collecting non-unique sample and later filtering duplicated vertices may not provide representative and unbiased sample; sample representativeness may me lost during the filtering process. Although graph sampling is an important topic in the graph processing field, currently no platform to our knowledge supports graph sampling, without the need of implementing the sampling algorithm.
1-2 Research Questions

Addressing the issues mentioned before, our thesis focuses on the following research questions:

1. What is the performance of current graph processing platforms, for typical real datasets algorithms?

2. Is it possible to design a graph processing platform with cloud capabilities and sampling features?

3. What are the accuracy, the performance, and the ability to provide unique vertices of graph sampling algorithms, for typical real datasets?

1-3 Approach

In this section we present the (research) approach we have taken to answer each of our research questions.

For the performance evaluation of graph processing platforms, we have taken an empirical research approach. First, we chose which platforms to evaluate. The choice of a first platform was rather straightforward as, currently there is only one popular platform supporting the Pregel programming model, called Giraph. Giraph was created by the Apache Software Foundation and is being still developed; thus, for our research we have used a (advanced) development version of the platform. We have not used the previous stable version, because it offers very little functionality for implementing graph processing algorithms; it was developed only as a foundation for future platform development. Hadoop is another platform which we have chosen for our evaluation; the MapReduce-based. Hadoop is a very popular Big Data Analytics platform, which is mature and widely used. Hadoop represents the distributed platforms which were not designed specifically for graph processing, but are capable of executing graph processing algorithms. Giraph and Hadoop were design to work in a distributed environment. To make our evaluation complete, we selected third platform that runs in a non-distributed environment: Neo4j. Neo4j is a sequential desktop NoSQL graph database, designed and optimized for graph storage and processing. The current version supports only single-machine computation. By comparing the stand-alone commodity machine (Neo4j) with a cluster of machines (Giraph and Hadoop) we can show experimentally the performance increase and potential benefits of processing graphs in distributed environment. Once the platform selection process was finished, we have chosen a number of algorithms to act as our evaluation benchmark. These algorithms represent some of the most commonly used classes of algorithms in the graph processing field. Furthermore, the datasets used in our experiments represent wide range of information categories such as social networks, logistics, online gaming, and co-purchased products structure.

To answer the second research question we have designed a new large-scale graph processing platform with cloud capabilities and sampling features, LudoGraph. Our graph processing platform was designed based on findings from the performance evaluation experiments, for example: algorithms implementation usability, platform performance, etc. Our main goal is to design LudoGraph such that the platform shall posses all beneficial design choices of evaluated platforms with none of the drawbacks.
To answer the question about sampling algorithm, we have followed similar steps as for the performance evaluation. We have selected two algorithms to compare. However, during this process we have noticed that neither of the algorithms provides a unique sample. Thus, we have created a new sampling algorithm which by design provides a unique sample. We compare these algorithms in terms of execution time and accuracy of the sample they provide.

1-4 Research and Technical achievements

During our work on this thesis we have accomplished the following research achievements:

1. Answered first main research question and added new knowledge (see Chapter 3).
2. In depth research of evaluated platforms (see Chapter 3).
4. Answered second main research question and added new knowledge (see Chapter 4).
5. Investigated YARN design potential (see Chapter 4).
6. Some of our work on LudoGraph platform will be extended as ICDE’14 article.
7. Answered third main research question and added new knowledge (see Chapter 5).
8. Investigated sampling algorithm performance in terms of execution time and output sample accuracy (see Chapter 5).

Furthermore, a side research achievements we have also realized the following technical achievements:

1. Performance evaluation of 3 platforms, 5 algorithms, and 18 datasets, leading to experimental space with 270 points (see Chapter 3).
2. In depth investigation of Hadoop platforms data processing mechanism (see Appendix A).
3. Investigated platform management overhead, data ingestion processes, resource utilization, and platform scalability of evaluated platforms (see Chapter 3).
4. Designed new large-scale graph-processing platform with cloud capabilities and sampling features; the LudoGraph platform (see Chapter 4).
5. Implemented prototype of LudoGraph platform (see Chapter 4).
6. Some of our work on LudoGraph platform will be extended as an open-source.
7. Sampling algorithms execution analysis on evaluated platforms (see Chapter 5).
8. New sampling algorithm which provides unique sample (see Chapter 5).
1-5  Thesis Structure

The reminder of this thesis is structured as follows. In Chapter 2 we explain background knowledge required to understand the reminder of the thesis. In Chapter 3 we present our performance evaluation of the chosen platforms. In the performance evaluation, we present findings regarding performance, platforms overhead, data ingestion, scalability, and resource utilization. Some of this work was included in *How Well do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis: Extended Report* [14] article. In Chapter 4 we introduce a high level design of the new graph processing platform the *LudoGraph*. Some of this work to be extended as open-source and as *ICDE’14* article. In Chapter 5 we presents performance of the sampling algorithms and the sampling accuracy of evaluated sampling algorithms. In Chapter 6 we conclude our thesis. In Chapter 7 we presents possible future improvements of our newly designed platform, evaluation methodology, and our new sampling algorithm, which provides unique sample.
Chapter 2

Background and Related Work

In this Chapter we explain the basic concepts and terminology required to understand the thesis. We start by defining what is Graph data structure and explain different graph types, in Section 2-1. In section 2-2 we introduce Pregel a popular programming model for graph processing. We also briefly describe each of the evaluated platforms in Section 2-3. In section 2-4 we describe YARN platform which will be a key implementation block for our newly designed LudoGraph platform (see Chapter 4). In section 2-5 we present the algorithms which were used in our performance evaluation.

2-1 Graph

A graph is a mathematical concept to represent a set of objects, where pairs of objects are connected by links. The objects are represented by mathematical abstractions called vertices, and the links that connect pairs of vertices are called edges. Figure 2-1 presents a graph example, where the edges represent the non-transitive connection, that is, a connection for which if a link exists in one direction, the converse is not necessary true. For example, directed edges can represent the link “supervises” between two employees; if A supervises employee B, it is unlikely that employee B supervises employee A.

Graphs can be divided into multiple categories, these categories define how vertices are connected by edges. Some of the categories are:

- directed graphs - graphs in which edges connecting vertices have direction. In these graphs we can distinguish two types of edges. The outgoing edges are edges which are directed outwards from a current vertex (called source vertex). The incoming edges are edges which are directed inwards from a current vertex (called destination vertex). If vertex A has an out-edge to vertex B, this implies that vertex A is connected to vertex B, however this does not imply that vertex B is connected to vertex A.
Background and Related Work

Figure 2-1: A directed graph example.

- undirected graphs - graph in which edges connecting vertices have no direction. These graphs consists only from outgoing edges. If an edge exists between vertices $A$ and $B$, this implies that these two vertices are connected.
- mixed graphs - graphs in which some edges may be directed and some may be undirected.
- multigraphs - graphs in which same pair of vertices can have multiple outgoing or incoming edges, also a vertex can be connected to itself (loop).
- weighted graphs - graphs in which a number (weight) is assigned to each edge.
- hypergraphs - graph in which a single edge can connect multiple vertices.

In our thesis we study both directed and undirected graphs. We do not consider other types of graphs or weights of vertices or edges.

2-2 The Pregel Programming Model

The Pregel programming model [7] was created by Google, it is based on a concept that each vertex in the graph acts as a single independent actor. Actors can communicate with each other by sending messages. Pregel is in the Bulk Synchronous Parallel (BSP) model of distributed platforms. A BSP computation proceeds in a series of global supersteps. Each superstep consists of three phases:

- concurrent computation - each vertex actor executes code
- communication - vertices communicate with each other
- barrier synchronisation - each worker waits until all workers reach synchronization barrier, before executing next superstep.

Termination condition is reached once all actors vote to halt computation.
2-3 Evaluated Large Scale Graph Processing Platforms

In this Section we will briefly introduce large-scale data processing platforms which we have chosen to evaluate.

Large-scale graph processing has gained significant attention in the last few years. With companies such as Google, Facebook and Twitter modern day industry is capable of generating large-scale graphs on a scale never seen before. Thus many researchers focus on the challenges of processing these graphs. In recent years many graph processing platforms have been created. From this wide variety of different platforms we have chosen three, which in our opinion are mature and well received by the community.

The goal of this performance study is to perform a number of experiments on each platform and evaluate its capability of processing large-scale graphs. We did not only concentrate on the cluster-based platforms, we also reached further into different technologies. In the end we have chosen these three platforms:

- **Neo4J** - desktop platform, NoSQL, graph database, version 1.5
- **Hadoop** - cluster platform, generic large-scale data processing platform, version 0.20.203
- **Giraph** - cluster platform, large-scale graph processing specialized platform, development version 0.2

Platforms of our choosing represent a wide spectrum of technologies available for large-scale graph processing. The generic large-scale data processing platform was designed to process any type of data and is not focusing exclusively on graph processing. Specialized large-scale graph processing platforms were designed specifically for processing graphs. By evaluating these platforms we were capable of answering these questions:

- are modern desktop computers any good for large-scale graph processing?
10 Background and Related Work

- is a specialized large-scale graph processing platform more useful than non-specialized platforms?
- what determines a good graph processing platform?

In the following Sections we will briefly present each of the platforms. In Section 2-3-1 we present Neo4j platform, in Section 2-3-2 we present Hadoop platform, and Section 2-3-3 is dedicated to the Giraph platform.

2-3-1 Neo4j

Neo4j [16] created by Neo Technology is an open-source NoSQL graph database implemented in Java. NoSQL is a class of database management systems, which is not built on top of table structures. Neo4j stores data as a graph, where the data is not limited to graphs only. Each vertex or edge can hold additional information, making the graph structure a high level storage structure.

![Graph representation in Neo4j](image-source.png)

Figure 2-3: Graph representation in Neo4j (image source [16]).

Neo4j was designed as a single machine database, however its creators claim that it can scale to multiple machines. To achieve multiple machine scale, users of Neo4j have to implement communication between these machines as well as manage partitioning, consistency etc. In our opinion though multimachine scale is possible, it would require a significant amount of additional work, besides the application implementation. For our evaluation we have executed Neo4j version 1.5 on a single machine.

Every stored graph consists of relationships and vertices, where the relationships represent edges between vertices. Neo4j supports different types of relationships in a single graph, however it does not support multigraphs based on a single type of relationship. Every vertex and relationship has a number of user defined properties, which can be used as a value storage for a particular vertex or edge; for example edge weight.

Neo4j processes graphs by traversing all vertices, with the use of either the Breadth-first search [17] or Depth-first search [18] traversal algorithm. To start graph traversal a program
has to define a special reference vertex. This vertex is not a part of the original graph, but an additional artificial vertex which is added to the graph structure and acts as a starting point of the graph traversal. All graph operations are performed as ACID transactions. ACID transaction holds the following properties: atomicity, consistency, isolation and durability.

User implements the application from the “walker” perspective; we define walker as an abstract entity which traverses vertices in the graph. This is achieved by specifying a graphs traversal loop, which at each iteration returns the currently traversed vertex. This level of abstraction makes graph processing algorithms relatively easy to implement, as every algorithm can be intuitively expressed by visiting every accessible vertex in the graph. Furthermore Neo4j uses a 2 level caching mechanism to improve its performance. The file buffer cache caches the storage file data in the same format as it is stored on the durable storage media. The object cache caches the nodes, relationships and properties in a format that is optimized for high traversal speeds and transactional writes.

2-3-2 Hadoop

Hadoop [19] created by the Apache Software Foundation is an open-source, generic large-scale data processing platform. It is based on the MapReduce [6] programming model, created by Google. The MapReduce programming model processes input data and divides it based on key/value pairs. Initial data processing is performed by the Mappers, which attach a specific key to each value in the dataset. The Reducers are responsible for gathering all values assigned to a certain key, single Reducer can process multiple keys.

All data used by Hadoop is stored in the Hadoop Distributed File System (HDFS). HDFS is not a part of Hadoop, although it is being used by it and the platform will not work without the HDFS.
Datasets which are stored in the HDFS are divided into $N$ blocks (chunks) of similar size. Each of these blocks is used as an input for a single Mapper. In case there are more blocks than Mappers, each Mapper can process multiple blocks sequentially. Hadoop’s typical job execution consist of few phases, these are:

- **Read phase** - responsible for reading data blocks for **Map phase**
- **Map phase** - responsible for attaching keys to values
- **Shuffle/Copy phase** - responsible for copying keys and values from Mapper nodes to appropriate Reducer nodes, based on the key value. (not depicted in Figure 2-5)
- **Reduce phase** - responsible for processing all values with the same key
- **Write phase** - responsible for writing output to HDFS

Between the **Map** and the **Shuffle/Copy** phase, a **Spill** process might occur on the Mappers. This process is executed when the in-memory data buffers fill threshold is reached. When this scenario occurs, part of the data is spilled to disk. Furthermore to group all values based on their key, a **sort-merge** process in executed between the **Copy** and **Reduce** phase, on every Reducer node. The **sort-merge** process is grouping all values received from all Mappers based on the key value.

**MapReduce** is executed in a cluster environment, where part of the available computational nodes act as Mappers, while the rest act as Reducers. The execution process is governed by the **Master** node, which is responsible for managing input/output for each of the workers,
as well as job execution. Furthermore the Master node is also responsible for scheduling submitted jobs to available workers.

Hadoop utilizes MapReduce design in a combination with underlying network file system the HDFS. Hadoop consists of the following components:

- Job Tracker - executed on the Master node, responsible for managing platform and job execution.
- Task Tracker - executed on the Worker nodes, responsible for Task execution, either Mapper or Reducer.
- Name Node - executed on the Master node or a dedicated worker node, is responsible for indexing data blocks in HDFS.
- Secondary Name Node - executed on one of HDFS nodes, responsible for taking HDFS snapshots.
- Data Node - executed on HDFS nodes, stores data blocks.
- HDFS - Hadoop’s network file system.

2-3-3 Giraph

Giraph [8] created by the Apache Software Foundation is an open-source, specialized large-scale graph processing platform. It is based on the Pregel [7] programming model, created by Google.

Giraph utilizes the Pregel design as well as Hadoop’s design. However, there are some significant differences between Hadoop and Giraph. The single biggest difference between Hadoop and Giraph is the fact that Giraph is executed in-memory only. Which speeds-up the execution significantly, but can also lead to out of memory errors for datasets that do not fit into the cluster memory. Giraph’s job is just a MapReduce job, which consists only from the Map phase. Within a single execution of the Map phase, Pregel’s supersteps loop is being executed.

Furthermore Giraph’s master is not designed in the same way as Hadoop’s. It rather uses the ZooKeeper [20] for this purpose, which acts as an additional Master, responsible for steps synchronization. Giraph also uses Hadoop’s HDFS file system to store and read datasets.

A typical Giraph job execution is a combination of Hadoop’s solutions with Pregel programming model abstraction. The dataset is read from HDFS and divided into blocks, from these blocks vertices are constructed and assigned to workers. Then the job executes a number of supersteps, with communication between vertices. Once there are no more messages to process and all vertices voted to halt, job execution is finished and the output is written to HDFS.
2-4 The YARN Large-Scale Data Processing Platform

YARN (also known as Hadoop YARN, MRv2 and Hadoop NextGen) [21] is a redesigned original Hadoop platform. Authors of the new platform have redesigned the original platform at the conceptual level. The new design allows new programming models and software applications to be easily implemented and run in the cluster environment. Our newly designed large-scale graph processing platform the *LudoGraph* was designed to work under supervision of the *YARN* platform.

The *YARN* platform consists of the following components:

- **Resource Manager (RM)** is responsible for starting the *Application Masters* and granting resources to it. It is the true master of the *YARN* platform. It consists of two components: *Scheduler* and *Application Manager*. The *Scheduler* is responsible for scheduling submitted jobs and the *Application Manager* is responsible for managing the life cycle of *Application Masters*.

- Application Master (AM) is a per-application master. This component is responsible for managing job execution. It requests resources required for the job and starts tasks on assigned containers and reacts to task failures.

- Node Manager (NM) is a per-machine manager responsible for launching containers allocated by the Scheduler for a job. It is also responsible for ensuring that none of the containers exceeds resources allocated to it, if any container does so it will be immediately terminated by the Node Manager.

- **Container** is an abstract component which incorporates elements such as memory, cpu, disk, network etc. It is an environment in which tasks are being executed. A single
computational node consists of many containers. The current implementation of the YARN platform specifies containers based on memory size only.

Figure 2-7: YARN Architecture (image source [21]).

In the new design the functionality of Job Tracker is split between RM and AM. Task Tracker can be viewed as NM. The Hadoop’s distributed file system HDFS has not changed significantly. Furthermore the YARN platform utilizes the functionality of ZooKeeper [20] to store the state of platform components. This allows the platform to quickly recover from any failure.

2-5 Large-Scale Graph Processing Algorithms

In this section we provide an overview of algorithms which we have used in our experiments. Section 2-5-1 presents algorithm used to calculate graph’s global properties. Section 2-5-2 is presents BFS algorithm. Section 2-5-3 introduces connected components algorithm, followed by Section 2-5-4 which describes community detection algorithm. Finally in Section 2-5-5 we present graph evolution algorithm.

Experiments results are presented in Chapter 3.

2-5-1 Graph Properties Algorithm

Graph Properties algorithm (GP) calculates graph’s basic properties. We define the basic properties of the graph as:
- number of vertices, that is, a count of all the vertices present in the graph.

- number of edges, that is, a count of all the edges present in the graph.

- average Local Clustering Coefficient (LCC), that is an average value calculated for all vertices in the graph. An LCC value quantifies how close vertex neighbours are to being a clique (complete graph, a graph in which each vertex is connected to all other vertices by unique edges).

We think that knowing these properties of a graph can be beneficial for its further processing as these graph metrics define how big the graph is and how dense it is. By knowing the graph size and its density, users can better understand how long an algorithm may execute and what type of computer resource will be utilized the most by the algorithm.

Algorithm 1 shows the pseudo-code of the GP. Algorithm takes as an input a vertex based representation of the graph; each line of the input holds a single vertex and all its neighbours ids. At the beginning of the algorithm, each vertex sends its id and its neighbours ids to all of its neighbours. Once every vertex receives its neighbours data it can compute its Local Clustering Coefficient (Algorithm 1 line 10). The local clustering coefficient is a proportion of links between the vertices within its neighbourhood divided by the number of links that could possibly exist between them. To calculate the total number of vertices and edges, each vertex updates global variables (Algorithm 1 lines 6-7).

Algorithm 1: Graph Properties Algorithm

<table>
<thead>
<tr>
<th>Input: Vertex-based dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: Graph’s average LCC and number of vertices, edges</td>
</tr>
</tbody>
</table>

```plaintext
1 verticesTotal = 0;
2 edgesTotal = 0;
3 avgLCC = 0;
4 foreach Vertex v do
5     SendMyOutEdges(EdgesDstList, NeighbourList);
6     verticesTotal += 1;
7     edgesTotal += getDegree(v);
8     VertexNeighbourhood = CreateVertexNeighbourhood([EdgeList]);
9     counter = CountEdgesBetweenNeighbours(VertexNeighbourhood);
10    avgLCC += CalculateLCC(counter, vertexDegree);
11 avgLCC = avgLCC / verticesTotal;
```

### 2-5-2 Breadth-first Search algorithm

The Breadth-first Search algorithm (BFS) [17] is an algorithm used in graphs/trees traversal. The algorithm itself can serve many purposes (shortest path, minimum spanning tree, etc.), however it was designed as traversal mechanism of the graphs/trees data structures.

The Breadth-first Search algorithm is a widely used algorithm in graph processing, it is often an underlying mechanism of a more complex algorithms. It can be used to compute many graph properties such as element search, distance calculation, diameter calculation, shortest
path, longest path etc. Because BFS executes a lightweight CPU computation in a loop (each algorithm step is a different iteration) it will allow us to understand what is the management overhead of evaluated platforms.

BFS algorithm starts from an initial vertex and traverses the graph structure, until the destination vertex is found or all vertices are visited. It achieves its goal by visiting all previously unvisited reachable neighbours of a current vertex. This process is repeated until the whole structure is traversed or an other stopping condition is fulfilled. In our research we have used the BFS algorithm to determine all vertices which can be reached from a pre-defined initial vertex. The pseudo-code for the algorithm is presented in Algorithm 2.

Algorithm 2: Breadth-First Search algorithm

Input: Vertex-based dataset and initial vertex N
Output: Traversed graph

1. queue = N.getNeighbours();
2. while queue not empty do
3.     vertex = queue.dequeue();
4.     foreach vertex : vertex.getNeighbours() do
5.         if vertex not visited then
6.             enqueue vertex onto queue

2-5-3 Connected Components Algorithm

In many cases graph datasets contain multiple disjoint graphs. The connected components algorithm divides these disjoint graphs into different datasets. Bin Wu et al. created a MapReduce version of the connected components algorithm [22], which we have used as one of our experimental algorithms.

We have chosen this algorithm, due to its nature of solving common problems in a graph processing field. Also this algorithm was specifically designed for the MapReduce programming model. With an implementation of this algorithm for other platforms, we would be able to determine if a platform is flexible and easy to translate algorithms to. As many graph processing algorithms are sequential in their design, executing them on a cluster/cloud platform requires their translation into a distributed version.

The algorithm requires the input graph to be represented in a special format. The format is: vertex$_i$ \(\text{at } Tag$_i$ \& vertex$_j$, vertex$_k$, ..., vertex$_n$. Where vertex$_i$ is a vertex’s id, Tag$_i$ represents the label of the vertex and vertex$_j$, vertex$_k$, ..., vertex$_n$ represents vertex’s neighbors. The algorithm uses Label Propagation Algorithm (LPA) [23]. A sequential description of the
Connected Components algorithm is presented in Algorithm 3.

**Algorithm 3**: Connected Components [22].

**Input**: Algorithm specific input format

**Output**: Connected Components

1. **foreach** vertex **do**
   2. vertex.label = vertex.id;
3. **while** labelChanged **do**
   4. labelChanged = false;
5. **foreach** vertex **do**
   6. smallestLabel = vertex.retrieveNeighboursLabel();
   7. if smallestLabel < vertex.label **then**
   8. vertex.label = smallestLabel;
   9. labelChanged = true;

### 2-5-4 Community Detection Algorithm

Community detection algorithms detect communities within the graph structure. According to M. Newman et al. [24] a community is a group of vertices which are highly interconnected with vertices belonging to the same community, having only a small fraction of edges connected to vertices outside of the community. Newman et al. have defined a measure of the goodness of communities called *modularity function*. The function is defined as follows:

\[
Q = \sum_{c \in C} \left( \frac{I_c}{E} - \left( \frac{2I_c + O_c}{2E} \right)^2 \right)
\]

*Figure 2-8: Modularity Function (source [25]).*

where \( I_c \) represents the total number of internal edges within a community, \( O_c \) is the number of edges which other end is outside of the community and \( E \) is the total number of edges in a graph.

Community detection algorithms try to maximize the *modularity* value for each community they detect. The higher the *modularity* value of a community the better its quality is according to Newman et al. community definition.

Community detection is a very important topic in social networks as users of these networks tend to group together forming a community. Within these communities users can exchange ideas, cooperate together or discuss certain topics. It can be reasonably assumed that users mostly interact with their friends belonging to the same community. Communities are also very important in the gaming industry. As games shift their focus from singleplayer to multiplayer (some games are even multiplayer only), communities of players play an increasing role in the game life cycle. Multiple players from all over the world cooperate or compete together in various tournaments. Community detection algorithms can improve the quality
of these tournaments by detecting existing communities and pairing players with similar skill set.

In our experiments we have used the Label Propagation Algorithm (LPA) [25] designed by Ian X.Y. Leung et al. The algorithm is based on a label propagation mechanism, which acts similar to the epidemic protocols [26]. In our research we have also experimented with another LPA based community detection algorithm designed by Usha Nandini Raghavan et al. [23] however the algorithm did not produce stable results. In Algorithm 4 we present pseudo-code for the Leung et al. algorithm.

Algorithm 4: LPA algorithm [25].

Input: Vertex-based dataset and $\delta$ parameter  
Output: Vertices with community label 

1 foreach vertex do
2  vertex.label = vertex.id;
3 while labelChanged do
4  labelChanged = false;
5  foreach vertex do
6   NewLabel = chooseLabel(receivedLabels);
7    ifNewLabel ! = currentLabel then
8      currentLabel = NewLabel;
9      labelChanged = true;
10     labelScore = updateLabelScore();
11     foreach neighbour do
12       send(currentLabel, labelScore);

The algorithm during its execution assigns to each community a label and each label is assigned a score value. A new label is selected based on a cumulative score which is the label score value multiplied by a comparable characteristic, for example label’s sender degree. After receiving labels from step $T - 1$, each vertex selects a new label based on a maximal cumulative score of all labels it received; scores of the same labels are aggregated. The further the community label traverses the graph from its source, the lower is its score; as each vertex in its path decreases the score value according to the $\delta$ parameter (unless the new label is equal to the current one).

This mechanism called epidemic spread control prevents large communities from dominating the entire network. If the algorithm would not use this mechanism a single highly connected vertex could dominate a large portion of the graph (or even dominate the entire network), as its cumulative score would be very high.

2-5-5 Graph Evolution Algorithm

Graph evolution algorithms are part of the graph processing field. The main objective of any graph evolution algorithm is to “predict” the future graph structure. Different algorithms “predict” different aspects of a graph structure (new vertices, new edges, future cliques etc.). An accurate algorithm not only can predict how a graph structure will evolve over time, but can also help to prepare for these changes (for example data size increase).
opinion graph evolution is an important topic in the field of large-scale graph processing. Most probably social-media oriented companies use graph evolution algorithms to predict future evolution of a product.

Leskovec et al. [27] designed a graph evolution algorithm called, *Forest Fire Model* (FFM). The FFM algorithm is a model, which tries to predict how a new vertex will join the graph. The output of the algorithm (modified graph with new vertices and edges) is based on two properties of random graphs over time. These properties are: *densification law* and *shrinking diameter*. The densification law states that graphs over time become more dense; the average vertex degree increases. The shrinking diameter law states that the graph diameter decreases over time as vertices become better connected. In Algorithm 5 we present pseudo-code for the FFM algorithm.

**Algorithm 5: Forest Fire Model [27]**

**Input:** Vertex-based dataset, p and r defined by user, v - number of new vertices, stepThreshold - stopping condition

**Output:** Modified graph structure

```plaintext
1 while counter < v do
2    step = 1;
3    ambassadors = chooseRandomAmbassador();
4    createEdge(ambassador);
5    while step < stepThreshold do
6        foreach ambassadors do
7            x = geometricallyDistributedMean(1 − p);
8            y = geometricallyDistributedMean(1 − rp);
9            createOutLinks = 0;
10           createInLinks = 0;
11           while createdOutLinks < x do
12               ambassadors = createOutLinks(step, ambassadors);
13               createOutLinks++;
14           while createdInLinks < y do
15               ambassadors = createInLinks(step, ambassadors);
16               createInLinks++;
17           step++;
18           counter++;
```

Each new vertex connects to *initial ambassador* at random, an ambassador is an already existing vertex in the graph dataset. From the neighbours of the initial ambassador algorithm creates x out-edges and y in-edges to these neighbours. If new vertex can not create x or y new edges it creates as much as it can. In the next step connected neighbours of the ambassador become the ambassadors themselves. This process is continued until the stopping condition is reached.
Chapter 3

A Performance Study of Graph Processing Platforms

In this chapter we answer the first research question (see section 1-2). In Section 3-1 we describe in detail the datasets we have used for our experiments as well as our experimental configuration and metrics used in our evaluation. In section 3-2 we present results of all experiments we have performed for our platforms evaluation. In section 3-3 we summarizes this chapter. Some of this work was included in How Well do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis: Extended Report [14] article.

3-1 Experimental Setup

To compare different platforms, we have chosen a set of algorithms and metrics based on which we will formulate our comparison. To make our research as thorough as possible, we have chosen a number of graph processing algorithms. The experimental algorithms had to represent real life scenarios and had to utilize different types of executions, to fully evaluate platform capabilities. We have chosen these algorithms as they calculate different properties, have different computation and data access patterns (for details see Chapter 2 Section 2-5). These are the five algorithms that we have chosen for this task:

1. Graph Properties Algorithm
2. Breadth-first search Algorithm
3. Connected Components Algorithm
4. Community Detection Algorithm
5. Forest Fire Model Algorithm (graph evolution)

With these algorithms, we perform an evaluation of our selected platforms (see Section 1-2).
3-1-1 Experimental Hardware

In our performance study we have used the DAS4 (the Distributed ASCI Supercomputer 4) [1], which is a six-cluster wide-area distributed system. During our experiments we have used the DAS4 Delft University of Technology site and Vrije Universiteit site; in our understanding a site represents a single cluster. Both the Hadoop and the Giraph experiments were performed either on the delft’s site or on the vu’s site. Neo4j’s experiments were performed on a commodity Desktop computer. In Table 3-1 we present detailed description of each sites available hardware configuration.

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Type</th>
<th>Speed</th>
<th>Memory</th>
<th>Local HDDs</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>VU</td>
<td>dual quad-core</td>
<td>2.4 GHz</td>
<td>24 GB</td>
<td>1 TB</td>
<td>IB and GbE</td>
</tr>
<tr>
<td>TUD</td>
<td>dual quad-core</td>
<td>2.4 GHz</td>
<td>24 GB</td>
<td>1 TB</td>
<td>IB and GbE</td>
</tr>
<tr>
<td>Desktop</td>
<td>dual quad-core</td>
<td>2.33 GHz</td>
<td>16 GB</td>
<td>300 GB</td>
<td>not used</td>
</tr>
</tbody>
</table>

In Hadoop’s related experiments we have used 20 workers, which were divided into two groups (10 mappers and 10 reducers). Furthermore we have allocated an additional node to act as a Master and another additional node to act as a speculative node (node which will support speculative execution [28]). Thus all datasets which were used in Hadoop’s experiments were divided into 10 blocks which were then distributed among mappers. Such a division allowed us to execute the Map phase on each worker only once, thus the Reduce phase workers were not waiting for a single Mapper to finish.

The Giraph’s experimental configuration was different than Hadoop’s as its execution model is different. Thus Giraph’s experimental configuration consists of 20 workers, a single (Hadoop) Master and an additional Zookeeper Master. To reflect these changes also the datasets were divided into 20 blocks and distributed among workers.

Neo4j’s experimental configuration is different from the other two platforms, as Neo4j is a desktop-based platform. The only configuration parameter used by Neo4j is the heap size which we have set to 12GB.

3-1-2 Experimental Datasets

Datasets which we have used in our experiments can be found in the SNAP [29] repository and the Game Trace Archive (GTA) [30]. For our experiments we have used 18 different datasets. These datasets have various properties, ranging from small dense graphs to large sparse graphs. By choosing graphs with such distinct properties, we believe that our experiments will not be biased towards any particular platform or execution model. The full list of the datasets which we have used for our experiments is presented in Table 3-2.

From each dataset we have isolated the biggest connected component and used this component as a single graph dataset. It is important to mention that the KGS dataset in a raw “state” was a multigraph. The graph was pre-processed prior to our experiments. During the pre-processing the graph was divided into 3 distinct graphs, based on edge type, where edge type represent different game parameters.
### Table 3-2: Experimental datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WikiVote_FCF</td>
<td>1.1MB</td>
<td>SNAP</td>
<td>Wikipedia who-votes-on-whom network</td>
</tr>
<tr>
<td>EUEmail_FCF</td>
<td>6.8MB</td>
<td>SNAP</td>
<td>Email network from a EU research institution</td>
</tr>
<tr>
<td>WebGraph_FCF</td>
<td>31MB</td>
<td>SNAP</td>
<td>Web graph of Stanford.edu</td>
</tr>
<tr>
<td>Citation_FCF</td>
<td>297MB</td>
<td>SNAP</td>
<td>Citation network among US Patents</td>
</tr>
<tr>
<td>road_PA_FCF</td>
<td>28MB</td>
<td>SNAP</td>
<td>Road network of Pennsylvania</td>
</tr>
<tr>
<td>road_TX_FCF</td>
<td>36MB</td>
<td>SNAP</td>
<td>Road network of Texas</td>
</tr>
<tr>
<td>road_CA_FCF</td>
<td>54MB</td>
<td>SNAP</td>
<td>Road network of California</td>
</tr>
<tr>
<td>amazon.302_FCF</td>
<td>18MB</td>
<td>SNAP</td>
<td>Amazon co-purchased products from March 2 2003</td>
</tr>
<tr>
<td>amazon.312_FCF</td>
<td>45MB</td>
<td>SNAP</td>
<td>Amazon co-purchased products from March 12 2003</td>
</tr>
<tr>
<td>amazon.505_FCF</td>
<td>47MB</td>
<td>SNAP</td>
<td>Amazon co-purchased products from May 5 2003</td>
</tr>
<tr>
<td>amazon.601_FCF</td>
<td>47MB</td>
<td>SNAP</td>
<td>Amazon co-purchased products from June 1 2003</td>
</tr>
<tr>
<td>slashdot.0902_FCF</td>
<td>11MB</td>
<td>SNAP</td>
<td>Slashdot social network from February 16 2009</td>
</tr>
<tr>
<td>web.notredame_FCF</td>
<td>23MB</td>
<td>SNAP</td>
<td>Web graph of Notre Dame</td>
</tr>
<tr>
<td>web.google_FCF</td>
<td>76MB</td>
<td>SNAP</td>
<td>Web graph from Google</td>
</tr>
<tr>
<td>WikiTalk_FCF</td>
<td>87MB</td>
<td>SNAP</td>
<td>Wikipedia talk (communication) network</td>
</tr>
<tr>
<td>KGS_0_FCF</td>
<td>140MB</td>
<td>GTA</td>
<td>Player interaction in the GO game</td>
</tr>
<tr>
<td>KGS_1_FCF</td>
<td>210MB</td>
<td>GTA</td>
<td>Player interaction in the GO game</td>
</tr>
<tr>
<td>KGS_2_FCF</td>
<td>4.1MB</td>
<td>GTA</td>
<td>Player interaction in the GO game</td>
</tr>
</tbody>
</table>

### 3-1-3 Performance Metrics

Before we could evaluate platforms, a set of evaluation criteria metrics had to be established. From all available metrics we have decided that the metric with the single greatest impact is the execution time metric.

The execution time metric can be expressed in many ways, some of them are: throughput, turnaround time, and computation time (details can be found in [31]). We have decided to present the execution time with two metrics, these are: **turnaround time** and **computation time**. The turnaround time metric represents total time which is required for the program execution since its submission. The computation time metric represents time spend by the program exclusively on performing algorithm based computation. These metrics allow us to observe the following:

- How much time it takes for a job to finish since its submission (turnaround time)?
- How much time is spend on actual computation (computation time)?
- How much total time is spend on platform management (data distribution, reaction time etc.)
- What is the ratio of actual computation time to platform management time?

In our opinion with these two metrics we can measure the efficiency of each platform, as the time is one of the dominant factors for large scale graph processing. We also investigate other
metrics than turnaround and computational time, these metrics are: data ingestion, platform overhead, resource utilization, and platform horizontal/vertical scalability. We define data ingestion as time required by the process of obtaining, importing, and processing data for later use or storage. We define platform overhead as time spend on platform management related tasks rather than computation related tasks. We define resource utilization as the percentage of computational resources utilization during computation. We define horizontal scalability as performance increase when more computational nodes are added to the system. We define vertical scalability as performance increase when more resources are added to a single computational node in a system.

3-1-4 Experiment Execution

Every job (algorithm) was executed 10 times, to create more accurate time results. The upper bound of a job execution time was set to 60 minutes. Whenever a job exceeded this threshold, its execution was terminated. We have run each job on all datasets, however certain jobs on certain platforms have exceeded the defined execution time threshold, thus not all results are presented in the thesis.

The majority of our experimental algorithms were designed for sequential execution. To perform the experiments the algorithms had to be translated into the platform dependant programming model. In the translation process we have used many platform-dependent features, by doing so the same algorithms differ in their implementation on each platform. Furthermore we have developed a number of different solutions for each platform to solve common problems. During the translation process, we have extensively used platform dependent APIs; thus, in some cases certain data operations were added to the original algorithm. For example, Giraph supports only undirected graphs so it was necessary to create a mechanism, that supports directed graphs (such a mechanism is not present in the Hadoop and Neo4j implementation). All additional pre and post processing times have been added in every platforms, in which such a solution is a necessity. We define pre-processing as a computational process which is required to be executed before the algorithm execution, however it not a part of the algorithm execution; for example transforming edge-based dataset to vertex-based dataset. We define post-processing as a computational process required to be executed once the algorithm execution is finished, this process is also not a part of algorithm execution. Furthermore for the Neo4j platform we are presenting average of warmed-up execution times. As each of our experiments is executed 10 times, only during the first execution Neo4j reads data from the disk-based database, all consecutive algorithms execution read data from the memory. Neo4j uses a two level caching mechanism as explained in the Chapter 2. Furthermore some results are not present for certain datasets as the execution time was shorter than milliseconds, which our benchmarking tool did not record, or the execution was longer than the 1 hour job execution threshold.

3-2 Experimental Results

In this section we present the results of experiments we have performed. First we present performance evaluation of our experimental platforms, per algorithm in Sections 3-2-1 through 3-2-5. Section 3-2-6 presents data ingestion results for the evaluated platforms. Section 3-2-7
describes platform-management overhead. Section 3-2-8 introduce results for the resource utilization of the evaluated platforms, and Section 3-2-9 presents both the horizontal and vertical scalability of evaluated distributed platforms.

### 3-2-1 Graph Properties Algorithm

In our research, we have calculated graph properties for our experimental datasets with the use of the Graph Properties algorithm. We present these properties in Table 3-3.

**Table 3-3:** Graph Properties Algorithm output of experimental datasets. Missing datasets have exceeded 60 minutes execution threshold.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#vertex</th>
<th>#edge</th>
<th>LCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>WikiVote_FCF</td>
<td>7,066</td>
<td>103,663</td>
<td>0.0798</td>
</tr>
<tr>
<td>EUEmail_FCF</td>
<td>224,832</td>
<td>396,140</td>
<td>0.2470</td>
</tr>
<tr>
<td>WebGraph_FCF</td>
<td>255,265</td>
<td>2,234,572</td>
<td>0.2699</td>
</tr>
<tr>
<td>Citation_FCF</td>
<td>3,764,117</td>
<td>16,511,742</td>
<td>0.0379</td>
</tr>
<tr>
<td>road_PA_FCF</td>
<td>1,087,562</td>
<td>3,083,028</td>
<td>0.0377</td>
</tr>
<tr>
<td>road_TX_FCF</td>
<td>1,351,137</td>
<td>3,758,402</td>
<td>0.0380</td>
</tr>
<tr>
<td>road_CA_FCF</td>
<td>1,957,027</td>
<td>5,520,776</td>
<td>0.0376</td>
</tr>
<tr>
<td>web.stanford_FCF</td>
<td>255,265</td>
<td>2,234,572</td>
<td>0.2699</td>
</tr>
<tr>
<td>amazon.302_FCF</td>
<td>262,111</td>
<td>1,234,877</td>
<td>0.1506</td>
</tr>
<tr>
<td>amazon.312_FCF</td>
<td>400,727</td>
<td>3,200,440</td>
<td>0.1496</td>
</tr>
<tr>
<td>amazon.505_FCF</td>
<td>410,236</td>
<td>3,356,824</td>
<td>0.1508</td>
</tr>
<tr>
<td>amazon.601_FCF</td>
<td>403,364</td>
<td>3,387,224</td>
<td>0.1493</td>
</tr>
<tr>
<td>slashdot.0902_FCF</td>
<td>82,168</td>
<td>948,464</td>
<td>0.4213</td>
</tr>
<tr>
<td>web.notredame_FCF</td>
<td>325,729</td>
<td>1,497,134</td>
<td>0.1396</td>
</tr>
<tr>
<td>web.google_FCF</td>
<td>855,802</td>
<td>5,066,842</td>
<td>0.2673</td>
</tr>
</tbody>
</table>

We have chosen these properties as each of them can provide an insight into graph processing time. The number of vertices defines the amount of structures for which calculation has to be performed. Number of edges determines if the graph is sparse or dense, also the graph’s average degree can be easily calculated from the number of vertices and the number of edges. The Local Clustering Coefficient quantifies how close vertex neighbors are to being a clique (complete graph). In Figure 3-1 we present execution times of different platforms which we have used in our experiments.

The GP algorithm generates large amounts of intermediate data, while multicasting vertex *id* and vertex neighbours *ids* data. This helps us understand how the evaluated platform copes with the growth of the data size during runtime. For example let us assume that a vertex has 100,000 edges, of which 50% (50,000 edges) are outgoing and that each edge-destination is stored, either in memory or on disk, using 6bytes. Thus the amount of data that has to be sent to all neighbours is over 30GB; based on formula \((50,000 \times 6) \times 100,000\). In a few of our experimental datasets, such a scenario takes place. One of these datasets is the WikiTalk graph. It has a single highly connected vertex, with a 100,032 degree value. If we take under consideration that there might be over 50 high degree vertices and a total of 750,000 vertices,
the intermediate data size “explodes” in size. Thus this algorithm will test the platform for a large data size increase during runtime.

The experiment results indicate that, from the platforms we have evaluated, Giraph is the platform that copes best with large-scale datasets. Giraph being an in-memory-only platform has an obvious advantage. In contrast to Giraph, Hadoop executes jobs with costly disk I/O and long job initialization overhead. Our tests have also shown that Hadoop has a significant performance decrease when the intermediate Map step produces more data than it receives as input. This decrease comes from the Hadoop design itself. If, during the Map phase the output buffer fill threshold is reached, the buffer will be spilled to disk. Furthermore, once the data is being transmitted during the Copy phase to the Reducers, a sort-merge process is executed on the reducer nodes. The sort-merge process decreases the performance even further. However, Hadoop is not bound by the memory size of the data processing machines and is capable of processing datasets larger than the memory size (although the execution time might make it unfeasible). Giraph can not process datasets which will create intermediate results larger than the total size of a cluster memory. Neo4j is surprisingly fast. However once the dataset size grows significantly, its execution time quickly falls behind the other platforms (Citation_FCF dataset). Neo4j performs computation for each vertex sequentially while other evaluated platforms perform computation in parallel.

A more in-depth description of the issues which arise from the design of this algorithm can be found in Appendix A.

3-2-2 Breadth-first Search Algorithm

Breadth-first Search is a widely used algorithm as a basic traversal strategy for more complex algorithms. It is also being used to find shortest paths from a vertex or between pair of
vertices. In our experiments, we have randomly picked a single vertex from the dataset and started the \textit{BFS} algorithm traversal from that vertex. In Table 3-4 we present the longest path from a single source vertex, which was computed by our \textit{BFS}’s implementation.

**Table 3-4:** Longest path from a single source

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Source ID</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>WikiVote_FCF</td>
<td>8,251</td>
<td>6</td>
</tr>
<tr>
<td>EUEmail_FCF</td>
<td>74,203</td>
<td>7</td>
</tr>
<tr>
<td>WebGraph_FCF</td>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>Citation_FCF</td>
<td>4,949,326</td>
<td>9</td>
</tr>
<tr>
<td>road_PA_FCF</td>
<td>1</td>
<td>543</td>
</tr>
<tr>
<td>road_TX_FCF</td>
<td>10</td>
<td>721</td>
</tr>
<tr>
<td>road_CA_FCF</td>
<td>1</td>
<td>556</td>
</tr>
<tr>
<td>amazon.302_FCF</td>
<td>99,843</td>
<td>67</td>
</tr>
<tr>
<td>amazon.312_FCF</td>
<td>100,253</td>
<td>33</td>
</tr>
<tr>
<td>amazon.505_FCF</td>
<td>100,251</td>
<td>34</td>
</tr>
<tr>
<td>amazon.601_FCF</td>
<td>10,000</td>
<td>35</td>
</tr>
<tr>
<td>slashdot.0902_FCF</td>
<td>1,062</td>
<td>8</td>
</tr>
<tr>
<td>web.notredame_FCF</td>
<td>10,302</td>
<td>52</td>
</tr>
<tr>
<td>web.google_FCF</td>
<td>10</td>
<td>36</td>
</tr>
<tr>
<td>WikiTalk_FCF</td>
<td>2,249,533</td>
<td>3</td>
</tr>
<tr>
<td>KGS_0_FCF</td>
<td>239,044</td>
<td>10</td>
</tr>
<tr>
<td>KGS_1_FCF</td>
<td>88,814</td>
<td>8</td>
</tr>
<tr>
<td>KGS_2_FCF</td>
<td>229,713</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 3-4 present some interesting insight of our datasets. For example: the initial vertex chosen for the WikiVote dataset exhibits typical small-world phenomenon properties. Small-world phenomenon for graphs was defined by Duncan J. Watts et al. in the \textit{Collective dynamics of ‘small-world’ networks} [32] article. The phenomenon states that every vertex can be reached within 6 steps of \textit{BFS} traversal (6 hops distance). Indeed we observe this in the WikiVote_FCF results. Another interesting property can be observed in the results of roads datasets. These dataset structures tends to have a wide structure which results in long paths. Unfortunately our experimental method has a disadvantage, which can be observed in the WikiTalk_FCF dataset. Our initial vertex for the \textit{BFS} traversal was chosen randomly and as such the majority of the WikiTalk_FCF’s vertices were not reached, leading to the longest path with a distance of 3 hops.

In Figure 3-2 we present \textit{BFS} execution turnaround time of our experimental graphs.

The execution time difference between platforms is clearly visible. As mentioned earlier \textit{Giraph} executes in-memory only, while \textit{Hadoop} has a significant job initialization time and I/O overhead. The job initialization time is even more visible when combined with an iterative algorithm, such as BFS. Due to the iterative nature of BFS’s execution \textit{Hadoop} spends a long time initializing jobs at each iteration. Every iteration is equivalent to a BFS step, also each step requires submission of a new job. If we take under consideration that a single BFS execution can require over 721 steps (road_TX_FCF), \textit{Hadoop}’s job initialization overhead will become a significant part of the total execution time. This platform behaviour increases
the total execution time for every iterative algorithm.

Another overhead which occurs in Hadoop but not in Giraph is the dataset transmission overhead. While Giraph reads the dataset only once and later on it only transmits execution required messages between computational nodes, Hadoop has to read the whole dataset and pass it between Mappers and Reducers in each step of the algorithm execution. This overhead has an obvious impact on the overall performance as well. There are design patterns which can decrease the data transmission overhead in the Hadoop platform such as Schimmy [33], however their usefulness and impact on the execution time is out of the scope of this thesis.

As the BFS algorithm is not computational intensive and not all vertices participate in the computation at each execution step, Neo4j achieves excellent execution time results. It outperforms all other platforms in each experiment. However in our opinion once the dataset size will increase, Neo4j’s execution time should fall behind other platforms, as it is limited by sequential computation.

### 3-2-3 Connected Components Algorithm

The Connected Component algorithm solves a common problem of many graph datasets. The algorithm iteratively finds all disjoint graphs (connected components) within a dataset. This algorithm was used to obtain our experimental datasets. In Figure 3-3 we present our results regarding Connected Components algorithm execution time on our experimental graphs.

Since the Connected Components algorithm represents an iterative class of algorithms, Hadoop overhead explained in Section 3-2-2 also applies to the execution of this algorithm. The Connected Components algorithm executes longer on the Hadoop platform than on the Giraph platform. This behaviour was expected by us, as the algorithm’s execution is comparable to

![Figure 3-2: Breadth-first search performance.](image-url)
3-2 Experimental Results

Figure 3-3: Connected Components performance.

the BFS’s execution. The difference is in the number of iterations. Label propagation executes similarly to the BFS graph traversal, as each label is propagated in a series of steps and labels follow BFS traversal scheme. The algorithm execution times only confirms all our findings regarding iterative algorithms execution on the evaluated platforms.

Also for this algorithm Neo4j achieves fast execution in comparison to other platforms. However even though the Connected Components algorithm is not highly computational intensive, the sheer amount of elements to iterate over during its execution has a profound impact on Neo4j’s overall execution time. This trend can be seen in the Citation_FCF dataset. This partially confirms our finding regarding Neo4j’s BFS execution.

3-2-4 Community Detection Algorithm

Our implementation of the LPA algorithm follows the authors design, except for the Giraph’s implementation. The algorithm requires a global view of stable labels, as when all labels are stable the algorithm finishes its execution. Unfortunately Giraph does not offer any form of global view, programmatically creating one would require additional computational steps; aggregators are global only per worker. Such an implementation will create messaging overhead and the program will halt with 2 steps delay, after reaching equilibrium. The first overhead step is the step in which algorithm has reached equilibrium, yet the “global view mechanism” has to receive this information. Second overhead step is when the “global view mechanism” notifies all vertices that the equilibrium was reached and the algorithm execution should be terminated. Thus in our Giraph implementation we have used a program argument which specifies the number of steps to execute. After obtaining the number of steps required for each dataset from other platforms we pass this knowledge to our Giraph implementation. Furthermore we have added an additional constraint of executing at most
20 algorithm iterations, as authors state that after 5 iterations 95% of labels are stable. In Table 3-5 we present results we have obtained from our experiments.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>nr. of communities</th>
<th>biggest community size</th>
<th>total nr. of vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>WikiVote_FCF</td>
<td>1</td>
<td>7,066</td>
<td>7,066</td>
</tr>
<tr>
<td>EUEmail_FCF</td>
<td>117</td>
<td>93,028</td>
<td>224,832</td>
</tr>
<tr>
<td>WebGraph_FCF</td>
<td>2,613</td>
<td>58,946</td>
<td>255,265</td>
</tr>
<tr>
<td>Citation_FCF</td>
<td>24,302</td>
<td>38,093</td>
<td>3,764,117</td>
</tr>
<tr>
<td>road_PA_FCF</td>
<td>22,066</td>
<td>481</td>
<td>1,087,562</td>
</tr>
<tr>
<td>road_TX_FCF</td>
<td>29,295</td>
<td>450</td>
<td>1,351,137</td>
</tr>
<tr>
<td>road_CA_FCF</td>
<td>42,007</td>
<td>527</td>
<td>1,957,027</td>
</tr>
<tr>
<td>amazon.302_FCF</td>
<td>16,706</td>
<td>2,195</td>
<td>262,111</td>
</tr>
<tr>
<td>amazon.312_FCF</td>
<td>18,369</td>
<td>3,471</td>
<td>400,727</td>
</tr>
<tr>
<td>amazon.505_FCF</td>
<td>18,527</td>
<td>3,155</td>
<td>410,236</td>
</tr>
<tr>
<td>amazon.601_FCF</td>
<td>18,640</td>
<td>2,917</td>
<td>403,364</td>
</tr>
<tr>
<td>slashdot.0902_FCF</td>
<td>395</td>
<td>80,093</td>
<td>82,168</td>
</tr>
<tr>
<td>web.notredame_FCF</td>
<td>1,420</td>
<td>133,218</td>
<td>325,729</td>
</tr>
<tr>
<td>web.google_FCF</td>
<td>9,575</td>
<td>8,483</td>
<td>855,802</td>
</tr>
<tr>
<td>WikiTalk_FCF</td>
<td>3</td>
<td>2,388,924</td>
<td>2,388,953</td>
</tr>
<tr>
<td>KGS_0_FCF</td>
<td>13</td>
<td>761,360</td>
<td>761,855</td>
</tr>
<tr>
<td>KGS_1_FCF</td>
<td>8</td>
<td>293,198</td>
<td>293,290</td>
</tr>
<tr>
<td>KGS_2_FCF</td>
<td>28</td>
<td>74,274</td>
<td>100,056</td>
</tr>
</tbody>
</table>

Our experimental results have provided us with an insight view of the communities existing within our datasets. For example WikiVote_FCF is an interesting dataset example, which consists of a single community. EU/E-mail_FCF dataset on the other hand has many communities, but there is one dominating community which consists of over 41% of all vertices in the dataset. The Road datasets consists of a large number of small communities, this occurs due to the structure of the datasets which was described in the BFS algorithm results (Section 3-2-2). In Figure 3-4 we present our results regarding Community Detection algorithm execution time on our experimental graphs.

Also this experiment confirms that Hadoop is not the most suitable platform for iterative algorithms. The execution time difference between Giraph and Hadoop is clearly visible. Once more Hadoop is paying the job initialization overhead price and I/O overhead price during each iteration. Neo4j achieves good performance time for datasets with a small amount of vertices, however once the number of vertices grows its execution time increases as well. Because the LPA algorithm is more computational intensive than BFS or Connected Components the decrease trend observable in Neo4j execution occurs much faster than in other algorithms.

3-2-5 Graph Evolution Algorithm

Forest Fire Model (FFM) is an algorithm which predicts how new vertices will join existing graphs. In our implementations of the FFM algorithm we have used the number of hops traversed as a stopping condition, also in each iteration we have used the same probability
of creating new edges as in the previous time step. We have made these changes as there are currently three versions of the FFM algorithm, these versions are present in the 2005 publication [27], 2007 publication [34] and the SNAP framework [29]. In our experiments FFM creates a number of new vertices close to 1% of existing vertices in the original dataset. Our implementation of the FFM has computed results presented in Table 3-6.

The execution of the FFM is random in its nature as the number of outgoing edges created by a new vertex is chosen randomly at each step. This is reflected in the results as there can not be found any relation between the number of newly created edges and any of the graph properties. For example a relatively small number of new vertices created for the WikiVote dataset has a significantly larger proportion between new vertices and new edges than the EUEmail dataset. In Figure 3-5 we present the results of our experiments regarding FFM execution time.

As in every other experiment, also in this one Hadoop’s execution time is highly dependent on the job initialization overhead and I/O overhead of each iteration. However for the first time Giraph has also exceeded our job time threshold with some datasets, these are: WebGraph_FCF, WikiTalk_FCF, KGS_0_FCF. In our opinion this longer than average execution time of Giraph’s job is caused by creating new structures within the dataset. At each step of the algorithm execution, new edges are created (new vertices are created only during the first step). Hadoop creates these new edges by simply attaching new strings to the datasets, before they are passed to the next iteration. Giraph creates new edges by modifying existing data structures, which are distributed across all workers. Furthermore these changes have to be propagated across all workers to maintain global consistency. In our opinion the process of modifying and synchronizing all graph related data structures, takes a very long time in Giraph, which is reflected in the execution time. A similar situation occurs with the Neo4j execution. Whenever a vertex is modified the change is immediately propagated to

![Figure 3-4: Community Detection performance.](image-url)
Table 3-6: Graph Evolution results, based on Giraph results.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>new vertices</th>
<th>new edges</th>
<th>avg. edges per vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>WikiVote_FCF</td>
<td>60</td>
<td>27,653</td>
<td>460</td>
</tr>
<tr>
<td>EUEmail_FCF</td>
<td>2,200</td>
<td>126,686</td>
<td>57</td>
</tr>
<tr>
<td>Citation_FCF</td>
<td>37,000</td>
<td>13,163,198</td>
<td>355</td>
</tr>
<tr>
<td>road_PA_FCF</td>
<td>10,000</td>
<td>98,722</td>
<td>9</td>
</tr>
<tr>
<td>road_TX_FCF</td>
<td>13,000</td>
<td>124,104</td>
<td>9</td>
</tr>
<tr>
<td>road_CA_FCF</td>
<td>19,000</td>
<td>185,725</td>
<td>9</td>
</tr>
<tr>
<td>amazon.302_FCF</td>
<td>2,600</td>
<td>490,182</td>
<td>188</td>
</tr>
<tr>
<td>amazon.312_FCF</td>
<td>4,000</td>
<td>1,346,450</td>
<td>336</td>
</tr>
<tr>
<td>amazon.505_FCF</td>
<td>4,100</td>
<td>1,392,946</td>
<td>339</td>
</tr>
<tr>
<td>amazon.601_FCF</td>
<td>4,000</td>
<td>1,375,962</td>
<td>343</td>
</tr>
<tr>
<td>slashdot.0902_FCF</td>
<td>800</td>
<td>291,220</td>
<td>364</td>
</tr>
<tr>
<td>web.notredame_FCF</td>
<td>3,200</td>
<td>285,698</td>
<td>89</td>
</tr>
<tr>
<td>web.google_FCF</td>
<td>8,500</td>
<td>1,366,714</td>
<td>160</td>
</tr>
<tr>
<td>KGS_1_FCF</td>
<td>2,900</td>
<td>36,548</td>
<td>12</td>
</tr>
<tr>
<td>KGS_2_FCF</td>
<td>1,000</td>
<td>9,304</td>
<td>9</td>
</tr>
</tbody>
</table>

the underlying database, which in return significantly increases the execution time, as the database consistency has to be kept.

Another interesting observation we have made during these experiments is the amount of logs Giraph produces while modifying the graph structure. Each modification is reflected in the logging system by default. This can lead to a disk space problems, as for example a single FFM job executed on the Citation_FCF dataset has generated 24GB of logs.

3-2-6 Data Ingestion

To be able to process data, data first has to be transferred to the storage system. This process is called Data Ingestion. We define data ingestion as the process of obtaining, importing, and processing data for later use or storage. We have performed a series of experiments of two underlying storage systems of our evaluated platforms. In these experiments we have tested the HDFS which is a distributed file system used by Hadoop and Giraph. Also we have performed the same experiments for the Neo4j database. As our experimental dataset we have used the Citation_FCF dataset, as it has the largest physical size. In Figure 3-6 we present our results.

From the Figure 3-6 it is clear that the data ingestion process of HDFS is a lot faster than the one for the Neo4j database. In our experiment on the HDFS system we have used replication’s default value (dataset blocks are stored in 3 copies on different datanodes). Neo4j’s data ingestion process takes a very long time when compared with HDFS, this can be caused by sub-optimal insertion process which we use in Neo4j. We perform data ingestion into the Neo4j database with the use of multiple batch transactions. We have done so to avoid the Garbage Collection Overhead error which occurred when we tried to insert the datasets in a single transaction. Our transaction threshold is set to 10,000 elements, meaning that every 10,000 vertices or edges we commit the transaction (Neo4j requires vertices to be
inserted prior to edges). In our opinion this process can be improved by increasing the batch transaction threshold, which will result in a shorter insertion time. However there is a risk that once the threshold is set too high the Overhead error will occur, breaking the database consistency. An in-depth analysis of the data ingestion process is out of the scope of this thesis.

Figure 3-5: Forest Fire Model performance.

Figure 3-6: Data Ingestion.
3-2-7 Platform Overhead

We have also performed more detailed analysis of a job execution of the platforms which we evaluate. Unfortunately we could only analyze the Hadoop and Giraph platforms, as for Neo4j this would require source code modifications, to acquire accurate results. To measure Hadoop’s platform overhead we have used the Starfish tool created by Herodotou et al.\[^{35}\]. To measure Giraph’s overhead we have modified the program to record computation time. We have divided the total job execution time into the computation time and the overhead time. We define the computation time as the time spent exclusively on performing computation related to the algorithm execution. The overhead time is defined as the time spend on performing platform related computation (including communication). In our experiments we have used the execution of the BFS algorithm with a Citation_FCF dataset as an input. We have chosen the BFS algorithm due to its lightweight computation execution, which will have little impact on the platform execution. The Citation_FCF dataset was chosen as it has the largest physical size and will allow us to measure data transmission overhead more accurately. We present our results in Figure 3-7.

![Figure 3-7: Platform execution analysis.](image)

In the Hadoop platform the overhead takes nearly 36% of the total execution time, while the algorithm related computation takes on average 64%. This means that over one-third of the execution time of any job is used for platform management, acquiring resources etc. However in these 36% of the overhead time we have included the essential tasks of the MapReduce programming model such as the shuffle process, reducer side sort-merge etc.

For the Giraph platform the overhead takes on average 97.3% of the total execution time. The algorithm related computation utilizes 0.27% of the total execution time, the remaining time is the Hadoop’s overhead. Again as the overhead we have included platform management, programming model related computation etc. Since Giraph is built on top of the Hadoop
platform we have created simple software to retrieve the detailed execution time of a Giraph job. The Giraph job was divided as following:

- job init - 2.022 seconds
- vertex input superstep - 24.378 seconds
- step related computation including communication - 15.749 seconds
- shutdown - 1.884 seconds

On average our experimental job took 65 seconds, from this time the Giraph platform related computation (including algorithm computation) took 44 seconds, the remaining time is the Hadoop platform overhead. The reason for such a short computation time of our experimental job can be found in the Pregel programming model itself. Pregel does not process all vertices during each step, it processes only vertices which actually participate in the computation of step \( i \). Thus Giraph achieves much shorter computation steps than Hadoop, in which every vertex participates in each step. Furthermore since BFS algorithm is data intensive not computation intensive, the bulk of the computation is related to communication, which we label as overhead.

### 3-2-8 Resource Utilization

Up till now we have performed performance evaluation of our experimental platforms, data ingestion of two underlying storage systems and compared time spend on computation against time spend on platform management. It is only logical to perform resource utilization experiments for our evaluated platforms. For the purpose of recording resource utilization we have used the Ganglia [36] tool, which is a scalable distributed monitoring system for high-performance computing systems such as clusters and grids. Unfortunately we could not obtain resource utilization metrics for the Neo4j platform as the execution time of each experimental algorithm is measured in milliseconds. Thus recording Neo4j’s utilization metrics would require recording metrics at least every \( N \) microseconds. Standard bash commands do not ensure such a level of time granularity. Thus we present our findings only for the Giraph and Hadoop platforms. We have performed resource utilization measurements both for the Master computation node and all the Workers computational nodes. We present normalized results of the actual program execution, which values are closest to the expectation (average from all experimental executions). The CPU utilization is a percentage of utilization of all 8 cores present in the CPU, even if not all of them are used. Also the memory utilization and network utilization is recorded for a whole computational node, whenever these resources are used or not. We have performed these experiments by executing the BFS algorithm with a Citation_FCF dataset as input. We have done so for the same reasons as in the platform overhead related experiments.

#### Master Resource Utilization

In each of our experiments the Master computational node is responsible for scheduling job processes to appropriate execution nodes, fault tolerance of the job execution and recording
platform dependant execution metadata. Although in all of our experiments we submit a single job to execute, the Master still has to manage the platform. Below we present Master resource utilization for our experimental platforms.

From the Giraph’s CPU utilization we can see that for the most time the Master is rather idle. Then by the end of the job its CPU utilization reaches 0.1%. In our opinion this is due to Giraph’s platform design. After the initial job submission the Master does not perform a lot of work, during this time Workers setup their local vertices data. After that phase is finished the computation begins, after which a global cleanup and resource deallocation is performed, which requires coordination with the Master node.

Hadoop’s Master behaves in a different way, which is dictated by the algorithm execution. Since the Hadoop Master coordinates each phase of the job execution (the algorithm execution consists of a number of these jobs), this is reflected in the Master’s CPU utilization. The algorithm execution consists of 10 iterations execution. After the job initialization, first few jobs are relatively short, thus the Master coordination is visible. However once a longer job is being executed Master’s CPU utilization drops significantly. The results of the experiments are presented in Figure 3-8.

![Figure 3-8: CPU utilization by platform’s Master.](image)

The memory usage is constant for both platforms. As each job is submitted only once and there are no other jobs running, thus the memory has to hold data related to only this particular job and platform management related data. The results of the memory utilization experiments are presented in Figure 3-9.

Network utilization for the Giraph platform’s Master confirms our CPU utilization findings. Initially the number of bytes received is relatively small, only to significantly grow by the end of the job. The number of bytes send by the Master is rather constant, this reflects the management role of the Master, which does not participate in the vertex setup phase.
The Hadoop platform behaves in very similar way as the Giraph, although its Master has participated in all phases of each job being executed. Thus the overall network utilization is higher. Network utilization experiments are presented in Figure 3-10.
Worker Resource Utilization

Worker computational nodes perform all algorithm related computation, they are also exchanging messages between each other and are responsible for sending heartbeats to the Master.

Giraph’s CPU utilization reflects the job execution phases. At the beginning of a job simple management operations are performed (including job scheduling overhead). Once the job is properly initialized, each Worker performs job related computation. It starts by initializing vertex input data and continues with job execution, this is reflected in the Giraph’s CPU utilization plot.

Hadoop’s CPU utilization is less constant than Giraph’s. The data shown in the plot is an actual execution of one of the workers. We can see that during different stages of the algorithm execution, CPU load was different, reaching its peak in the middle of the execution. The results of the experiments are presented in Figure 3-11.

![Figure 3-11: CPU utilization by platform’s Worker.](image)

The memory usage is rather constant, because at each Worker a single job is being executed, thus data required for the job related computation is constant. The small deviations can be explained by additional data required by the algorithm. This additional data is just a small fraction of the overall memory utilization. The results of the experiments are presented in Figure 3-12.

Giraph utilizes the network very similarly to the CPU. During the initial phases there is very little communication between Workers. However once the job data has been initialized and Workers perform algorithm related operations, the number of sent and received bytes grows.

The Hadoop uses the network in a number of continuous “bursts” between Map and Reduce phase. Because both Map and Reduce phases are short in our experiments, these bursts seem
to look like a continuous usage. However we can clearly distinguish the beginning and the end of each map-reduce cycle. In the middle of each cycle a network utilization “burst” occurs. The results of the experiments are presented in Figure 3-13.
3-2-9  Scalability

One of the key factors in performance evaluation of the distributed graph processing platforms is their capability to scale. Thus in our research we have performed scalability experiments. In our experiments we have used the largest size dataset available from our previous experiments, the Citation_FCF graph and the BFS algorithm due its computation lightweight execution. We have performed both horizontal scalability experiments and vertical scalability experiments. Horizontal scalability is achieved by adding more computational nodes to the system. Vertical scalability is achieved by adding more resources to a single computational node in a system, in our experiments we were increasing CPU cores usage. In this section we will present our findings regarding Hadoop and Giraph scaling capabilities.

Horizontal Scalability

In our horizontal scalability experiments we have created custom size clusters for each of the experiments. Our experiments were run using 20 nodes with 1 CPU utilization. For the horizontal experiment we have decreased the size of the cluster, thus in our experiments we have used cluster configurations presented in Table 3-7.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Giraph</th>
<th>Hadoop</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 nodes</td>
<td>6 workers</td>
<td>3 mappers, 3 reducers</td>
</tr>
<tr>
<td>10 nodes</td>
<td>10 workers</td>
<td>5 mappers, 5 reducers</td>
</tr>
<tr>
<td>16 nodes</td>
<td>16 workers</td>
<td>8 mappers, 8 reducers</td>
</tr>
<tr>
<td>20 nodes</td>
<td>20 workers</td>
<td>10 mappers, 10 reducers</td>
</tr>
</tbody>
</table>

As in every experiment also this time we have performed 10 runs for each experiment and calculated the average execution time. Due to the large difference in the execution time between platforms, the Giraph’s error bars are scarcely presented. The turnaround execution times of the experiments are presented in Figure 3-14.

Though from the plot it seems that Hadoop has a better horizontal scalability, that is not the case in our opinion. To understand the reason for better performance increase in the Hadoop’s scalability we have to mention one more time the I/O overhead for iterative algorithms which Hadoop suffers from. Each iteration of the algorithm requires Hadoop to perform HDFS read and write. From our previous experiments we know that this overhead is a crucial factor in the execution time of Hadoop’s iterative jobs. When the number of workers increases, the amount of data that each worker processes decreases, thus the overall time of the job execution decreases as well. If we compare the input size for a single task between 6 worker and 20 workers configuration (around 110MB and around 32MB respectively), it is easily noticeable that the amount of data which participates in the I/O overhead for a single task decreased significantly, which in return had decreased the execution time.

Giraph performs the HDFS read and write only once (during initialization and committing results), thus the input size does not play a key factor in the execution performance, as in Hadoop. Furthermore Giraph’s submits a single job, thus the job initialization overhead
occurs only once. This results in a less visible horizontal scalability for our experimental dataset.

**Vertical Scalability**

In our vertical scalability experiments we have again created custom size clusters for each of the experiments. Our vertical scalability experiments were executed with the use of 20 nodes with 1-7 CPU’s core utilization. Although DAS4 offers 8 cores at each computational node, it is advised to leave a single core for *Hadoop’s daemons* and the operating system. In each vertical scalability experiment we have increase the number of cores utilized. As usual for each configuration we performed 10 runs.

Unfortunately during our experiments we have encountered problems with inconsistent results for *Hadoop*. After investigating the issues we have reached the conclusion that the dataset size is too small to perform vertical scalability experiments (only 320 MB). In our opinion vertical scalability experiments with the *Citation_FCF* dataset as input will not record the speedup while vertically scaling resources, because for such small dataset the execution speedup is just a small fraction of the total execution time (which largely consists of a constant platform overhead, for details see Section 3-2-7). As from the results we can see that the performance increased initially only to decrease for configurations exceeding 4 CPU cores. Similar problem occurs also for the *Giraph* platform, as *Master* has to coordinate increasing number of resources, while the execution speedup is just a small fraction of total execution time. Our solution for this problem was to increase the experimental dataset size. Vertical scalability experiments results with the use of *Citation_FCF* dataset are presented in Figure 3-15.

To overcome the problem of insufficient dataset size we have performed another round of vertical scalability experiments with the use of the *Friendster* dataset from the SNAP repos-
The dataset consists of more than 65,000,000 vertices, 1,806,067,135 edges and its size is 32GB. Unfortunately due to time constraints we have performed only 5 BFS algorithm executions for each configuration. Results are presented in Figure 3-16.

Increasing the dataset size has increased the amount of data which has to be processed. Thus the platform overhead is less visible as platform has a significantly larger amount of computation to perform. The platform overhead is still present, yet it represents only a small fraction of the total execution time. The recorded results show that Hadoop scales vertically as expected. The performance gain between 2 CPUs per worker and 7 CPUs per work is roughly equal to 70 minutes.

We have also performed Giraph’s experiments for the new SNAP dataset, unfortunately all of our test cases have failed during execution due to platform internal errors. In our opinion Giraph’s current development version can not cope with such large datasets. We have tried to update our Giraph instance to match the “trunk” version. This change required us to rewrite the source code of the BFS job. Unfortunately the trunk version was not stable, as every job failed during the initialization process.

However during the vertical scalability experiments which used the large dataset we have discovered that Giraph requires a significant amount of memory to initialize the internal platform graph structure. For example in the single CPU’s core configuration each worker receives input of 1.6 GB. Giraph workers require in some cases nearly 10 GB to fully initialize the structure (before algorithm execution), which is nearly 1/3 of the total input size.
3-3 Summary

Answering the main research question 1 (see Section 1-2), in this chapter we have presented an extensive performance evaluation of three large-scale graph processing platforms. We have shown not only their performance but also their scalability, data ingestion time, and also performance break-down and resource utilization results. We will now perform a final comparison regarding each of the evaluated platforms.

The goal of our performance evaluation is to give a first answer to the following questions:

- Are modern desktop computers good for large-scale graph processing?
- Is a specialized large-scale graph processing platform more useful than non-specialized platforms?
- What determines a good graph processing platform?

Based on the evidence shown by our experiments, we have reached the following conclusion. Modern desktop computers can handle most of the smaller or sparser graph datasets. However, once the dataset size significantly increases or if the graph is dense, the execution time increases significantly. For this reason, desktop-based graph-processing platforms cannot compete with distributed platforms.

In our experiments, we have used two graph-processing platforms (Giraph, Neo4j) and a generic data-processing platform (Hadoop). The platforms which focus on processing graph datasets achieve significant performance advantages over the generic platform. All algorithms implemented for the purpose of this evaluation were easier to express in the graph specialized

Figure 3-16: Vertical scalability of evaluated platforms with the use of Friendster dataset. (Vertical axis does not start at 0)
environments. However, for the Forest Fire Model graph evolution algorithm, the Hadoop platform turned out to be a better choice, than the specialized platform for certain datasets (see Section 3.2.5). As Hadoop does not maintain the relations between data and treats every vertex as a disjoint piece of information, it does not have to maintain global consistency of the graph dataset, which other platforms have to and pay a performance penalty for. Thus, for certain datasets Hadoop has achieved better performance than the graph-processing platforms.

Our evaluation indicates two significant factors for the large-scale graph-processing platforms: the programming model and the platform design. We have shown comprehensive empirical evidence that the Pregel programming model is more suitable for implementing graph processing algorithms than the MapReduce programming model. For the platform’s design/implementation, although it is possible to implement graph processing algorithms in the MapReduce programming model, the Hadoop’s design is not well suited for this task. Its design requires to perform every step of the algorithm in a separate job. This results in a costly job initialization overhead and I/O overhead during each algorithm’s step. The Giraph platform is more suitable for iterative processing as it reads input data only once and performs the algorithm computation in-memory, which results in much better performance. However, Hadoop is not limited by the total cluster memory, as it can spill its data to local disks during the computation, which is a significant advantage specially for smaller clusters. Thus even though the Giraph platform implements a suitable programming model and has more suitable design, as it has reliability issues that limit its usability.

The Neo4j platform which, represents the desktop, NoSQL type of processing platforms, has achieved very good performance for the smaller or sparser datasets. Graph processing algorithms were easy to express in the Neo4j’s graph traversal programming model. The API offered by the platform was solid and the documentation provided by the authors allows to quickly learn all features of the platform.

The Hadoop platform, which represents the large-scale data processing platforms, was the slowest platform of all evaluated platforms, although Neo4j’s performance for the larger or denser datasets is lower than the Hadoop’s. Only for graph evolution and a few datasets Hadoop had better performance than other platforms. Graph processing algorithms are not easily translated to the MapReduce programming model. However a pattern can be created which will be used by the majority of graph processing algorithm implementations. In this pattern the Map phase is treated as a data distribution phase, which is responsible for transmitting data to the proper vertices. The Reduce phase acts mainly as a main computation phase, in which the bulk of the algorithm is implemented. For optimization purposes some deviations from this pattern are possible. The platform is very well documented and has strong community support.

The Giraph platform, which represents the distributed large-scale graph processing platforms, was the fastest platform in all of our experiments. It has achieved high performance due to only in-memory processing. The Pregel programming model was proven to be very flexible and graph processing algorithms are easy to express in the programming model. For our evaluation, we have used a development version of the platform (0.2), which unfortunately was poorly documented. Thus, we conclude that Giraph can be difficult to learn for newcomers. Giraph’s current most significant limitation is the only in-memory execution model: because all the computation is performed in-memory, dataset size can not exceed the total memory size of the available cluster.

Marcin Biczak  Master of Science Thesis
Further experiments were conducted within the PDS group using the experimental approach described in this thesis [14]. These experiments which used more datasets and platforms (notably YARN [21] and Stratosphere [37]), confirm our findings.
Chapter 4

LudoGraph Design & Validation

To answer the second man research question (see Section 1-2), in this chapter we design LudoGraph, a Pregel-based graph-processing platform with cloud-capabilities. We also describe the goals of our designed graph processing platform in Section 4-1. We present the novelty of the platform and describe the entire platform design in Sections 4-2 and 4-3, respectively. We introduce our prototype validation in Section 4-4. In Section 4-5 we summarize this chapter. Some of this work to be extended as open-source and as ICDE’14 article.

LudoGraph is a large-scale graph processing platform. It was originally designed to process game graphs, which explains the 'Ludo' part of the name (from 'Ludic' game-like playful). LudoGraph is a component of CAMEO [38] gaming analytics element of our platform for massively social gaming, @large [39].

4-1 Goals of LudoGraph

LudoGraph was designed with the goal of bringing cloud capabilities to the graph-processing. To achieve this without the need of a complex configuration procedure, the provisioning mechanism should be self-contained and dependent on the specifics of each job, rather than on a global configuration. Small and medium enterprises who are willing to trade-off execution time for cost of execution should benefit from this functionality. Another goal of the LudoGraph platform is to support the Pregel programming model. However, during our work on the performance evaluation of graph processing platforms (Chapter 3), we have noticed that new functionalities can be added to the existing Pregel programming model to make it easier to use and more flexible (see Section 4-3-2). LudoGraph also aims to support users who are willing to trade-off results accuracy for runtime, via sampling of the input graph. Furthermore, while performing experiments with the sampling algorithms (Chapter 5), we have realised that sampling can be performed in parallel with the graph properties computation. LudoGraph will be executed under the YARN’s platform management.
4-2 Overview of novelties in LudoGraph

In this section we explain each of the novelties of the LudoGraph platform. We concentrate on the three main contributions, these are:

1. Cloud capabilities automated by the platform (Section 4-2-1)
2. Extended Pregel programming model (Section 4-2-2)
3. Graph sampling and custom graph traversal as core functionality automated by the platform (Section 4-2-3)

4-2-1 Cloud Capabilities

LudoGraph is by design a cloud platform that can also use standard cluster resources. Our goal for this element is to make this process as simple as possible for the user, by automating all resource-related processes (that is, leasing, using, cleaning up, and releasing). The user only has to specify which provisioning policy to use and the threshold which will trigger the provisioning mechanism on a job level.

4-2-2 Implementation of Pregel in LudoGraph

The main component responsible for executing jobs is the PregelEngine (PE), our implementation of the Pregel programming model (details in Section 4-3-6). However, LudoGraph adds several additional features to the original Pregel design:

- Support for incoming edges - Pregel supports only outgoing edges for vertices, which might lead to additional code for directed datasets and can reduce performance. LudoGraph supports by design both incoming and outgoing edges for vertices.

- PUSH and PULL mechanisms for message exchange - Pregel supports only PUSH mechanism for message based communication. However, certain algorithms are easier to implement with the use of a PULL mechanism; for example, in the Forest Fire Model (see Section 2-5-5) the ambassador election is easier to express with the use of a PULL mechanism.

- Synchronized global variables - many graph processing algorithms were designed for sequential execution, in which a global view of the data is always available (through the use of global variables). By providing synchronized global variables, these algorithms are easier to implement in the LudoGraph extended-Pregel than in the classic Pregel model.

- Fetch messages - are a special type of messages whose purpose is to request certain vertex data. These messages, when sent at step $T$ of the computation will return the requested data at step $T+1$. Similar functionality can be implemented within the existing Pregel programming model, but it requires custom code and the requested data would only become available at step $T+2$; the latter further complicates the source code.
4-2-3  Runtime Sampling and Custom Traversal

Our last novelty is the “at-runtime” sampling functionality of the LudoGraph platform. Sampling is an important part of the graph processing field. Most sampling algorithms return a sample which is later processed by the user in a separate program. This process introduces a lot of overhead, as the same vertices are traversed twice (during sampling and during sample processing) and the output of sampling may have to be spilled, and read from disk (a slow operation). Our idea is to perform the sampling related computation during the actual sampling process, this eliminates the additional overhead.

From the perspective of the platform, sampling is just a specific type of graph-traversal. We have made a decision to extend LudoGraph with a generic process for custom graph traversal. Making out platform easy to use, we give user the ability to wrap computation code in a custom graph traversal algorithm. For example, user may want to use the BFS graph traversal algorithm, but perform computation only for vertices with a degree higher than a certain threshold (the computational threshold). LudoGraph automates this process, and makes this process intuitive and easy to implement.

4-3  LudoGraph Design

In this section we detail the design of LudoGraph. We explain how we plan to achieve all of the novelties described in Section 4-2. Furthermore, we describe the basic operation of the platform: platform bootstrapping, job execution, etc. In Section 4-3-1, we explain why we have chosen to support the YARN platform as a basic resource management block. In Section 4-3-2, we present LudoGraph’s components. In Section 4-3-3, we describe the platform bootstrapping process. In Section 4-3-4, we explain LudoGraph’s job execution. In Section 4-3-5, we introduce the cloud capabilities of LudoGraph. In Section 4-3-6, we explain our implementation of the extended Pregel programming model. In Section 4-3-7, we explain how we achieve runtime sampling and custom graph traversal.

4-3-1  LudoGraph and YARN

LudoGraph uses YARN as a basic resource management block. The main reason for choosing YARN is its design, which offers much freedom for creating distributed applications, while providing basic cluster-level resource management and fault-tolerance.

Once YARN’s Application Master (AM) is started by the Resource Manager (RM) and granted all resources required for the job execution, the AM becomes responsible for job management, programming models, fault-tolerance policies, etc. Our idea of implementing new LudoGraph’s AM is to create an independent cloud-based platform that extends the cluster managed by YARN. Furthermore, we introduce the Pregel [7] programming model into the YARN ecosystem. From a user perspective, LudoGraph acts as an independent cloud platform embedded in the YARN cluster. From the perspective of the YARN platform, LudoGraph is just a long-running job.
4-3-2 LudoGraph Components

The components of the LudoGraph platform are divided among two separate node types (see Figure 4-1): the master node (AM) and worker node (Worker). The Application Master, which is responsible for managing the LudoGraph platform, is initialized by the Cluster Client during the platform bootstrapping process (details are presented in Section 4-3-3). The worker is responsible for performing the computation.

Figure 4-1: LudoGraph Components. YARN is the basic resource management block.

We now describe each component, grouped by their residence on specific resource types (for YARN’s architecture terminology see Chapter 2 Section 2-4).

Client Components

The Cluster Client is a program which is responsible for LudoGraph’s bootstrapping process. Before the platform can be used, the Cluster Client has to contact the Resource Manager to initialize the Application Master and pass the platform configuration as well as allocate the initial amount of computational containers.

The Job Client is a program which is responsible for initializing job execution. Once the LudoGraph platform is initialized the Job Client will contact the Resource Manager to be redirected to the Application Master to schedule the job execution.

Master Components

The Platform Status Observer (PSO) manages the health status of each container in the system. PSO is based on the Publish-Subscribe design pattern, where each Container is
a publisher and the AM is the subscriber. PSO module receives data propagated by the Heartbeat module from each container. Its purpose is to maintain “health” status of platform’s computational containers. Whenever a computational container stops working properly, its Heartbeat module will stop sending status messages. The Platform Status Observer will detect this and mark the computational container as “dead”, no job will be scheduled to this container until the problem is fixed. The PSO module can also be coupled with the provisioning mechanism: in case of a container critical failure, the Provisioning Mechanism can be contacted to request an additional container to maintain the minimal amount of “healthy” containers required by the platform.

The Fault Tolerance Mechanism (FTM) works in cooperation with the Platform Status Observer. The PSO module maintains the list of “healthy” and “dead” containers and the FTM module is responsible for taking appropriate actions based on the amount of “healthy” containers in the system. It can either reschedule a task assigned originally to a “dead” container to a “healthy” one, or it can contact the PM module to increase the amount of “healthy” containers to maintain the required minimal amount.

The Resource Utilization Observer (RUO) is another module based on the Publish-Subscribe design pattern. This module receives data from the Resource Utilization Recorder module residing on each Container, which provides resource utilization metrics of a computational container. Each Resource Utilization Recorder is a publisher and the Resource Utilization Observer is the subscriber. The main purpose of this module is to maintain resource utilization metrics for each container in the system. If any job exceeds the lower-bound or the upper-bound of the resources utilization, the Provisioning Mechanism will be contacted to take proper action.

The Provisioning Mechanism (PM) works in cooperation with the Resource Utilization Observer. Whenever a job resource utilization exceeds the lower-bound or the upper-bound the PM module will free the under provisioned resource making them available for other pending jobs (or will deallocate them) or will request additional computational resources to speedup a high utilization job.

The Job Scheduler (JS) module maintains a queue of pending jobs and cooperates with the Job Dispatcher module. The JS module is responsible for contacting the Job Dispatcher module to initialize the job, maintaining a job priority queue, and scheduling the job with the highest priority as the first one to be executed. Furthermore the JS module will try to execute jobs on computational containers with the best data locality possible.

The Job Dispatcher module is responsible for initializing job execution. During the job initialization this module will pass all parameters required by the job, all dependency libraries and will start job tasks on each Container.

The Web UI module is a simple lightweight web-application, which is responsible for presenting the platform and job status to the user in a web browser. It will collect data about resource utilization from the Resource Utilization Observer, will also present health status of each computational container in the system and will present data regarding pending jobs in the Job Scheduler queue.

The Netty Server/Client is a module which provides communication means between Containers and Master. Netty is a client server framework which enables development of network
applications such as protocol servers and clients. *Master* has both the server to receive messages and the client to send messages.

**Worker Components**

The *Pregel Engine* is the core module of the *LudoGraph* platform. This module is our implementation of the *Pregel* programming model. It is responsible for executing user-defined computational code for each vertex encountered by the traversal scheme in the current computational step. Furthermore, this module cooperates with the *Message Observer* module to support the communication between vertices.

The *Traversal Guard* module determines how the program traverses the graph dataset. In the *Pregel* programming model, each vertex may participate in every computational step, in parallel. The *Traversal Guard* module is an essential part of the runtime sampling and custom graph traversal features (for examples, see Chapter 5). The module determines for which vertices computation will be executed at each step. The scheme for graph traversal is specific for each job and defined by the user.

The *Message Observer* module is based on the *Observer* design pattern. During computational steps of the *Pregel* programming model vertices can send messages to each other. This module is executed after the current computational step has finished and all sent messages are buffered in the local message buffer. This module will perform communication optimizations such as message aggregation, message combination (similar to the combiner phase in *MapReduce*) to improve the platform communication throughput.

The *Checkpoint Recorder* is an essential fault-tolerance module in the *LudoGraph* platform residing on the *Container* side. Its purpose is to record the state of the program and the dataset after each step of the *Pregel* programming model. After the current step $S$ finishes its execution, the current state of the dataset (values of vertices, graph structure etc.) and the state of the program (global variables, aggregators etc.) are stored into the underlying distributed file system. Thus whenever one of the containers experiences critical failure or some run-time error occur, program execution can continue from the last stored step $S$, without the need of repeating any steps or computation leading to step $S$. Because the *Pregel* programming model runs in super-steps. The platform will simply restore the last stored checkpoint and continue computation. In case of critical failure of one of the computational container, it may be required to request an additional container to keep the load balance or repartition data amongst “healthy” containers if the request can not be fulfilled.

The *ZooKeeper Communicator* is a module responsible for providing communication with the *ZooKeeper* service [20]. Many *LudoGraph* features, such as aggregators and global variables are implemented with the use of the *ZooKeeper* service, *ZooKeeper* has been shown to be robust and scalable. This module cooperates with the *ZooKeeper* API to create an abstraction layer for the users, which increases platform usability.

*Aggregators* are globally available key-value pairs, where each new value is aggregated with the previous one. They follow standard *CRUD* (Create, Read, Update, Delete) scheme. Each *Container* uses the *ZooKeeper Communicator* module to perform operations on the aggregators.
Global Variables work similarly to the Aggregators, however instead of aggregating values they act as a regular variable. These variables are accessible by each vertex in each computational step. The variable value is synchronized with the use of a FIFO scheme. FIFO scheme specifies that the first data to be received is the first one to be processed, regardless of the actual order of incoming data.

The Heartbeat Module works in cooperation with the Platform Status Observer module residing at the Application Master node. This module sends “heartbeats” (simple status messages) every configurable time $T$ to the PSO module. Thus the master can maintain the health status list of computational containers within the platform.

The Resource Utilization Recorder is another module which works in cooperation with the RUO module residing on the master node. This module records the resource utilization of the Container and transmits this information to the Resource Utilization Observer module residing on the master node. Thus the master has the up-to-date knowledge regarding platform resource utilization. This knowledge is used to determine if a job requires additional resources or if some resources can be scheduled to a different pending job or deallocated.

The Netty Server/Client is a module which provides communication means between Containers and Master. Each Container has both the server to receive messages and the client to send messages.

The HDFS Driver is a module which provides API for cooperation with the underlying HDFS storage system.

### 4-3-3 Platform Bootstrapping

The platform must be initialized before any LudoGraph job can be executed. This will be achieved by the Cluster Client (CC) component. The CC is responsible for requesting an initial computational container in which LudoGraph AM (see Section 4-3-2) will be executed. During the bootstrap process, the AM will request the initial resources as specified in the LudoGraph's configuration. Once the platform is initialized jobs can be executed. The configuration passed during the bootstrapping process will specify the platform’s global properties such as scheduling policy, provisioning policy etc. Once the platform is initialized these properties can not be changed or overwritten. Some of the platform properties specific for the job execution are allowed to be overwritten by the Job Client.

### 4-3-4 Job Execution

To execute a job, a Job Client (JC) has to be started on a user’s client. The JC will contact YARN’s RM and will be redirected to the LudoGraph’s AM which will be responsible for a proper job execution. The main functionality of the Job Client is to pass the job execution specification, job input dataset and optionally overwrite platform’s job specific properties, for example job provisioning policy and threshold.

Furthermore, the job will be executed in-memory to improve the platform performance. However as the Giraph platform evaluation has shown this approach has a significant limitation as to the size of the datasets which can be processed. Thus each job will have the option...
of spilling data onto a local disk in case the amount of memory available is insufficient to perform computation.

User can either wait for the end of job execution to receive the job execution statistics or the Job Client can be terminated and the job execution statistics will be stored in a special file in the output directory of the job.

### 4-3-5 Cloud Capabilities

The provisioning mechanism embedded in the LudoGraph will be triggered by the Resource Utilization Observer (RUO) component. The RUO component runs on the LudoGraph’s AM and receives metrics data from every container in the platform after every step. This solution will allow to carefully monitor resource usage and adjust resource usage to the actual need of every job.

To receive additional resources for a job, the AM receives an EVENT from the RUO which will trigger the process of requesting additional resources at first. The AM then contacts the RM for additional resources. However, this resource request will not be treated as a standard YARN’s platform resource request; to achieve this, YARN’s RM component has to be modified. After receiving this special provisioning request the RM will mark it as a top priority request and put it into the resource request queue. Once requested resources become available, newly allocated containers ids are returned to the AM, which starts additional tasks on these containers. To release unutilized resources, LudoGraph uses the resource deallocation mechanism already present in the YARN platform.

### 4-3-6 Pregel Programming Model Implementation Design

The Pregel programming model executes programs in a finite number of steps. User-defined code is executed for each vertex in the dataset which participates in the current step. Our
implementation follows *Pregel’s* design. However, we have modified the original design to allow run-time sampling and custom graph-traversal features. The *computational threshold* is a graph-traversal parameter specified by the user and used automatically by *LudoGraph* to filter out vertices. The high-level design is presented in Figure 4-3.

![Figure 4-3: High-level LudoGraph’s Pregel implementation.](image)

After the job is dispatched by the *Job Dispatcher* module and the graph data is initialized, *LudoGraph* begins to execute the steps of the *Pregel* programming model. However, before the first step starts, *LudoGraph* can execute an optional initialization step (with code provided by the user). During this optional step, initial messages can be propagated amongst vertices, global variables can be defined, etc. Once the initialization process is finished, each *Container*
iterates over its local vertex data. The iteration process is controlled by the Traversal Guard module. If a vertex passes the computation threshold, it will participate in a regular Pregel step; if not, it will be omitted in the current step. Once the termination condition is reached each, Container’s local task will perform a cleanup process, during which all intermediate data is deleted and the final output is stored onto the distributed file system. The clean up process leads to storage efficiency and process reliability.

4-3-7 Runtime Sampling and Custom Graph Traversal Design

The component responsible for runtime sampling and custom graph traversal features is the Traversal Guard component. As depicted in Figure 4-3, the Traversal Guard component determines if a vertex will participate in the computation step $S$. To utilize these features, the user has to define graph-traversal or sampling algorithm either, by implementing it or by choosing one of the already available traversals schemes in the platform.

The traversal scheme implementation follows the Pregel programming model, is executed in steps (called traversal steps) and allows message exchanges between vertices (traversal messages). From a high-level design perspective, custom graph-traversal is a regular Pregel-based program that controls the execution of another, user-defined Pregel program. All traversal-related computation for step $S$ is executed before main program computation. Thus, before actually executing step $S$ the platform first retrieves information regarding the traversal scheme in step $S$.

The LudoGraph platform already provides a library of common traversal-related methods. For example, a user might wish to perform computation only for vertices whose degree or other property is higher than a threshold $D$. For these situations, LudoGraph provides a simplified traversal scheme, which does not perform the traversal steps; instead the traversal is based on a simple boolean condition. The condition is a form of traversal guard (see Figure 4-3), which checks if the vertex passes the computation threshold.

4-4 LudoGraph’s Prototype Validation

In this section we present validation of our prototype of LudoGraph platform. Furthermore, we will compare LudoGraph’s execution time with Giraph’s execution time. Both of these platforms are dedicated to processing large-scale graphs, also both of them execute in a distributed environment. For the purpose of performing prototype validation experiments we use similar experimental setup as for other platforms evaluation (see Chapter 3 section 3-1). Unfortunately we limit the experimental algorithms and datasets to BFS algorithm and WikiVote_FCF, KGS_3_FCF, EUEmail_FCF datasets respectively (the rest of experimental setup was not changed). To obtain Giraph’s results for KGS_3_FCF datasets we perform additional experiments.

4-4-1 Validation of Cloud Capabilities

To validate LudoGraph’s cloud capabilities, we start job execution with only 10 containers, after executing 2 steps we increase the number of containers to 20, only to decrease it to 10
containers 2 steps before the end of the execution. Results of our experiments are presented in Figure 4-4.

![Breadth-first search execution with Cloud Capabilities](image)

**Figure 4-4:** LudoGraph’s (with cloud capabilities) and Giraph’s execution.

Our experiments indicate that *LudoGraph* executes faster than *Giraph*. Although the utilization of provisioning mechanism in *LudoGraph* adds *provisioning overhead*; we define provisioning overhead as additional time required to allocate and initialize additional containers, and the time required to repartition the input dataset.

### 4-4-2 Validation of Implementation of Pregel in LudoGraph

We validate *LudoGraph’s Pregel* implementation by executing *BFS* algorithm; during the execution *LudoGraph* does not utilize cloud capabilities, sampling feature, or custom graph traversal feature. In Figure 4-5 we present recorded execution times of *LudoGraph* and *Giraph* processing our validation experiments datasets.

Also this experiment indicates that *LudoGraph* achieves better execution time than *Giraph*.

### 4-4-3 Validation of Custom Graph Traversal

Custom graph traversal feature was validated by executing experiments with *LudoGraph’s* computation threshold. *LudoGraph* custom traversal computation threshold for these experiments was set to perform computation only for vertices with a degree equal or higher than 5. To ensure proper execution we change *BFS* initial vertex. In Figure 4-6 we present recorded execution times of *LudoGraph* custom graph traversal validation experiments. Since *Giraph* platform does not offer this functionality we will compare regular *Giraph’s* execution.
Figure 4-5: LudoGraph’s and Giraph’s Pregel implementation execution.

Figure 4-6: LudoGraph’s (with Custom Traversal) and Giraph’s execution.

This experiment as well indicate that LudoGraph outperforms Giraph’s execution. Although Giraph performs full BFS execution, while LudoGraph performs computation only for vertices
with a degree equal or higher than 5; this results in different number of \textit{Pregel} steps executed for the \textit{KGS\_3\_FCF} dataset.

\section*{4-5 Summary}

In this chapter we have presented the design of the \textit{LudoGraph} platform, which implements an extended \textit{Pregel} programming model.

We have presented three main novelties of the \textit{LudoGraph} platform, these are: cloud capabilities, improved \textit{Pregel} programming model implementation, and runtime sampling and custom graph-traversal. We believe that the design of \textit{LudoGraph} utilizes some of the best practices found in the platforms evaluated in Chapter 3. Furthermore, \textit{LudoGraph} increases the usability and expressiveness of the original \textit{Pregel} programming model, making large-scale graph processing easier.

Implementing \textit{LudoGraph} as an \textit{Application Master} within the \textit{YARN} platform was possible and relatively straightforward. As the \textit{YARN} platform is responsible for managing platform’s computational nodes on a global level, job-level management is left to the \textit{Application Master}.

Our prototype implementation of \textit{LudoGraph} shows promising results. We have successfully validated each of \textit{LudoGraph}’s novel features. Furthermore, we have compared obtained results with the \textit{Giraph} platform. Our experiments indicate that \textit{LudoGraph} achieves better performance than \textit{Giraph}. 
Chapter 5

Graph Sampling Functionality in LudoGraph

This chapter is dedicated to answering the third main research question (see Section 1-2) about the performance of sampling algorithms. These algorithm are elements of LudoGraph’s extension of the classic Pregel programming model (Chapter 4 in particular Section 4-3-7). We explain each of the sampling algorithms and we compare their execution in our experimental environment, in Section 5-1. We also present our findings regarding each of the sampling algorithms accuracy, in Section 5-2. In Section 5-3, we summarize this chapter.

5-1 Sampling Algorithms

An important research topic in large-scale data processing is the data sampling process. In the sampling process, a representative sample is chosen out of a whole dataset, then the sample is processed to calculate the needed graph properties. We define the representative sample as a sample which accurately holds all original graph properties. The sample is chosen by a sampling algorithm, typically based on some random condition. Sampling conditions which return representative sample can not be biased towards selecting any particular group of vertices or edges (for example highly connected vertices).

Sampling algorithms are very useful as they return a subset of a graph, which is much smaller than the original graph. These samples can be used an input dataset for graph processing algorithms for example to calculate graph properties. Without a sampling algorithm, graph properties have to be calculated on the whole dataset, which would significantly increase the execution time and the amount of resources required for data processing.

In our research, we have investigated two sampling algorithms: Metropolized Random Walk and Random Walk. These algorithms do not provide unique sample; thus, we have created a third sampling algorithm. Our Unique Metropolized Random Walk by design returns only unique samples, defined as samples which consist only of unique vertices. (Conversely we
define non-unique samples as samples that are not unique samples, that is, they include duplicate vertices.) Section 5-1-1 presents the Random Walk algorithm. Section 5-1-2 presents the Metropolized Random Walk algorithm. Section 5-1-3 presents our Unique Metropolized Random Walk algorithm.

5-1-1 Random Walk

*Random Walk* (RW) is one of the simplest sampling algorithms. Sample vertices are chosen by randomly choosing a neighbour of the current vertex. *RW* does not guarantee uniqueness of sampled vertices. The same vertex may be sampled several times. The pseudo-code for the *RW* algorithm is presented in Algorithm 6.

Algorithm 6: Random Walk algorithm

<table>
<thead>
<tr>
<th>Input</th>
<th>Vertex-based dataset and initial vertex and SampleSize</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Graph Sample</td>
</tr>
<tr>
<td>1</td>
<td>sample = 0;</td>
</tr>
<tr>
<td>2</td>
<td>while sample &lt; SampleSize do</td>
</tr>
<tr>
<td>3</td>
<td>next = random(neighboursList);</td>
</tr>
<tr>
<td>4</td>
<td>nextStep.set(next);</td>
</tr>
<tr>
<td>5</td>
<td>sample++;</td>
</tr>
</tbody>
</table>

5-1-2 Metropolized Random Walk

Stutzbach et al. [40] designed the *Metropolized Random Walk* (MRW) algorithm, which collects unbiased graph samples. The condition for choosing an unbiased sample vertex is a uniformly distributed random number $P$, which value is between 0 and 1. Number $P$ is compared with the ratio of the current vertex’s degree to random neighbour’s degree. If the ratio is greater than or equal to $P$, then the random neighbour becomes a sample vertex; otherwise the current vertex is added again to the algorithm’s output sample. Each vertex may be added more than once to the output sample. The *MRW* algorithm does not guarantee the uniqueness of vertices in the output sample. The pseudo-code of the *MRW* algorithm is
presented in Algorithm 7.

**Algorithm 7: Metropolized Random Walk algorithm**

**Input:** Vertex-based dataset and initial vertex and *SampleSize*  
**Output:** Graph Sample

```plaintext
sample = 0;  
while sample < SampleSize do  
  P = random(0,1);  
  randomNeighbour = random(neighbourList);  
  if \( p \leq \frac{\text{degree}(\text{currentVertex})}{\text{degree}(\text{randomNeighbour})} \) then  
    nextStep.set(randomNeighbour);  
    sample++;  
  else  
    nextStep.set(currentVertex);  
    sample++;  
```

5-1-3 **Unique Metropolized Random Walk**

In some cases, the analyst requires that the sample only includes *unique vertices*. For example, calculating the minimum spanning tree on a unique sample may provide more accurate results than on non-unique samples. Because unique samples are larger snapshots of the original graph than non-unique samples, a minimum spanning tree calculated on unique samples will provide more accurate results.

For the purpose of providing unique samples, we have designed the **Unique Metropolized Random Walk** (UMRW) algorithm. The algorithm is based on the MRW algorithm, but guarantees that the output sample will be unique. The simplified pseudo-code of the algorithm is presented in Algorithm 8.

**Algorithm 8: Unique Metropolized Random Walk algorithm**

**Input:** Vertex-based dataset and initial vertex and *SampleSize*  
**Output:** Graph Sample

```plaintext
sample = 0;  
while sample < SampleSize do  
  p = random(0,1);  
  if \( p \leq \frac{\text{degree}(\text{currentVertex})}{\text{degree}(\text{randomNeighbour})} \) then  
    nextStep.set(randomNeighbour);  
    sample++;  
  else  
    nextStep.set(currentVertex);  
    //repeat step – but do not sample vertex
```

*UMRW* in its design is very similar to the MRW algorithm: it also uses number of walkers to collect the sample, uses the same mathematical foundation for choosing the next vertex, etc. The **key difference** between these two algorithms lays in the method of collecting the sample.
While walkers in the MRW collect every vertex they encounter, the UMRW algorithm walkers collect only previously unsampled vertices. If a walker encounters a previously sampled vertex, it continues its graph traversal. However, this continuation may be time-consuming and requires a specific mechanism (see the following).

**Mechanism used to guarantee uniqueness of a sample**

To preserve uniqueness of the sample algorithm uses the STEP, MOVE, JUMP, and RANDOM JUMP mechanisms. The definitions of these mechanism are:

- **STEP** is executed when an unsampled vertex is encountered by a walker. It increases the size of the sample by adding the encountered unsampled vertex to the output sample.
- **MOVE** is executed when an already sampled vertex is encountered by a walker. Walker will continue its execution from this vertex in the next step.
- **JUMP** is executed when all neighbours of the current vertex are already sampled. Walker is moved to a random neighbour in hope that some of its neighbours were not yet sampled.
- **RANDOM JUMP** is executed when a walker has not performed any STEP in the past $N$ steps (random jump threshold). The walker is moved to a random already-sampled vertex in the graph, with the hope that the already sampled vertex will have unsampled neighbours. The walker continues sampling process from that new vertex and the random jump threshold counter is reset.

Whenever multiple walkers chose the same next unsampled vertex, only the first one can execute STEP; the others execute MOVE. All ties are solved with the First-Come, First-Served (FCFS) strategy. Furthermore, if a STEP is possible, it takes precedence before any other action; for example, even when the RANDOM JUMP threshold is reached but executing STEP is possible, the STEP will be executed and the threshold counter will be reset.

**Pros of the algorithm**

With the use of this mechanisms we have created an unbiased sampling algorithm which provides a unique graph sample. Our algorithm prevents walkers from choking on sparse parts of the graph, we define choking as a scenario when multiple walkers enter same vertex and each walker waits for its turn (step) to leave the vertex. In the UMRW algorithm walkers always make “progress” either by STEP or by MOVE. Furthermore because no walkers are being choked, the graph is traversed faster. Also in some cases of sparse graphs, walker can create sampled cliques, parts of graphs which are already sampled. If a walker enters such a group of vertices, it may take a long time for a walker to leave such a clique. Our algorithm solves this problem with the RANDOM JUMP mechanism. The mechanism was designed to force the walker to leave such a clique.

In our opinion the algorithm’s execution time can be decreased by adjusting the RANDOM JUMP threshold parameters per dataset. For example we have empirically determined that
high value of the RANDOM JUMP parameter is more suitable for denser graphs, while lower value is preferred for sparser graphs. Unfortunately in-depth research of the algorithm is outside of the scope of this thesis.

Cons of the algorithm

Unfortunately, our sampling algorithm is not without any drawbacks. We present these drawbacks in Section 5-2-3.

5-2 Performance Evaluation of Sampling Algorithms

In this section we will present performance evaluation of our experimental sampling algorithms. For each experiment we have used the same parameters. Sample size collected in each experiments is roughly 10% of the original graph size. In all of the distributed experiments we have used different number of walkers per dataset. Smaller datasets use 100 walkers, medium datasets use 1000 walkers, and the large datasets use 10,000 walkers. In contrast all Neo4j experiments use a single walker.

In Section 5-2-1 we present performance evaluation of the Random Walk algorithm. Sections 5-2-2 and 5-2-3 present performance evaluation of Metropolized Random Walk and Unique Metropolized Random Walk algorithms respectively. Section 5-2-4 presents our research results regarding sample accuracy collected by the evaluated algorithms.

5-2-1 Random Walk

The Random walk algorithm (RW) is an algorithm which can be used to collect an unbiased graph sample. It is an example of an unsupervised random algorithm. Unsupervised random algorithms are algorithms whose execution is not supervised by any logic based condition, rather they execute randomly. In Figure 5-1 we present our results regarding the RW algorithm execution time on our experimental graph datasets.

Our distributed implementations of the algorithm are using multiple walkers, where each walker executes its own algorithm. In case two walkers access the same vertex, both of them can record this vertex as sampled. The algorithm output is the union of all samples of all walkers. Neo4j’s sequential implementation uses only one walker.

It is important to notice that the overall amount of walkers per computational node is higher on the Hadoop platform. The reason for this difference lays in the fact that Hadoop uses 10 groups of computational nodes (a mapper and a reducer can be seen as a group) to perform calculations, while Giraph uses 20 computational nodes. This of course had an impact on the wall clock time of the evaluated platforms. However in our opinion the Hadoop’s overhead pose a much larger factor in the execution time difference.

As we expected Hadoop’s execution time is much longer than the Giraph’s execution time. Again this is a direct result of the Hadoop’s overheads mentioned in Chapter 3. As expected Neo4j achieves very good execution time for the sampling algorithm. Its execution time could be decreased even further, however the algorithm requires a random vertex as a sampling
traversal starting point. To fulfill this requirement, Neo4j iterates once per algorithm execution over the whole dataset to collect all vertices ID’s. This process increases the overall execution, as the size of the dataset impacts the execution time. This can be observed for the KGS_0 and KGS_1 datasets (which are among the largest datasets in terms of physical size).

5-2-2 Metropolized Random Walk

The Metropolized Random Walk algorithm (MRW) is used to collect an unbiased sample from graph datasets. It is an example of a random sampling algorithm with guarantee. Random sampling algorithm with guarantee is an algorithm that is executed randomly, however certain guarantees are enforced by design. In Figure 5-2 we present our results regarding the MRW algorithm execution time on our experimental graphs.

The number of walkers during execution time as well as their behaviour are identical to these presented in the Random Walk algorithm section.

Since the RW algorithm differs only in an unsupervised method of randomly choosing the next vertex when compared to the MRW algorithm, the results are similar to these of the RW algorithm.

5-2-3 Unique Metropolized Random Walk

The Unique Metropolized Random Walk (UMRW) is a sampling algorithm we have created by extending MRW sampling algorithm design. UMRW guarantees that each vertex in the collected sample will be unique.
When comparing execution times of the MRW and UMRW algorithms, we find that the execution time is noticeably different in favour of the MRW algorithm. The key factor which impacts the execution time of the UMRW algorithm is the sample uniqueness requirement. To achieve unique sample algorithm has to perform more steps during its execution than the MRW algorithm. Another factor which increases its execution is the possibility that a walker will enter a sampled clique. Upon entering such a clique there is a high probability that the walker will execute number of steps (before being forcibly removed from the clique by RANDOM JUMP mechanism) without making any progress. Furthermore during our experiments we have observed that the majority of walkers finish their execution quicker than the recorded execution time. Few walkers continue their traversal until collecting their full sample while the majority of walkers have already collected required sample, these “dangling” walkers increase overall execution time of the algorithm.

In Figure 5-3 we present our results regarding UMRW algorithm execution time on our datasets.

UMRW’s execution time is largely dependant on the random factor which guides the walkers. Each walker has to collect an unique sample of predefined size. As we have explained earlier few “dangling” walkers can significantly increase the overall execution time.

Giraph is once more the platform which achieves on average the best execution time in comparison to other platforms. Hadoop’s execution time is coupled with the number of iterations (I/O read and write overhead and job initialization overhead), as explained earlier the number of iterations depends on random factors. Neo4j does not achieve as good on execution time as with other algorithms, when coupled with Hadoop and Giraph. This is a direct result of parallelism used in other tested platforms, which Neo4j does not offer. Thus the overall execution time is much higher, as Neo4j gathers the sample with the use of a single walker.
5-2-4 Sampling Accuracy

One of the most important issues regarding sampling algorithms is their accuracy in preserving graph’s global properties. Thus we have performed accuracy experiments of our experimental algorithms. In these experiments we have used the Graph Properties algorithm extended with additional graph properties. These properties are: average vertex degree and degree distribution metric, we also compare graph’s local clustering coefficient property and sample uniqueness. We believe that with these properties we will be able to determine how accurately sample returned by sampling algorithm hold graphs properties, as if these properties are not accurate, no other properties will be. For the purpose of these experiments we have used the original datasets and their collected samples. Unfortunately, due to problems occurring for certain datasets (for performance problems see Appendix A) we have excluded the WikiTalk, KGS_0, KGS_1 and KGS_2 datasets from these experiments. Each accuracy metric presented is an average calculated based on the algorithm output for 10 collected samples for each dataset.

Our first accuracy metric is the sample uniqueness defined as the amount of unique vertices in the sample. From our collected sample we have filtered out duplicated values, leaving only unique vertices. We present the average uniqueness of each algorithm in Figure 5-4.

The average uniqueness of the RW is much higher than the one of MRW. The reason for this difference lays in the algorithm design. The MRW algorithm allows walkers to stay/sample the current vertex, if the conditions are fulfilled. The RW algorithm forces walker to change vertex at each step, which in return provides on average a higher uniqueness of the collected sample, as walkers travers a larger part of the graph than the MRW algorithm. The UMRW algorithm provides a unique sample by design.

Although the MRW algorithm does not provide a highly unique sample, it returns a sample
with the closest value of average degree metric to the original dataset. When compared with other sampling algorithms used in our experiments the \textit{MRW} algorithm outperforms its competitors. The results are presented in Figure 5-5.

For the average local clustering coefficient computation the \textit{UMRW} algorithm has provided on average the most accurate sample. The \textit{UMRW} algorithm has provided the most accurate
results for 6 datasets, MRW for 5 datasets and RW for 3 datasets, out of a total of 14 datasets. However in case of Slashdot dataset results returned by the MRW algorithm were inaccurate by an order of magnitude; this was the only dataset in our experiments for which MRW algorithm has provided highly inaccurate sample. Furthermore the UMRW algorithm on average was the most accurate algorithm in this experiment. The results are presented in Figure 5-6.

![Figure 5-6: Average local clustering coefficient.](image)

Degree distribution based experiments (Figure 5-7) have confirmed previous results, as the curve representing degree distribution from a sample provided by MRW is very close to the original dataset degree distribution. The RW and UMRW algorithm have similar results and they are both less accurate than the MRW algorithm.

From our sampling accuracy experiments we have concluded that the Metropolized Random Walk [40] algorithm while providing the least unique sample, it provides sample that is the most representative. In most of our experiments MRW algorithm has provided the most accurate results. UMRW algorithm while collecting only unique vertices, does not guarantee more representative sample than the MRW algorithm. The RW algorithm provides the least accurate sample of all tested algorithms, however on average its sample uniqueness is higher than the MRW algorithm.

### 5-3 Summary

The goal of the research regarding sampling algorithms was to investigate the performance of graph-sampling algorithms for real datasets representative sample for the fields of social networking, logistics, and online gaming (novel), etc. We have expressed the performance of graph-sampling algorithms in terms of execution time, accuracy, and uniqueness of vertices in...
Figure 5-7: Sample Degree Distribution (MRW and original dataset curves are nearly identical, thus the later is barely visible).

outputted sample. As many large scale graphs have a very large size, it might not be feasible to process these datasets in the original structure (either time constraints or insufficient resources). Thus, collecting a sample which is representative is a challenging and needed part of the graph processing domain. Furthermore, we have investigated if samples with unique vertices can provide more accurate results than samples with non-unique vertices.

To fully investigate sampling accuracy we have used two existing graph sampling algorithms and we have created an additional algorithm. The two used existing algorithms provide samples which are not unique, while the one created by us provides unique samples by design. By collecting both unique and non-unique samples, we were able to determine which type of sample is more accurate for the real datasets explored in this work. Our experimental algorithms represent both the unsupervised random algorithms, and the random algorithms with guarantees.

From our experiments, we have concluded that the most accurate sampling algorithm is the Metropolized Random Walk algorithm create by Stutzbach et al. [40]. It has provided the most accurate sample in the average degree experiment, was second in the average local clustering coefficient experiment, and has provided very accurate samples for the degree distribution experiment (which was nearly 100% accurate). All samples provided by the MRW algorithm were also accurate when compared with original results; only in a few instances the difference was significant.

Furthermore, we have concluded that a unique sample does not guarantee more accurate results and is not always favorable over not unique samples (degree distribution experiment). However, in certain datasets a unique sample indeed has provided more accurate results than non-unique samples (the local clustering coefficient experiment).

The algorithm which achieves the best execution time in our experiments is the Metropolized Random Walk algorithm. The MRW outperforms Random Walk algorithm in our experiments, however the time difference is relatively small in comparison to the total execution
time of the algorithms. *Unique Metropolized Random Walk* achieves poor performance when compared with other two algorithms, this is a result of the unique sample guarantee. Even if majority of walkers finish their execution, some may still continue the process of sampling, which leads to increased total execution time.

Our average sample uniqueness experiments indicate that *Random Walk* algorithm provides sample with better vertices uniqueness than *Metropolized Random Walk*. RW output sample uniqueness ranges from 51%-78% (depending on dataset). MRW output sample uniqueness ranges from 5%-49% (depending on dataset). RW algorithm in contrast to MRW algorithm forces walkers to enter new vertices at each computational step, which leads to increased sample uniqueness. *Unique Metropolized Random Walk* algorithm provides unique sample by design.
Chapter 6

Conclusion

In recent years, large-scale graph processing has gained significant attention from research and business. Graph dataset sizes are growing with current size, reaching sizes petabytes. Companies such as Facebook and Google are at the frontier of large-scale graph processing, but their solution platforms may not address the needs of small and medium enterprises.

At the beginning of this thesis we asked three research questions, for which we have tried to find answers. These questions were:

- What is the performance of current graph processing platforms, for typical real datasets algorithms?
- Is it possible to design a graph processing platform with cloud capabilities and sampling features?
- What are the accuracy, the performance, and the ability to provide unique vertices of graph sampling algorithms, for typical real datasets?

Now we will briefly summarize our findings for each of the research questions.

Platform Evaluation

We have evaluated three distinct large-scale data processing platforms. We have chosen two graph-processing platforms (Neo4j and the Pregel-based Giraph) and a single large-scale general data-processing platform (the MapReduce-based Hadoop). We have tried to determine what benefits specialized graph-processing platforms offer over generic data processing platforms. During our research we have added new knowledge about Big Data Analytics and some of our work was included in How Well do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis: Extended Report [14] article.

From the evaluation we have performed, we have learned that not only the programming model influences platform performance, but also the platform design. We have concluded that
specialized graph processing platforms offer higher usability for expressing graph processing algorithms over generic data processing platforms. However, the design of the platform has a profound impact on its performance: the Giraph platform executing in-memory has a significant advantage in terms of performance over the Hadoop platform, but the in-memory processing limits the dataset size which can be processed by the platform (the size can not exceed the total amount of memory available in the cluster). Hadoop can process any dataset size (unless it exceeds the disk space in the cluster), but its performance is much worse than the performance of Giraph and graph-processing algorithms are not as easy to express as in the Giraph platform. An interesting case is the desktop-based Neo4j platform. Its obvious disadvantage is the lack of parallelism and the resource limitation of a single commodity desktop computer; however, for small datasets its performance has been proven to be equal or even better than the performance of the distributed Giraph platform. Both of Neo4j and Giraph achieve high performance due to in-memory processing and specific execution mechanisms.

**LudoGraph**

Since there are only few specialized large-scale graph processing platforms and all of them have performance and/or design drawbacks, we have taken up the challenge of designing a new specialized large-scale graph-processing platform. Ideally, our platform should offer all the benefits of other platforms with none of their known drawbacks. Thus we have designed the LudoGraph platform. LudoGraph is a new platform with cloud-based system and sampling features for graph processing. Some of this work will be extended as open-source and as ICDE'14 article.

LudoGraph’s platform design focuses not only on the performance issues but also on the improvement of platform usability. The platform implements an extended Pregel programming model, including support for directed graphs, support for PUSH and PULL mechanism for message exchange, support for graph sampling and custom graph traversal, etc.

Because creating a new distributed large-scale data processing platform is a difficult and time-consuming task, we have chosen the YARN [21] platform as the basic resource manager of LudoGraph. YARN is a cluster-based platform which was designed to allow to create new programming models and processing schemes without the need of implementing the platform management functionality. YARN manages platform resources at the global level, leaving the job level to be managed by the Application Master. LudoGraph embraces YARN’s flexibility and introduces the Pregel programming model into YARN’s ecosystem.

To achieve good performance, LudoGraph uses some of the best solutions found in all of the evaluated platforms. It offers the performance of in-memory processing without the memory limitations forced by the Giraph platform. It also allows to dynamically allocate resources required for the dataset processing with its novel cloud-based provisioning mechanism. No other platform offers such a combination of features. The cloud features allow LudoGraph to serve small and medium enterprises who value efficiency (performance vs cost).

Furthermore, LudoGraph offers for specific use-cases improved usability, when compared to the platforms evaluated in this thesis. LudoGraph provides an API which offers sampling algorithms and custom graph-traversal algorithms as an abstraction.
Sampling Algorithms

In some cases, the dataset size and complexity lead to large computational requirements. This issue can be overcome with the use of sampling algorithms for use-cases when a loss of accuracy can be tolerated. Furthermore, we have designed a new sampling algorithm for specific use-cases; when unique sample is required. In this thesis we have investigated the accuracy and other performance aspects of three sampling algorithms, thus adding new knowledge.

Among the evaluated algorithms, *Metropolized Random Walk* and *Unique Metropolized Random Walk* algorithms are random algorithms with guarantees and the *Random Walk* algorithm is an unsupervised random algorithm. We have designed the *Unique Metropolized Random Walk* algorithm to provide new unbiased sampling functionality: samples with unique vertices.

When processing only a subset of vertices from the graph, the subset’s properties may be different than the original graph’s properties. This may lead to inaccurate or invalid results. Thus with a series of experiments we have evaluated the accuracy of each of the sampling algorithms. Our evaluation has proven that a good sampling algorithm can provide an accurate sample which will hold the global graph’s properties. Furthermore we have compared accuracy of a unique sample and non-unique sample. Our results show that the unique sample does not provide better accuracy than a good not unique sample. Thus, our algorithm remains useful only for use-cases where the uniqueness of vertices in the sample is mandatory.
In this chapter we present our ideas for future improvements of our platform evaluation methodology, LudoGraph platform design, and possible improvements of our UMRW sampling algorithm.

**Platform Evaluation**

Our platform evaluation methodology was limited by the upper-bound job threshold; all jobs executing longer than 60 minutes were terminated. Although it is possible to evaluate platform with this limitation, in our opinion removing this threshold could result in more accurate results. By ensuring execution of all jobs on all datasets, regardless the execution time, we could achieve broader picture of each evaluated platform, and we could find more corner cases (as FFM algorithm execution).

Another improvement of our platform evaluation methodology might be increasing the datasets size. Our platform evaluation was performed with datasets from many distinct fields and with different properties. Although this approach has ensured that our evaluation was not biased, all our datasets were smaller than 400MB (except dataset used in vertical scalability experiments). By increasing the dataset size we could investigate more thoroughly the actual computation time of each evaluated platform, also the resource utilization evaluation would be more accurate, as larger datasets will require more intensive resource utilization.

In our evaluation we have evaluated three platforms. However as we have mentioned in Chapter 1, there are many large-scale data/graph processing platforms. To continue our research on platforms performance we could use our methodology to evaluate different platforms. With these new platform we would gain more in-depth view of currently available large-scale data/graph processing platforms.

Furthermore as every platform cooperates with an underlying storage system, performing evaluation of these storage systems may provide even more in-depth analysis of each platform. In our thesis we have performed data ingestion experiments for the HDFS distributed file system and the Neo4j’s graph database (see Chapter 3 section 3-2-6). However there are other
popular storage systems such as: HBase [41], Cassandra [42], etc. Also with the demand for fast storage systems, SSD-based disks are gaining popularity. Comparing SSDs-based storage systems with HDDs-based storage systems may provide interesting results.

**LudoGraph**

Although LudoGraph originally was designed to support only the YARN platform, in our opinion it can become a vendor independent cloud platform. The main components of LudoGraph are self contained (they cooperate only with other LudoGraph based components); for example the Pregel execution engine. In our opinion we can achieve this goal by creating a number of interfaces which will act as mediators between the underlying cluster platform and LudoGraph. To port LudoGraph to another platform one would have to implement two components, which will provide the required functionality to LudoGraph. These components are Distributed File System Driver component and Resource Management component. The DFS Driver component would be responsible for performing any I/O related operations. The RM component would be responsible for managing resource allocation and deallocation process.

Another improvement might be the possibility to dynamically switch between different programming models. YARN offers possibility to create multiple different programming models which will work under a single platform. Our idea is to give a user the ability to dynamically switch between them within a single program, without the need of complex implementation. An example case scenario for this functionality might focus on using MapReduce and Pregel programming models. Many graph datasets are stored in the edge format (rows of edges), such a dataset would be difficult to process in a Pregel programming model, however it is well suited for the MapReduce model. With dynamic programming model switching the user could specify that the input should be processed by a MapReduce model, which would output vertex based format. This new output will be pipelined to the Pregel Engine as an input, which will be processed by the main computational algorithm.

Some of the more complex graph processing algorithms consists of multiple stages (many smaller algorithms). Some of the stages might provide better results when processed synchronously and some when processed asynchronously. Thus in our opinion it would be beneficial to give an option to the user to dynamically switch between synchronous and asynchronous execution model. Running different stages with a different execution model may result in an overall execution time decrease.

The current LudoGraph’s design supports only vertices as a basic parallelization unit. We define parallelization unit as a data structure which can not be shared between threads/processes, meaning it can be processed only by a single thread/process. Social networks or gaming oriented companies might be more interested in processing larger structures than a vertex. Thus in our opinion it would be beneficial to increase the abstraction level for the parallelization units. For example in gaming communities a single community of dedicated players is more important than a single player. Performing computation on a community level might provide better results in some cases. Currently to achieve this users first have to implement a community detection algorithm and then wrap each community in a vertex form. LudoGraph should implement a number of predefined algorithms which would compute different parallelization units, allowing the user to concentrate only on implementing the main algorithm.

Marcin Biczak

Master of Science Thesis
Data locality is a very important issue in distributed computation as it has a significant impact on the overall execution time. Currently LudoGraph achieves data locality by executing workers on the same (or as close as possible) computational nodes as the data is stored. However this solution does not take under consideration the neighbourhood of each vertex; most vertices communicate mostly with their direct neighbours. With the use of Ghost Zone [43] optimization, a well known strategy in the parallel computation field, we could significantly decrease the communication between vertices.

Current support of the cloud capabilities in the LudoGraph platform is rather simple in its nature: it allows users to allocate and deallocate resources on demand during the job execution. However, the user still has to pay attention to different metrics in order to fully utilize allocated resources. Thus, in our opinion adding an Intelligent Scheduler which would perform job resource management for the user is an interesting improvement for the platform. The Intelligent Scheduler will allow users to specify if the job should execute with the maximum resource utilization, concentrating on the job execution time, or to focus on the cost metrics, increasing the computation time but decreasing the cost of the job.

### Sampling Algorithms

Our UMRW sampling algorithm has a significant drawback of “dangling” walkers (issue which we have raised in the conclusion of Chapter 5). Even though the majority of walkers have collected their sample, few remaining walkers continue their process of collecting sample, which results in the increase of the overall execution time. In our opinion, addressing this issue would significantly improve the execution time of the UMRW algorithm.

Another possible improvement of the UMRW algorithm is the improvement of the RANDOM JUMP mechanism and more thorough research on the RANDOM JUMP threshold. Currently, when the RANDOM JUMP threshold is reached, the walker “jumps” to random already-sampled vertices, to avoid progress starvation (more details can be found in Chapter 5). However the algorithm does not ensure that the jump destination vertex is part of new clique. If each sampled vertex would hold the number of unsampled neighbours, the jump destination vertex could be picked based on this knowledge instead of choosing random sampled vertex, ensuring immediate progress. Furthermore, we have empirically determined that sparser graphs benefit from lower RANDOM JUMP threshold and denser graph benefit from higher RANDOM JUMP threshold. Investigating this relation more thoroughly and determining optimal RANDOM JUMP threshold for graphs with different density would improve the execution of the UMRW algorithm.
Appendix A

Hadoop’s limitations for the Graph Properties algorithm execution

In this appendix we will describe in detail issues which occurred during Hadoop’s performance evaluation; specifically executing the Graph Properties algorithm. The appendix is structured as follows: in Section A-1 we present high level code for the Graph Properties algorithm, we mostly concentrate on the second job; as this job is the main source of computation issues. Section A-2 presents a detailed problem definition. Section A-3 presents our experiments regarding Hadoop platform tuning. Section A-4 presents available algorithmic solutions. Section A-5 presents a theoretical algorithmic solution. We present our conclusion in Section A-6.
# A-1 Graph Properties MapReduce Implementation

In Algorithm 9 we present high level pseudo-code of *Hadoop’s Graph Properties* algorithm.

**Algorithm 9:** MapReduce algorithm implementation.

**Input:** Graph dataset (any format)
**Output:** Graph properties

// JOB_0 Create Vertex-based data structures. Format dependent.
// JOB_1 Create NodeNeighbourhood-based data structures.

1. map(Record *input*)
   2. vertex = new Vertex(*input*);
   3. foreach *n* : vertex.getNeighbours() do
      // outputs vertex id and destinations of outgoing edges
      // in format nodeId@[nodeId, ,]
      4. output(*n*, vertex.toOutput());
   5. }
6. reduce(Key *n*, VertexNeighbour [vertices])
   7. output(new VertexNeighbourhood(*n*, [vertices]));
   8. }

// JOB_2 Calculate graph properties.
9. map()
10. perform vertex properties calculations;
11. }
12. reduce()
13. perform graph average properties calculations;
14. }

The problem arising from the algorithm execution can be traced to the execution of the second job (Job_1). During its execution each vertex has to sends its out-edges to all of its neighbours (both in and out). For small degree vertices this poses no issues. However once a vertex with high degree is encountered, the amount of information that has to be transferred increases significantly. A detailed description of the data size growth can be found in the next section.

## A-2 Problem Definition

When it comes to calculating the Local Cluster Coefficient (LCC) of a vertex, degree plays a crucial role both in terms of time complexity and space complexity. In Table A-1 we present degree metrics for three selected datasets. These datasets are the ones which we referred to in Chapter 3 Section 3-2-1. We found no feasible solution of processing these datasets, with the resources at our disposal.

Equation presented in Figure A-1 determines the amount of data which has to be transferred by a single vertex during the *Map* phase of the second job. The output size is given by:

where \( S \) represents total output size of a vertex, \( N \) represents the number of neighbours connected via outgoing edges, \( I \) represents the number of bytes used to store a single vertex.
### Table A-1: Number of vertices with degree value within different ranges.

<table>
<thead>
<tr>
<th>Degree Metric</th>
<th>WikiTalk_FCF</th>
<th>KGS_0_FCF</th>
<th>KGS_1_FCF</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;10,000; 20,000</td>
<td>21</td>
<td>41</td>
<td>77</td>
</tr>
<tr>
<td>&lt;20,000; 50,000</td>
<td>3</td>
<td>27</td>
<td>21</td>
</tr>
<tr>
<td>&lt;50,000; 70,000</td>
<td>0</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>&lt;70,000; 100,000</td>
<td>0</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>&lt;100,000; 200,000</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>max</td>
<td>100,032</td>
<td>120,739</td>
<td>96,278</td>
</tr>
<tr>
<td>avg</td>
<td>76,491</td>
<td>28,947</td>
<td>135,698</td>
</tr>
</tbody>
</table>

\[ S = (N \times I + I) \times V \]

**Figure A-1:** Size of data to be transmitted by a vertex.

**id** and **V** represents the total number of neighbours.

Let us use this equation on a theoretical vertex. Our theoretical vertex has the following properties: degree value of 100,000, 50% of edges are outgoing and each vertex id is stored with the use of 6 bytes. When we apply our equation on this theoretical vertex, the total amount of data which has to be transmitted by this vertex is over 30 GB. This simple example shows the scale of the problem.

To cope with massive data size growth (between **Map** phase and **Reduce** phase) **Hadoop** has to spill data to local disk during map phase. The spill process results in significant performance decrease. If it is combined with reducers sort-merge process the total performance of the job execution is decreased even further.

### A-3 Hadoop Tuning

One of our very first attempt to solve the issue of datasize “explosion” was a custom tuning of **Hadoop** platform per job. **Hadoop** offers a number of platform configuration parameters and mechanisms to help its users tune the platform to their needs, these are the parameters we have used in our attempts:

- **io.sort.mb** - Higher memory-limit while sorting data (addresses data spilling issue).
- **fs.inmem.size.mb** - Larger amount of memory allocated for the in-memory file-system used to merge map-outputs at the reducers (addresses sort-merge performance).
- **io.sort.factor** - Number of streams to merge at once while sorting files (addresses sort-merge performance).
- **dfs.block.size** - **Hadoop** block size. Smaller block size results in smaller map phase output.
• **Compression** - Compress intermediate datasets. Decreases size of data which has to be transmitted to the reducers.

• **SequenceFiles** - Reduce the amount of data that needs to be materialized between map phase and reduce phase.

As tuning test datasets we have chosen the WikiVote FCF, EUEmail FCF and WebGraph FCF datasets. We have done so for the following reasons:

• Smallest datasets available. Small size will result in short computation time, which will allow to observe results instantly.

• Datasets represent both sparse graphs (WikiVote FCF local clustering co-efficient 0.07986) and denser graphs (EUEmail FCF and WebGraph FCF local clustering coefficient 0.24709 and 0.26996 respectively).

• Reasonable graph size (number of vertices and edges) for test purposes (smallest 7000 vertices and 100,000 edges, largest 255,000 vertices and 2,200,000 edges).

In our opinion any performance increase visible for these datasets, will be also noticeable in other datasets.

In our experiments with the tuning parameters, we were adding new parameters to the test configuration which returned the best result so far. We have performed our tuning experiments in the *pseudo-distributed* mode (single node cluster, capable of parallel execution of 2 mappers and 2 reducers, 1.5GB of memory per task). We have used the following configurations:

• Configuration 0 - untuned version of the program.

• Configuration 1 - sort-merge buffer size increased from 100MB to 756MB.

• Configuration 2 - added compression (BZip2Codec) to configuration 1.

• Configuration 3 - increased mapper sort buffer size from default value of 100MB to 768MB and increased the number of sort-merge streams, in addition to configuration 1.

• Configuration 4 - decreased block size by factor of 2, in addition to configuration 1.

• Configuration 5 - further increased the spill and sort-merge buffers size to 1024MB, however our test datasets were unable to fully utilize this new buffer size.

• Configuration 6 - usage of SequenceFiles.

Our experiments have proven that increasing the sort and sort-merge buffers size combined with increasing the number of sort-merge streams result in the best performance (that we have achieved). Other tuning parameters have not resulted in further performance increase. However in our opinion the SequenceFile experiment and compression experiment could not fully utilize the mechanisms put to the test. The test datasets were too small to use these tuning parameters; the overhead produced by the parameters was too large for our small test datasets. In Figure A-1 we present results of each experimental configuration.
In an attempt to solve the Hadoop problem, we have turned our attention to an algorithmic solution. Sergei Vassilvitskii et al. [44] present one of the possible solutions, a MapReduce algorithm called NodeIterator++ for counting triads. The pseudo-code for the algorithm is presented in Algorithm 10.

In Algorithm 10 $\Gamma(v)$ is a set of neighbours of vertex $v$, $V$ denotes all set of all vertices in the dataset, $S$ is a set of values passed to reducer for certain key, and $u, w, v$ are vertices ids.

Unfortunately in our opinion the NodeIterator++ has some limitations in its design. The algorithm requires the dataset to have duplicated edges; even though the very first instruction in the Mapper ignores half of the data. Duplicating edges result in a bigger physical size of the dataset. Furthermore, the algorithm works only for undirected graphs, which in our opinion is a great disadvantage.

Though the NodeIterator++ algorithm has certain limitations and it was not designed to calculate local cluster coefficient (LCC); in our opinion with a few changes it can be transformed into an LCC counting algorithm, which will work both for directed and undirected graphs. Thus we have designed the NodeIterator+2 algorithm, which is built on top of the NodeIterator++ design.

The NodeIterator+2 algorithm at its core uses the same idea as the NodeIterator++: it generates permutations and checks if the permutation exists. If a permutation for certain vertex $n$ exists, this means that an edge between neighbours of $n$ exists. These edges are summed up and used in the equation for calculating LCC. We do not provide pseudo-code for the algorithm as the algorithm was created only for the tests purpose.

Though it is indeed an improved version of the NodeIterator++, it has not solved the overall...
Algorithm 10: NodeIterator++ algorithm

**Input**: Graph dataset (edge based format)

**Output**: Number of triangles in the graph

1. Map 1: Input: $\langle (u, v); \emptyset \rangle$
2. if $v > u$ then
   3. emit($u; v$);
3. }
4. Reduce 1: Input: $\langle v; S \subseteq \Gamma(v) \rangle$
5. foreach $u, w \in S$ do
6. emit($v; (u, w)$);
7. }
8. Map 2: {
9. if Input of type $\langle v; (u, w) \rangle$ then
10. emit($((u, w); v)$;
11. if Input of type $\langle (u, v); \emptyset \rangle$ then
12. emit($((u, v); \$)$;
13. }
14. Reduce 2: Input: $\langle (u, w); S \subseteq V \cup \{\$\} \rangle$
15. if $\$ \in S$ then
16. foreach $v \in S \cap V$ do
17. emit($v; 1$);
18. }
19. }

problem which we have hoped to solve. The reason for that lays in the number of permutations which are generated for certain vertices. During the permutation creation process we use the UniqueCollection data structure to avoid duplicated permutations. Unfortunately the number of permutations is very large for certain vertices (highly connected vertices), which results in larger intermediate results than the memory available.

A-5 New Algorithm

Each algorithm we have experimented with produces large amounts of intermediate datasets, which have to be spilled, copied and sort-merged. This causes the Hadoop platform to “choke” on these processes making it unfeasible for larger datasets then toy ones. A New Algorithm is designed to overcome these drawback.

The New Algorithm should create vertex neighbourhood in an iterative process. Based on an input dataset size the algorithm creates the vertex neighbourhood not from a whole dataset but rather from the dataset chunk, we define chunk as a fragment of the whole dataset. The main idea is to take a chunk of an input dataset (with relatively small size in comparison to the whole dataset) and create the vertex neighbourhood for that chunk. After completion the algorithm takes a next chunk and performs the same operation. This process continues until all chunks are processed. In theory this solution should eliminate spills or at least decrease
them significantly, also the sort-merge process will operate on much smaller input as data for a specific vertex will be transferred in multiple iterations, not in a single one.

Though in theory this approach should work it has two significant drawbacks, both caused by Hadoop’s internal dataset sort process.

- Because vertices are sorted in ascending order after the first job of Graph Properties algorithm, it is possible that all “neighbour data transmissions” will take place within a chunk, thus the spill, sort-merge problem can still occur and cause performance decrease (even though it should be decreased).

- Sorted vertices will cause the current iteration chunk to be present on a single mapper (most probably, though it will depend on the chunk size, at most 2 mappers utilized). Thus during each neighbourhood iteration at most 2 mappers will be utilized out of a total of N mappers.

To solve these drawbacks chunks could be treated not as a continuous set of vertices but rather as a list of single elements divided by n non-chunk elements. This would enforce rather uniform distribution of active vertices per iteration.

A-6 Conclusion

Although throughout this appendix we have presented extensive research on how to improve Hadoop’s poor performance regarding the GP algorithm for selected datasets, in our opinion Hadoop is a still sound platform for performing these type of calculations. Even with its disadvantages Hadoop in our opinion still outperforms other tested platforms; namely Neo4j and Giraph. Giraph is limited by the memory available in the cluster and can not process datasets larger than the combined memory size. Neo4j does not have any form of parallel processing and as such will always have worse performance than any distributed/parallel platform when processing large-scale datasets. Hadoop even with its limitations is capable of processing datasets, which can not be processed by Giraph in a time shorter than Neo4j. This alone makes it a first choice platform in computing graph properties.
Bibliography


---

Master of Science Thesis                          Marcin Biczak


