UNIVERSITY OF TASMANIA

DOCTORAL THESIS

Dynamic Graph Partitioning in Streaming Manner

Author:
Md Anwarul Kaium PATWARY

Supervisor:
Professor Byeong KANG and
Dr. Saurabh GARG

A thesis submitted in fulfilment of the requirements for the degree of

Doctor of Philosophy

in the

School of Technology, Environments and Design

May 4, 2020

Statement of Co-Authorship

The following people and institutions contributed to the publication of work undertaken as part of this thesis:

Candidate- Md Anwarul Kaium Patwary, School of Technology, Environments and Design
Author 1 -- Saurabh Garg
Author 2 -- Byeong Kang
Author 3 – Sudheer Kumar Battula

Contribution of work by co-authors for each paper:

PAPER 1: Located in Chapter 5
Md Anwarul Kaium Patwary, Saurabh Garg and Byeong Kang, "Window-based streaming graph partitioning algorithm" Proceedings of the Australasian Computer Science Week Multiconference, ACSW-2019, Sydney, 2019

Author contributions:
Designed Algorithm: Candidate, Author1
Implemented, experimented: Candidate
Analysed the data: Candidate, Author 1
Wrote the manuscript: Candidate
Supervised in writing and developing idea: Author1, Author2
Contributed in reviewing the project and monitoring the progress: Author 1, Author 2

PAPER 2: Located in Chapter 6
Md Anwarul Kaium Patwary, Saurabh Garg and Byeong Kang, " SDP: Scalable Real-time Dynamic Graph Partitioner in the Cloud" IEEE Access Journal (Submitted)

Author contributions:
Designed Algorithm: Candidate, Author1
Implemented, experimented: Candidate
Analysed the data: Candidate, Author 1
Wrote the manuscript: Candidate
Supervised in writing and developing idea: Author1, Author2
Contributed in reviewing the project and monitoring the progress: Author 1, Author 2

PAPER 3: Located in Chapter 4
Md Anwarul Kaium Patwary, Saurabh Garg, Sudheer Kumar Battula and Byeong Kang, "ASGP: Auto-Scaling for Streaming Graph Partitioning in Cloud" Journal of Computational Science (Submitted)

Author contributions:
Designed Algorithm: Candidate, Author1
Implemented, experimented: Candidate, Author 3
Analysed the data: Candidate, Author 1
Wrote the manuscript: Candidate
Supervised in writing and developing idea: Author1, Author2
Contributed in reviewing the project and monitoring the progress: Author 1, Author 2

We, the undersigned, endorse the above stated contribution of work undertaken for each of the published (or su er-reviewed manuscript c nt uting to this thesis:

Signed:			
	Md Anwarul Kaium Patwary **Candidate** School of Technology, Environments and Design University of Tasmania	Byeong Kang **Primary Supervisor** School of Technology, Environments and Design University of Tasmania	Jason Byrne **Dean** School of Technology, Environments and Design University of Tasmania
Date:	02/09/2019	02/09/2019	02/09/2019

Authority of Access

I Md Anwarul Kaium Patwary declare that, this thesis is not to be made available for loan or copying for one year following the date this statement was signed. Following that time the thesis may be made available for loan and limited copying and communication in accordance with the Copyright Act 1968.

Md Anwarul Kaium Patwary
School of Technology, Environment and Design
University of Tasmania

Declaration of Authorship

UNIVERSITY OF TASMANIA

Abstract

College of Science and Engineering

School of Technology, Environments and Design

Doctor of Philosophy

Dynamic Graph Partitioning in Streaming Manner

by Md Anwarul Kaium PATWARY

This dissertation addresses the problem of dynamic graph partitioning in a streaming manner in the cloud. This problem applies to real-world graph applications such as PageRank, Social Networks, Shortest Path and so on. The scale of graphs of these applications has increased to such a degree that a single machine is not capable of efficiently processing large graphs. Thereby, efficient graph partitioning and wise resource allocation are necessary for these large graph applications.

At the beginning of this study, this dissertation evaluates two existing streaming graph partitioning algorithms in the cloud. After having completed the empirical study of these algorithms in the cloud machines, we identified the following research problems: 1) There are no existing streaming graph partitioning methods to find an optimised number of machines and scale the resources, as per the demands of an ever-increasing graph dataset. 2) How can we minimise the number of edge-cuts while balancing the load in a streaming manner in the cloud environment? 3) How can we use dynamic graph partitioning in a streaming manner to reduce the edge-cuts and the load imbalance during the partitioning? We also address the scaling of the resources as per the demands of dynamic graph data.

Streaming graph partitioning is a variant of traditional graph partitioning which accepts graph input in a one-pass manner. This partitioning technique was introduced to overcome a memory bottleneck issue in traditional graph partitioning. In streaming graph partitioning, it is necessary to utilise the resources as per the demands of graph data stream. This thesis proposes an auto-scaling algorithm to determine the required number of machines, based on the upcoming stream data rate and the service time at the worker machines. The proposed method helps to minimise the cost and provide the best use of cloud resources by allocating the number and types of machines wisely. Once the optimised resources and costs are fixed, this study looks into the problem of graph partitioning in a streaming manner, with the aim of minimising inter-machine communication and reducing the computational load imbalance as much as possible. In order to achieve these goals, we propose a window-based, streaming graph partitioning algorithm. The proposed method utilises sliding window technology with a partitioning strategy and the load balancing method as well.

After exploring the streaming graph partitioning with the static datasets, we studied the problem of the dynamic behaviour of graph datasets. This problem was how to partition dynamic graph data while minimising the edge-cut and keeping the computational load imbalance to a minimum. In addition to this partitioning technique, we also proposed an auto-scaling algorithm which adaptively scales in and out the machines, as per the demands of the computational cost.

Acknowledgements

First and foremost, I would like to thank my Almighty Allah for giving me the mental strength, and courage to overcome the difficulties throughout the last 4 years.

I am very grateful to my supervisors Dr Saurabh Garg and Professor Byeong Kang for their guidance, and direction and for the time they spent with me. They were very supportive and, helpful, and I have been well-guided by them to complete this study.

My gratitude goes to my parents for their sacrifices and wonderful support in this journey. I would also like to extend my thanks everyone who supported me along the way to complete this dissertation.

I also thankful to the members of Smart Services & Systems group for their valuable comments and technical support.

I would like to thank my friends particularly Sudheer Kumar Battula for his kind help and support.

Md Anwarul Kaium Patwary

Tasmania,Australia

March 2020

Contents

List of Figures

List of Tables

Dedicated to my parents

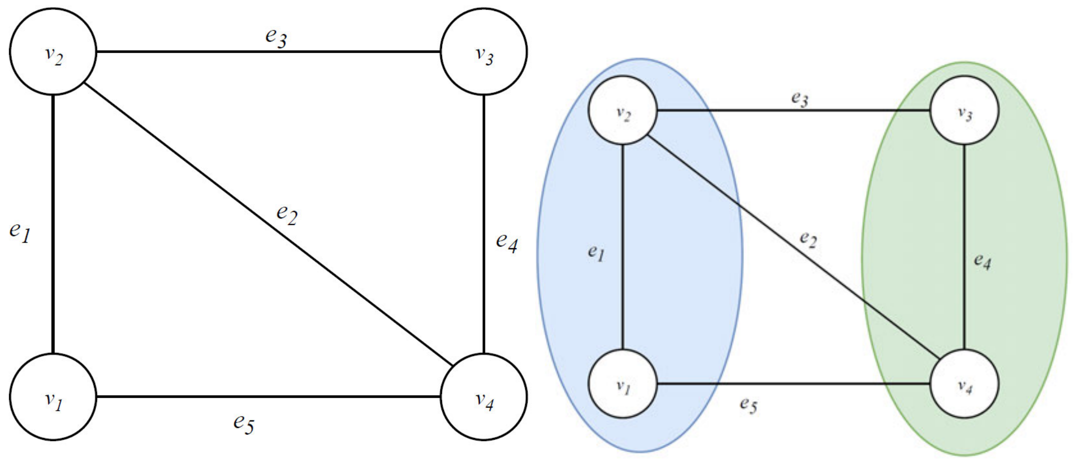
Chapter 1

Introduction

1.1 Research Problem and Motivation

At present, graph data are huge, and increasing incrementally. These huge datasets are: knowledge graph, web data, social network data, and biological network to name a few. These data can be represented as a graph, for example, in a social network; a user can be represented as a vertex and their relation to other users are edges. In addition to growing data in a network, these real-world networks or graphs change continuously over time, by adding or removing vertices or edges that generate a large dynamic graph. For example, online activity and interactions from electronic communication, social media and content sharing are ever-changing. These processes produce a huge amount of continuous, interaction data over time, which is represented as dynamic graph. A recent statistic [19] shows that Twitter has over 43 million users and there are more than 1.5 billion social relationships over this network. This indicates that the trend of exponential growth in data is significant. In graph streams, individual edges of the underlying graphs arrive sequentially in a stream, unlike traditional graph-data which has fixed number of vertices and edges. A huge growth of information production has been observed in the last few years. IBM estimates that 2.5 quintillion bytes of data are being generated every day. This makes it easy to conclude that 90% of today's data in the world has been generated in the last 2 years [49]. To meet today's huge graph data processing demands, efficient graph processing systems are required.

A distributed system is one of the best solutions to allocate enormous amounts of graph data before processing real-world applications. Graph partitioning is the



(A) Before Partition a Graph

(B) After Partition a Graph

FIGURE 1.1: Graph Partition

technique which distributes numerous graph data between distributed machines. It achieves good performance by allocating computational workloads among the machines and creates communication channels between them. Graph partitioning algorithms distribute the nodes to among distributed machines with the aim of minimising graph node communication and minimising edge-cut between partitions. Another aim of good partitioning of huge graph-structured data is to allocate equally the graph vertices between the machines; this is called balanced partitioning. One of the most popular balanced partitioning is called k-way partitioning [9] which is an NP hard problem which means that the best partitioning solution is difficult but an optimal solution is possible. Much research has been undertaken in order to compute large-scale graph data,the most popular being Google's Pregel [72] which aims to process large-scale graph data by using a message passing technique from one node to another. It uses the Bulk Synchronous Processing (BSP) system to pass the message between nodes and process them in parallel. In Figure 1.1 the before and after partitioning a graph is depicted.

In streaming graph partitioning, the algorithm receives graph input in a streaming manner and the algorithm immediately decides the respective partition. This is also called a one-pass algorithm as each vertex appears once as an input. The basic problem of this one-pass streaming algorithm is that the current input does not have any idea what is coming up next. It does affect the entire algorithmic process. Another challenge in streaming partitioning is the arrived data has little or no information of itself. This kind of partitioning algorithm has the chance to read vertices and edges only once and it is processed as each vertex arrives. In contrast, typical graph partitioning algorithm loads the entire graph into the machine before sending it to the respective partition. This makes streaming partitioning faster than traditional partitioning and creates an efficient system for post-partitioning computation. However, since the vertices comes one by one it contains limited information of each vertex, which degrades the quality of partition. In Figure 1.2 shows the flow of streaming partitioning technique

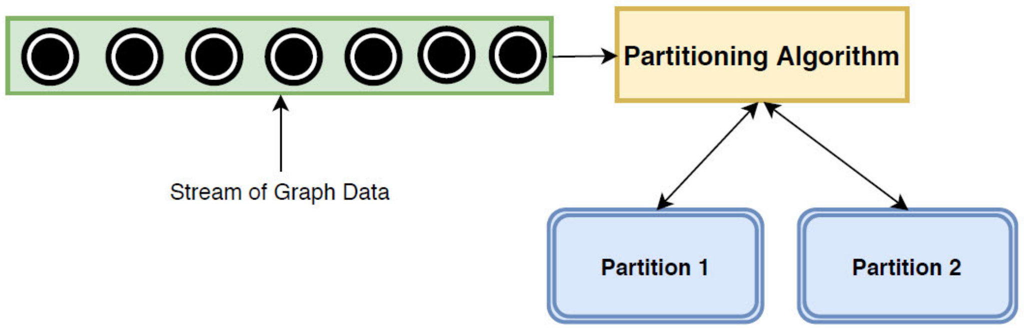


FIGURE 1.2: Streaming Graph Partitioning

In many real-world scenarios, graph data are evolving and analytics happen continuously as graph structure changes dynamically over time. For example, new accounts are created and deleted every day in online services such as Facebook, Skype and Twitter. There are a few other applications in the real-world dealing with dynamic graphs. They are social networks, communication networks, weather forecasts, biological networks and web data in which the edges of underlying graphs are received and updated sequentially. Vertices and edges of a dynamic graph are continuously being added and removed from the partitioned graph. Consequently, it is difficult to manage and locate newly added vertices and edges to a proper partition, while maintaining the balance of the workload and reducing the vertex-cuts/edge-cuts between partitions.

A cloud distributed system has much more flexibility than any other shared nothing cluster system. Given the success of distributed computing, cloud storage has become one of the popular distributed systems due to its cost effectiveness, quick deployment facility, and easy access information. Single machine storage approaches do not scale due to limited capacity. Due to the ever-increasing size of the graph, deployments of applications are moving from small-scale cluster server towards the cloud which provides massive storage and significant parallelism. A cloud consists of tens of thousands of inter-connected machines, which provides much more flexibility in deploying graph data. On top of that, cloud computing has tremendous data recovery options if data is lost.

1.2 Limitation of Existing Partitioning Technique

Graph partitioning is one of the classic graph theory problems in connection with some well-known graph-oriented applications, such as social networks, web crawling, ranking a web page and so on. The two main goals of a graph partitioning algorithm are: 1) Minimising the inter-partitioning communication and 2) Balancing the load as much as possible between partitions. With this aim of partitioning, time efficiency and memory bottlenecks play a role when an ever-increasingly large graph is required for partitioning. Streaming graph partitioning was introduced in order to overcome the memory bottleneck and address time efficiency issues.

A number of streaming graph partitioning algorithms was proposed to address the memory bottleneck issue. However, there are still some limitations to overcome in order to improve the performance of real-world graph applications. For example, most of the proposed study did not consider real-world graph datasets to evaluate their partitioning algorithms. Moreover, no streaming graph partitioning algorithms were evaluated in the cloud environment as cloud computing is one of the most flexible distributed computing environments for the recent real-world applications.

In a dynamic graph application, vertices and edges are frequently being removed or added over time from the partitioned graph. It is difficult to keep communication to a minimum when the graph changes in real-time during partitioning. Consequently, it is challenging to maintain performance in graph partitioning in a streaming manner, while graph is being updated. There has been study undertaken to partition a dynamic graph; however, no study has considered the partitioning technique in a streaming manner to partition a dynamic graph.

1.3 Problem Statement and Objectives

This dissertation focuses on the following problem: *How to develop an efficient dynamic graph partitioning technique in a streaming manner in order to minimise inter-machine communications, and to reduce the load imbalance in a scalable cloud environment?*

Based on the limitations and research problems, our aim is to improve the efficiency of partitioning dynamic graphs by designing a cloud based distributed streaming algorithm which will improve scalability in a big graph dataset. We have the following objectives to achieve in this dissertation:

- To design an optimised distributed dynamic graph partitioning in a steaming manner that will minimise the edge cut/vertex cut between partitions.
- To develop a load balancing distributed algorithm by maintaining good vertex locality.
- To develop a scalable dynamic graph partitioning method that optimises the cloud resources and cost.

1.4 Research Questions

In this section, we discuss the research question we address in this dissertation:

Research Question 1: Allocating and utilising the cloud resources in the distributed environment is a fundamental problem. In streaming graph partitioning, it is necessary to determine the number of allocated machines to distribute the graph, in order to utilise the cloud resources and meet the demands of the upcoming stream of data. How can we properly utilise the cloud resources and optimise resource costs in streaming graph partitioning?

Research Question 2: Minimising edge-cut/vertex-cut is the main concern in a good partitioning algorithm. How can we minimise the edge-cut/vertex-cut to maintain the least communication in a distributed system for static graph data? For example, an application with a huge graph dataset requires dividing up the computational load in a distributed environment when it receives streams of data, over time, to handle. Thus, partitioning is one of the techniques to distribute the workload between machines. A streaming graph partitioning method takes a vertex as an input and the vertex arrives with its associate edges. The graph partitioner will decide immediately the respective partition of a vertex as it arrives. A vertex definitely has a connection to other vertices. This vertex might reside in another partition, this is called external connection of a vertex. Partitioning has to be done in such a way that inter-partition communication could be reduced substantially and outperform the previous work.

Streaming partitioning is partially behaves as a partitioning of dynamic graph. To answer this research question, we use static graph to understand the behaviour of streaming partitioning with static graphs before we use the dynamic graph.

Research Question 3: A dynamic graph changes over time. The vertex of a graph might get a new or lose connection over time. How can we provide good partitions in a streaming manner by maintaining good vertex locality, in order to minimise the communication between machines, when vertices are coming in and going off continuously over time? A time evolving graph changes over time. A best example is a Twitter post; adding a new Twitter post or deleting a post by a user in the social network creates new connections and changes the graph structure. How can we decide a partition to send this newly added connection (vertex and edge) taking into account the need to minimise the communication? An efficient partitioning technique is required to handle this dynamic behaviour of a graph.

1.5 Proposed Solution

In streaming graph partitioning, the data comes in a stream manner one by one. Based on the arrival rate of the upcoming stream of data, it is necessary to determine the number of machines required for a distributed graph processing system. We use a $M/M/\infty$ queuing theory model to select the number of machines before starting partitioning. This is one of several well-studied solutions to select the number of machines in advance based on data arrival rate. Additionally, we use a threshold based auto-scaling technique to select the type of machine in order to optimise the cost of resources. The auto-scaling machine decides the type of machines (Small or Medium) based on the machine capacity and computational load.

Some research has been undertaken in order to address this streaming graph partitioning issue. However, we suggest that a novel streaming window-based graph partitioning in the cloud environment has not been yet studied. Creating a stream window helps to improve partitioning quality, as the window keeps more than one vertex and provides more information of a vertex than a single vertex.

Receiving the graph input in the one-pass manner and partitioning a dynamic graph at the same time is a new kind of partitioning technique which we studied in this dissertation. A dynamic graph is continuously being updated (adding and deleting vertices) from its partitioned graphs. We propose a partitioning algorithm that assigns the vertices and edges based on the current graph partitioning information which was stored in the master machine. The algorithm uses the information to decide the proper allocation. In deciding the proper allocation a vertex assigning method was employed to achieve a good quality of partition. A communication-aware balancing strategy is used to minimise the computational load between partitions. This balancing strategy has taken a number of communications into account to decide the imbalance between partitions.

1.6 Contributions of this Dissertation

This study contributes towards the problem of dynamic partitioning in a streaming manner in a cloud environment. This study also considers the scaling and cost optimisation of cloud resources dynamically. The contribution of this dissertation are as follows:

1.6.1 Literature Studies

This dissertation provides a comprehensive background and literature study of graph partitioning algorithms and frameworks. The contents focus on graph-oriented applications and their uses, graph data structure and their representation, well-known graph partitioning algorithm, streaming partitioning and dynamic partitioning algorithms.

1.6.2 Auto-scaling method in streaming graph partitioning

This dissertation designed an auto-scaling method to optimise the cost and resources for the graph partitioning problem in the cloud. The study considers the one-pass graph partitioning technique to evaluate the auto-scaling performance in utilising the machine and cost.

1.6.3 Streaming graph partitioning algorithm

A graph partitioning algorithm is proposed in a streaming manner which is known as a one-pass manner algorithm. The proposed method explores the sliding streaming window technology with a partitioning technique and a load balancing strategy.

1.6.4 Dynamic Graph Partitioning

This dissertation also provides a scalable dynamic graph partitioning algorithm in a streaming manner. The proposed study contributed the following things: a vertex allocation technique, a communication-aware load balancing technique, and a scaling algorithm to scale in and out of cloud machines, as per the demands of the computational load.

1.7 Thesis Outline

Figure 1.3, shows the outline of the rest of this dissertation. The organisation of the remaining chapters is as follows:

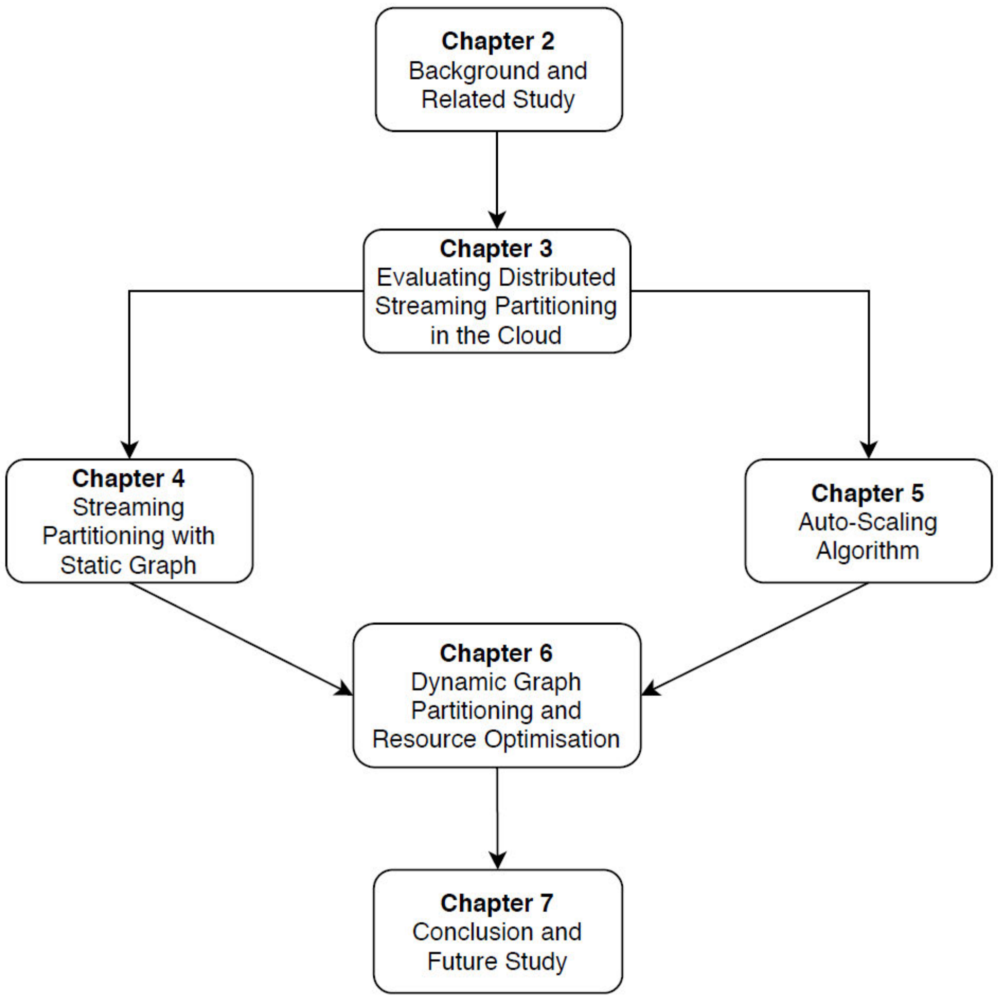

Chapter 2
Background and Related Study
Chapter 3
Evaluating Distributed Streaming Partitioning in the Cloud
Chapter 4
Streaming Partitioning with Static Graph
Chapter 5
Auto-Scaling Algorithm
Chapter 6
Dynamic Graph Partitioning and Resource Optimisation
Chapter 7
Conclusion and Future Study

FIGURE 1.3: Thesis Outline

Chapter 2 represents the background of graph partitioning, streaming partitioning and dynamic graph partitioning with a different type of dataset. This chapter also discusses some well-studied graph partitioning frameworks in the real-world. Extensive related studies of those are also discussed in this chapter. After reviewing the literature studies, at the end of the Chapter 2, a gap analysis is provided.

Chapter 3 provides an experimental study of existing streaming graph partitioning algorithms in the cloud environment. This study finds the limitation of streaming graph partitioning in the cloud environment.

Chapter 4 studies the auto-scaling of cloud resources and cost optimisation in respect to dynamic graph partitioning in a streaming manner.

Chapter 5 explores the streaming graph algorithm technique with the static graph dataset. This chapter proposes a novel streaming partitioning algorithm which is window-based alongside load balancing technique.

- Chapter 5 derived from the following publication:

  - **Md Anwarul Kaium Patwary**, Saurabh Garg, & Byeong Kang, Window-based Streaming Graph Partitioning Algorithm, *Proceedings of the Australasian Computer Science Week Multiconference, ACSW-2019*, Sydney, Australia.

Chapter 6 contributes a graph partitioning algorithm for the dynamic dataset in a streaming manner. This chapter also proposes a machine provision/de-provision algorithm to scale the number of machines to use in the cloud environment.

Chapter 7 concludes the whole study in this dissertation and indicates some future directions for research in relation to dynamic graph partitioning.

Chapter 2

Background and Literature Review

This chapter discusses the exclusive backgrounds of graph partitioning algorithms. Some well-known graph processing frameworks and their performance in relation to graph partitioning algorithms are also discussed. A thorough literature review in dynamic graph partitioning in a streaming manner is completed. We discuss the key performance criteria and drawbacks in streaming graph partitioning with the dynamic graph datasets. We also summarise the key findings of streaming graph partitioning, dynamic partitioning algorithms and graph processing frameworks in several tables. A gap analysis is discussed at the end of this chapter.

2.1 Graph Partitioning Background

In this section, we discuss the background of graph, graph partitioning and related applications of graph partitioning in detail.

Graph partitioning has a long and rich history. It has been studied for many years and these studies have uncovered many problems with many proposed solutions. Large-scale graph-structured data is necessary to partition into several machines in a cluster or several cloud machines, in order to attain efficient processing systems. The amount of graph-structured data has been growing exponentially, at an unprecedented pace. Consequently, graph related applications become sophisticated to manage, represent and interpret. The partitioning technique has a large impact on the performance of graph computation. Good partitioning algorithms always improve performance to some extent (for example, reducing inter-partition communication, providing good locality). The cost of communication between vertices depends on the number of edges between different machines. Partitioning technique plays an important role since it determines the communication cost and balances the workload between computer nodes. Good partitioning algorithms also aim to balance the load between distributed machines. Making graph partitioning balanced while minimising the number of edges between machines is highly important in the production of an efficient distributed system. Graph partitioning is an NP-hard problem [35]; it is really difficult to make the best solution for it. However, it is possible to make an optimal partitioning solution. Despite this limitation, a good number of high- quality graph partitioning algorithms has been developed in the past decade.

2.1.1 Basic Definitions

In this subsection, we discuss the basic notation, definitions, structure, and representation of a graph.

Graph Representation: Let $G = (V, E)$ be a graph, where $V = v_1, v_2, v_3, ...v_n$ is the set of vertices and $E \subset VxV$ is a set of edges of G. A pair $(v, w) \in E$ is called an edge from v to w. A graph can be directed or undirected of its kind. A typical undirected graph is shown in Figure 2.1.

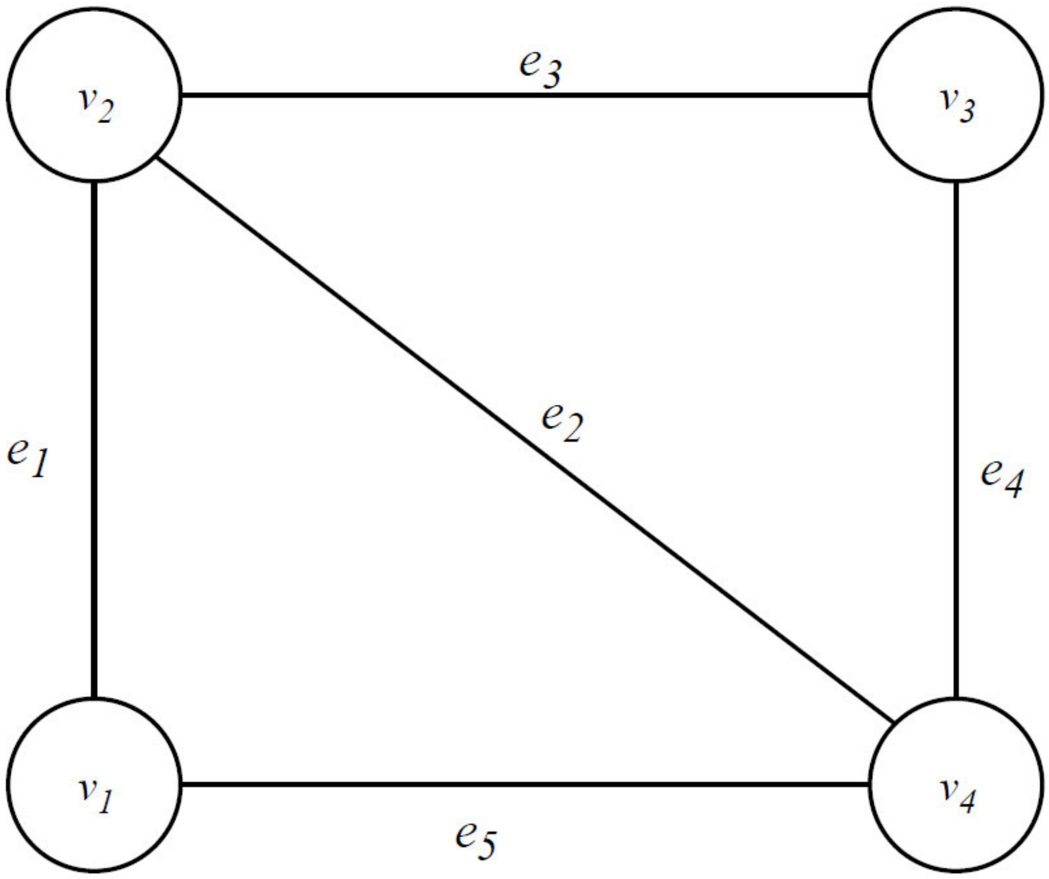


FIGURE 2.1: An Undirected Graph

The two most popular ways to represent a graph in a computer are as follows:

- **Adjacency Matrix** An adjacency matrix is defined as follows: A graph $G = (V, E)$ is represented by a $|V| \times |V|$ boolean matrix A, in which

  $$a_i j = \begin{cases} 1 if (i, j) \in E \\ 0 if (i, j) \in E \end{cases}$$

  The storage requirement of this graph representation is $O(n^2)$.

- **Adjacency List** A graph $G = (V, E)$ is representation by n linear lists. The $i - th$ list contains all nodes j with $(i, j) \in E$. Storage of this representation is $O(n + e)$. Adjacency list representation requires a lower storage requirement than adjacency matrix

Objective Function: We consider an undirected graph G, with a set of edges E and vertices V, such that $G = (V, E)$. A balanced k-way partitioning divides the graph into almost equal subsets. The graph partitioning algorithm uses a balancing constraint to keep all the partitions balanced. The balancing constraint can be defined by Equation 2.1:

$$\forall_i \in \{1..k\} : |V_i| \leq L_{max} := (1+\alpha)\lceil |V|/k \rceil \tag{2.1}$$

where, α is the imbalanced parameter and is a non-negative real number. The vertex v is adjacent to vertex u given there is an edge $\{u, v\} \in E$. If vertex v and vertex u reside in different partitions, this is called the *cut edge*. Thus, $E_{ij} := \{\{u, v\} \in E : u \in V_i, v \in V_j\}$ is the set of edge-cuts between partitions. Edge-cut graph partitioning always aims to reduce this cut.

2.1.2 Performance Matrices

A standard graph partitioning performance can be measured with the following matrices:

Computation time: The amount of time it captures while a graph-partitioning algorithm is running to partition a graph. More computation time is required if we have more vertices to traverse. Consequently, the large-scale graph does require more computation time to loop through all the vertices. Many graph partitioning algorithms were developed to reduce computation time.

Communication cost: Sending and receiving messages between computer nodes is a fundamental feature in a distributed system after partitioning a big graph. The number of the messages has an impact on a system's performance, as well as partitioning performance. The number of communications is dependent on the number edge-cuts occurring in a vertex partitioning. Good partitioning algorithms aim to reduce the sum of the edge-cut, which can be defined by

$$\sum_{i=1}^{k} |e_i| \tag{2.2}$$

where, e_i is the edge of a partition connected to other disjoint partition and k is the total number of partitions.

2.1.3 Dynamic Graphs

The graph changes over time in any application and are known as dynamic graphs. In this section, we discuss the notation of dynamic graphs and their behaviour in detail, for example how they are updated over time and how graph transition occurs. A dynamic graph can be categorised in two ways according to the type of changes happening in a graph [25]:

1) Fully Dynamic: A graph is said to be a fully dynamic graph in an application if insertions and deletions of vertices or edges are allowed.

2) Partially Dynamic: A graph is called a partially dynamic graph in a graph application if either deletions or insertions occur.

2.1.4 Edge-cut Partitioning

Edge-cut partitioning divides the graphs by cutting connections between vertices and distributing the vertices to a different partition. The objective is to minimise the number of edges across different partitions, while maintaining the balance of the vertices. This can be defined as follows:

$$\min_{A} |\{e|e = (v_i, v_j) \in E, v_i \in V_x, v_j \in V_y, x \neq y\} \tag{2.3}$$

2.1.5 Vertex-cut Partitioning

Partitioning sale-free graphs is difficult if using edge-cut partitioning. Vertex-cut is an alternative and is a better solution for scale-free graphs. Vertex-cuts divide graphs by vertices and allocate the edges to the distributed machine. In vertex-cuts, the number of edges on each machine is used to estimate the computation cost of that machine, and the number of the replication of the vertices is used to estimate the communication cost. Let each from graph dataset $e \in E$ be assigned to a partition $P(e) \in \{1, ..., p\}$. Then each vertex spans a set of different partitions $A(v) \subset \{1, ..., p\}$. Consequently, $|A(v)|$ is the number of replications of v among different machines. The balanced vertex-cut partitioning can be defined by,

$$\min_{A} \frac{1}{|V|} \sum_{v \in V} |A(v)| \tag{2.4}$$

Each of these has different advantages and disadvantages in different perspectives. For example, a study[126]shows that vertex-cut partitioning is better than edge-cut partitioning because the natural graph's degree distribution is highly skewed, thus vertex-cut partitioning performs better in such a case.

2.1.6 Applications of Graph Partitioning

There are several real applications available to generate massive datasets and those datasets can be represented as graph-structured data, for example, a website in which a page is considered as a vertex and links to another page as an edge. Consequently, graphs have become key components of a wide range of applications, such as PageRank, connected component, protein interaction, semi-supervised learning based on random graphs walks, web search based on link analysis, scene reconstruction based on Markov random fields, and social community detection based on label propagation to name just a few examples.

PageRank: PageRank [12] is a well-known algorithm to rank a webpage and find a most impacted webpage. Massive web pages consider graph-structured data in which edges link one page to another and vertices as a webpage. A time efficient PageRank calculation is possible by partitioning this huge graph into several machines. This process distributes the computational load between machines. A good partitioning algorithm reduces communication between machines. This application calculates the linking relationship between web pages. The Web can be represented as a giant graph, in which the web pages are nodes, and links from one page to another are edges. For example, if many people follow a Twitter user, the user will be ranked highly. Despite the fact that the PageRank is very effective in calculating the rank of a page, it is really computationally intensive for several reasons [17]: i) The size of web graph data is huge; it is learnt that a web graph contains 1 trillion web pages. ii) Secondly, the dynamism of web graphs: new web pages are always being added and existing pages are always being deleted from a website and web graph data. Thus, it does require repeated computation to keep updating the rank of a page. An efficient algorithm is required to control this dynamicity. iii) Thirdly, sometimes it is necessary to compute more than one PageRank vector. This happens when there is more than one preference view for a page's importance [17].

Social Networks: Partitioning a dynamic graph efficiently in a streaming manner is the primary objective of this research. The dynamic structure of a graph network is present in most recent applications for example, social networks (such as, Facebook and Twitter) that represent a large portion of the data on the internet technology today. Social media is one of the largest and most important graph networks which provides dynamic behaviour of a graph network. Twitter recorded 13,000 Tweets per second [49]. It is also learned that there are 400 million Tweets per day on average.

Shortest Path: Transportation networks and finding the shortest path from a big map are one of the common applications in graph computing. The shortest path algorithm calculates the shortest path between two destinations when there are two or more ways to reach a destination from u to v, for example, the GPS system which people use to find the shortest way to drive from one destination to another. Airline carriers uses a map route which can naturally be formed as a graph; the vertices are the airports and there is an edge from u to v, if there is a flight from location u to location v. This path could be as directed (u, v) or undirected $(u, v)(v, u)$. Observing such networks, we notice that there is small number of vertices (location), with an enormous number of incident edges (connection between locations). In a similar manner, another transportation network also can be formed. For example, rail networks, having each terminal as vertices and an edge is a route from one terminal to another.

Data Clustering: Graph partitioning is also applicable in data clustering [51]. Researchers use the graph cut to cluster microarray data in a bioinformatics application [46]. They group huge biological graph data based on their similar gene activity. In computer vision, the application computer scientist also uses graph partitioning in image segmentation.

Circuit Design: A practical example of this graph partitioning problem is circuit board design [100]. How can a component of an electronic circuit be placed onto a circuit board while minimising the number of wires or connections between cards? A very large-scale integration (VLSI) system is one of the graph partitioning issues that arises in order reduce the connection between circuits in designing VLSI. The main objective of this partitioning is to reduce the VLSI design complexity by splitting circuit component into smaller components. Another goal of good partitioning is to reduce the number of connections among those circuit components. Here, vertices are the cells and the edges are the wires between them.

Image Processing: Image segmentation is a most fundamental problem in image processing of any applications. Graph partitioning is one of the most attractive tools to split into several components of a picture. Pixels are denoted as a vertex and if there are similarities between pixels, they are represented as an edge.

Connected Components: The connected component algorithm [41] labels each connected component of the graph with the ID of its lowest-numbered vertex. For example, in social networks, connected components can approximate clusters.

Parallel Computing: Another important use of graph partitioning is in parallel computing [37]. Partitioning helps to divide the computational load equally to parallel machines, in order to achieve faster computation and better performance.

Query Processing: Graph partitioning techniques play a pivotal role in answering a query and processing a query of an application. Graph partitioning distributes the workload of these applications equally between machines and reduces the communication between machines. Finding good partitions of a dynamically changing graph is a challenging task and an NP-hard problem, because it does change over time when adding and deleting connections of graph data. A new post on social media from a user creates a new connection, and interacting with a post generates a new connection in a graph.

2.2 Graph Partitioning Algorithms

In this section, we discuss some well-established graph partitioning algorithms and analyse their limitations.

2.2.1 Kernighan-Lin Algorithm

Kernighan and Lin proposed a ground-breaking partitioning algorithm [62], which is one of the oldest graph partitioning algorithms. This algorithm aims to reduce the edge cut between two partitions by swapping vertices. It improves an initial partitioning by finding optimal subsets $A \subset P_1, B \subset p_2$. A major disadvantage of this algorithm is the huge running time. In the worst case, the running time for one iteration of the Kernighan-Lin algorithm is $O(n^2 logn)$. The algorithm works by exchanging vertices between machines or blocks. It repeatedly finds such set A, to B until it reaches the optimum value. In Algorithm 1 shows the complete pseudocode of Kernighan-Lin algorithm.

Algorithm 1 Kernighan-Lin Algorithm

```
Divide the graph into two parts $A$ and $B$ of equal size arbitrarily
Repeat until no more vertices are left:
Select $a_i \in A$, $b_i \in B$, such that the reduction in cost as large as possible and neither $a_i, b_i$ has been chosen before.
Swap $a_i$ and $b_i$
Let $C_i$ be the cost of the partition after swapping $a_i$, $b_i$
Return($A', B'$) corresponding to the smallest $C_i$ observed.
```

2.2.2 Fiduccia and Mattheyes Algorithm

Some improvements have been made to the Kernighan-Lin Algorithm by Fiduccia and Mattheyes [30]. The most significant improvement was in the reduction of running time. The main idea was that a vertex would move to another partition almost immediately.

2.2.3 Multilevel Algorithm

This algorithm uses multi-level approaches by cutting a big graph into several subgraphs based on edge matching criteria before partitioning. Multilevel graph partitioning is based on edge coarsening and local search algorithm. It was initially proposed by [62] to speed up spectral partitioning. Because, spectral partitioning require huge computation of the eigenvector. Later, much improvement has been made and it was declared as a multilevel algorithm by [13] as it is known today. This approach has three phases to complete. They are:

Coarsening: In this phase, a coarse graph is constructed by matching edges by using a matching algorithm. Matching is the most widely used method for coarsening a large graph. In the partitioning algorithm, KaPPa [47] uses a two-phase matching method to address two issues: i) a rating function and ii) matching algorithm. Based on local information, a rating function calculates the possibility of contracting an edge. A matching algorithm does maximise the sum of the rating of the contracted edges. In this, process the newly constructed subgraph uses the input graph to form a related, coarser graph, until a sufficiently small graph is obtained.

Initial partitioning: If the graph is sufficiently small enough to partition, then the coarsening process will stop. Any initial graph algorithm, such as spectral partitioning, or graph growing does the partitioning of these coarsened subgraphs.

Un-coarsening: Un-coarsening is also called the local improvement method. To uncontract (un-coarsen) matching edges is the task of this phase. There are two popular methods in this phase: i) max-flow min-cut computation between pairs of blocks. ii) Multi-try Fiduccia and Mattheyes' method [30]. It is a $k-$way local search algorithm which obtained highly, localised performance.

George Karypis proposed a fast multi-level graph partitioning algorithm [58] for irregular graphs. The proposed algorithm introduced an heuristic for the coarsening part called heavy-edge heuristic in a multi-level partitioning scheme. The heuristic generates a coarse graph based on a small factor from the size of the partition obtained by multi-level refinement. Another multilevel algorithm was proposed by Karpis [59] with the focus to minimise the time complexity for $k-$way partitioning.

METIS [62] was proposed to do graph partitioning based on the multi-level scheme. METIS is also considered as a de facto standard for near-optimal partitioning in distributed graph partitioning and is one of best performing offline graph partitioning algorithms. METIS can reduce the communication cost between distributed machine, despite having lengthy processing time for small graphs. However, for medium or large graph datasets, METIS is not suitable. It also is not suitable for online graph partitioning and so, to overcome this drawback few online graph partitioning were proposed [29] over METIS algorithm by utilising the partitioning scheme from METIS.

2.2.4 Graph Partitioning for static graph

In this section, we will discuss the related work on graph partitioning for static data and some facts on static graph partitioning.

Three factors affect a static graph partitioning. They are: 1) The graph algorithms' execution while performing the partition. Algorithms have a significant impact on the performance of partitioning; some algorithms create longer runtime but the vertices of a graph distribution are better in this perspective. 2) Secondly, the structure of a graph. Different graphs produce different results in partitioning, for example, social network graphs, biological network graphs, web graphs; each of these have different structures and properties. This has a huge impact on partitioning 3) Lastly, worker tasks across distributed computing nodes. Some fast workers have to wait for the slow workers to complete a job. In this case, the asynchronous system does the best to counteract this issue, in that the processing continues without waiting for another worker [93].

Two main objectives of graph partitioning are as follows:

- To minimise of communication between processors/machines in distributed system. This is the main objective of graph partitioning. The objective function of graph partitioning is as follows:

  $$\sum_{i<j} w(E_{ij}) \tag{2.5}$$

  The main goal of this function is to compute the sum of the weight of the cut edges.

- To allocate the computational load equally between the machines.

The following related works aim to achieve these objectives by proposing different techniques and methods.

Label propagation method: A parallel graph partitioning proposal [76] based on the *label propagation* [85] technique which is developed for graph clustering. This parallel label propagation uses a coarse-grained distributed memory evolution algorithm to compute a high quality partitioning. A parallel graph data structure has been used in this technique to perform the partitioning. In a parallel graph data structure, each processing element receives a sub-graph. Each node in a sub-graph has its own ID from the interval $I := a..b$. Each sub-graph also contains edges, associated with the nodes, as well as the end points of an edge which are not in the interval I. Each PE performs the algorithms on its part of the graph in parallelising the label propagation algorithm.

Based on the label propagation algorithm, another balanced partitioning technique was proposed [114] while edge locality is maximised by using a greedy algorithm in which the number of edges are assigned to the same partition. In order to ensure that all the partitions are balanced, a partitioning constraint is applied, using lower bounds and upper bounds. This constraint makes sure that all partitions are well-balanced during the allocation of vertices.

Multi-level propagation algorithms have been used widely to propose multi-level graph partitioning. Another multilevel algorithm was proposed [5] to partition power-law graphs. This algorithm aimed to partition high performance computing applications whose computation involves a power-law distribution curve of degree distribution of graphs.

Degree based technique: A novel vertex-cut method, called degree-based hashing (DBH) [127] proposes the partitioning and distribution of a power-law graph in a distributed system. This approach aims to minimise the communication cost, at the same time providing a well-balanced partitioning system. Vertex-cut partitioning is well suited for power-law distribution natural graphs, and this has been exploited here in aiming to achieve less communication cost and better balance.

Balanced Partitioning: One of the main objectives of graph partitioning is to allocate vertices equally among the distributed machines. Each partition size should be as close as possible to the average of a partition $|V|/k$, where k is number of partitions. This is a classic NP-hard problem in distributed computing [35]. The input of this problem is an undirected graph $G(V, E)$ and an integer $k \in Z^+$; the output of this partitioning is the set of vertices. Partitioning occurs in such a way that the number of edges across the partitions is reduced. Balance constraint is defined by an unbalanced parameter. To define the balance constraint an unbalanced parameter, v is used. A partitioning algorithm will divide the set of vertices in such a way that each of the partition's size are at most vn/k, where n is number vertices in a G.

2.2.5 Distributed Graph Partitioner

Distributed partitioning plays a key role in processing large-scale graph applications. A $k-$way distributed algorithm was proposed in [66], by using a combinatorial optimisation technique called Simulated Annealing [64] and Terminal Propagation [26]. The proposed method addresses the issues which arise in a typical $k-$way partitioning. In a recursive bisection process, lack of global graph information produces degradation in the partitioning result.

In order to provide a scalable communication a distributed evolutionary algorithm was proposed [94] called KaFFPaE, based on the KaFFPa platform. KaFFPaE work was based on crossover and mutation operators to minimise the time complexity for partitioning a large graph.

A distributed graph partitioner called Sheep was proposed in [73]. Before partitioning the graph, Sheep converts the input graph into a tree by using a Map-Reduce function. The proposed method partitions the edges to the distributed machine in a significantly less time than the state-of-the-art METIS algorithm.

Based on FENNEL [113], a distributed FENNEL was proposed called AsyncFENNEL [105]. The proposed method aims to improve efficiency and scalability by using a tree-shaped map-reduce network.

A query-aware partitioning algorithm, TAPER, [34] was proposed for a heterogeneous graph in a distributed environment. TAPER aims to reduce the inter-partition migration of a vertex when processing a query by a user in a partitioned graph.

Edge-based partitioning is another type of partitioning in addition to vertex-based partitioning. A distributed edge partitioning for a large-scale graph has been proposed [42] that distributes the edge rather than distributing the vertex in vertex partitioning. This overcomes one of the big disadvantages of vertex partitioning: unbalanced edges in all the partitions. By exploiting local search and simulated annealing a distributed graph partitioning was proposed in [87] called Ja-BE-Ja. The proposed algorithm supports both edge-cut and vertex-cut partitioning. Ja-BE-Ja performed better in reducing edge-cut/vertex-cut and outperformed the classic partitioning algorithm METIS.

2.2.6 Parallel Processing

Parallelism in graph partitioning is mandatory in achieving high scalability in a large-graph application. A parallel partitioning heuristic [95] was studied to produce a balanced partitioning. It allows swapping of a large number of elements between two sets of the vertex. It chooses randomly the vertices to swap to make size of the partitions equal. The parallel multi-level graph partitioning technique [60] is used to minimise the communication between processors. It achieves less communication than the one-dimensional distribution for a graph with a relatively high degree. For irregular graph partitioning, multi-level $k-$way partitioning was proposed [57] in a parallel manner that extended the parallel implementation of the multi-level $k-$way algorithm [59].

Microsoft research proposed a partitioning system [122] that deal with billions of nodes. It aims to address the issues in load balancing and communication overhead. The authors use multi-level label propagation algorithm for partitioning a large graph. Another label propagation-based, parallel implementation proposed in [108] is called xtraPULP. It uses a scalable scheme to minimise the computation time of partitioning. The algorithm is able to partition a large-scale graph in a minute.

A clustering algorithm was used to partition a graph; the clustering technique uses the coarsening scheme that finds the vertices that are highly connected to each other. A genetic algorithm-based graph partitioning has been proposed in [14]. This algorithm has a schema processing feature that improves the capability of space searching of a genetic algorithm. Thus, it performs better in the partitioning of the graph by using the genetic algorithm.

For the complex network, a parallel graph partitioning algorithm was proposed in [76]. The authors uses the label propagation and parallel evolutionary algorithm to generate a better coarse graph in the refinement process of the multi-level scheme of graph partitioning. This study achieves more scalability and high-quality partitioning.

ParMetis [96] is the extended study and parallel version of METIS partitioner. The proposed method uses a repartitioning scheme to adapt to change in a partitioned meshes. It minimises the data migration from one partition to another, in order to reduces the edge-cut.

2.2.7 Hypergraph Partitioning

A graph in which each edge is linked to more than two vertices is called a hypergraph [16]. Many real-world applications are related to hypergraph partitioning, for example, VLSI design, Boolean satisfiability, numerical linear algebra and scientific computing. A formal definition of a hypergraph is as follows: A hypergraph $H = (V, N)$, where V is a set of vertices and N is a set of hyperedges also known as nets. Hypergraph related tools are powerful for representing complex and non-pairwise relationship.

In hypergraph partitioning, given a graph H, the partitioning scheme assigns vertices of H to the number of disjoint partitions.

Henne et al proposed a hypergraph partitioning algorithm [45] by exploiting label propagation local search algorithms, with the aim reducing the computational complexity. Most of the standard graph processing framework converts the hypergraph into typical graphs to process them. A distributed scalable hypergraph processing framework called HyperX was proposed[55] to avoid the graph conversion task. The proposed framework minimises significant computational time compared with the previous algorithm.

A multilevel spectral partitioning was proposed for the hypergraph in [134]. The authors proposed the partitioning technique based on the arbitrary vertex size.

2.3 Graph Partitioning Frameworks

Data is increasing exponentially in different domains; for example, a sensor network, as people are trending to rely on more sensor devices to meet different challenges. Web data has increased tremendously in the last couple of years, in several domains, such as information systems to manage and run business, writing blogs, news sites, social network sites and personal websites. This has produced a demand for an efficient system to manage this data but with less computational time. Researchers have been working to meet today's demand in handling the sheer amount of data and analysing and presenting them efficiently. Various graph-structured data partitioning frameworks have been proposed in order to process an enormous amount of data from various applications.

In the following subsection, we will discuss several graph-partitioning frameworks, which have been proposed. Each framework uses different techniques and different kind of graph.

2.3.1 Vertex-centric Programming

The vertex-centric programming model is a distributed processing of large-scale graphs by sending a message from one vertex to another. Traditional big data processing tools (for example, MapReduce) are not well-suited to an iterative process. Consequently, it is not well- supported for large-scale graph processing. The proposed vertex-centric programming model is the solution to overcome this issue. This approach improves locality, is able to process many iterative computing problems in a natural way, and provides greater linear scalability [75].

A well-known and widely used graph processing system is Google's Pregel [72], later on, a few number of graph processing framework were proposed based on Pregel. Pregel is based on the vertex-centric programming model; this model iteratively executes a user- defined program over vertices of the graph. A defined function usually takes input from adjacent vertices or incoming edges, then sends output to outgoing edges. This message passing technique is also known as Bulk Synchronous Programming (BSP) model [116].

Giraph [92] is an open source implementation of Pregel and has been developed based on Pregel. Giraph is currently being used by Facebook to analyse users' connections to one another and their associated connected groups on Facebook. Giraph has high scalability and iterative features to process a large graph-structured dataset. Any graph processing system requires an efficient iterative feature to process large-scale graph applications. Giraph has this efficient iterative processing power in terms of runtime. Despite having effective features in that system, many well-known proposed graph-processing systems are suitable to process large-scale graphs., yet they are unable to support many important data mining and machine-learning algorithms. In order to overcome these issues, a large-scale graph processing system

GraphLab [71] was proposed to facilitating machine learning and a data mining process. This system has vertex-program access to a distributed graph by exploiting the asynchronous distributed share-memory architecture. Vertex-programs are allowed to access relevant information on the current vertex and associated edges of that vertex, and adjacent to that vertex. Thus, it reduces the network latency and network bottleneck and ensures data consistency, achieving a high level of graph-data parallel computation.

GraphX [39] is a distributed data-flow framework that has been developed on top of Apache Spark [131], a popular and widely used distributed data-flow system. GraphX is a graph-processing library that aims to overcome the issue of typical general-purpose data-flow frameworks like MapReduce [24]. General-purpose distributed frameworks are well suited for handling unstructured and tabular data. However, processing graph data that has an enormous iterative process can be challenging in a general-purpose distributed framework. Spark is developed based on Resilient Distributed Dataset [39]. RDDs are a partitioned collection of data, and it is created by data-parallel operators. GraphX used a vertex-centric programming model [75] in order to make communication channela and exchange messages between vertices. A Vertex-centric function iteratively executes a user-defined function and this function takes input from adjacent vertices or incoming edges. Consequently, the resultant output sends a message to incoming vertices with edges information of that vertex. This is how the process goes on until reaching all the vertices.

A 'think like a graph' [111] programming model was proposed based on Apache Giraph. Pregel and Giraph have a limitation that hides partitioning information from the user and this prevents the optimisation of a particular algorithm. Giraph has some limitations in terms of computational time. To overcome the limitations, a few improvements of Giraph model has been undertaken by a new system called Giraph++. This model is called a graph-centric programming paradigm rather than using a vertex-centric programming technique. This technique allows information to flow within a partition.The vertex-centric model requires many computational steps to pass the information from one vertex to another. This graph-centric model reduces the computational steps in terms of passing information to other vertices.

2.3.2 Power-law Graphs

Power-law graph, or scale-free graph, is a kind of graph in which there a few vertex with a high degree (degree is the number of edges of a vertex) and many vertices with low degree [6]. This means that the number of vertices y, of given degree x is proportional to $x(\beta)$ for some constant $\beta \geq 0$. Today's number of large-distributed system, for example, communication, social, and biological networks, demonstrate power-law distribution in their graph structures.

A real world graph, for example, social network, and the web typically have power-law degree distribution. Partitioning power-law graphs are very difficult and it is very hard to distribute them in a distributed system due to it's scale-free distribution of edges. PowerGraph [38] aims to reduce inter-partition communication by computing edges over vertices. It follows the GAS (Gather, Apply and Scatter) model and uses the vertex-cut partitioning technique. This technique distributes vertices into multiple machines in a replica of a single vertex to parallelise the computation. As a result, PowerGraph achieves better parallelism, reduces storage cost and provides effective partitioning. PowerGraph has been implemented in the EC2 environment, using some real-world applications such as PageRank, Greedy Graph Colouring and Single Source Shortest Path. Hama [101] aims to compute high level matrix computations as computational, intensive graph applications which are trending higher.

PowerLyra [20] adapt their computation and partitioning strategies for different vertices. PowerLyra has a partitioning technique called hybrid-cut, which is a combination of both edge-cut and vertex-cut. This hybrid-cut provides efficient performance and outperforms other distributed graph computation systems. This hybrid-cut provides some impressive features to improve performance: i) it gives a much lower replication factor; ii) it provides unidirectional access locality for low degree vertices (This is useful in the hybrid computational model.); iii) it is very efficient in graph accessibility as it exploits hash-based partitioning for both low-degree and high-degree vertices.

A distributed memory cloud system named Trinity [103] was proposed, which aims to provide a storage infrastructure and graph computation framework, as well as in the distributed memory cloud environment. Trinity does support online and offline graph data as well as some real-life applications. Trinity plays an important role in optimising the memory and network communication; consequently, it provides fast processing of graph computation and efficient parallelism. In addition, Trinity provides a specification language for users to declare data schemes and communication protocols. Trinity achieved high throughput graph analytics on large web scales and numerous vertices.

Graph Streaming Partitioner (GraSP) [36] is a distributed graph processing system. GraSP is able to handle a stream of graph data and partition them in real time. GraSP used MPI (Message Passing Interface) to develop the distributed system for communicating among distributed machines. This is the first distributed system which has been developed by MPI in order to handle streaming graph data. GraSP was able to make a significant improvement in scalability with the 1024 machine.

2.3.3 Scalability

Graph computation is one of the key examples of various scientific computations, such as machine learning, information retrieval, bioinformatics and social network analysis. Hamma [101] is a scalable graph processing tool proposed to address computation need of scientific applications. Hama was implemented in the MapReduce environment which is very efficient in large-scale data intensive computation. Hama aims to achieve performance improvement in the following matrices: compatibility, scalability, flexibility and applicability. Hama is capable of performing all the functionalities of Hadoop and its associated packages. Consequently, Hama is able to exploit any large-scale distributed system, such as EC2 without making any modifications. It also has flexibility in computing any pattern of applications and is advantageous in dealing with any scale of applications. Hama is applicable to various range graphs and matrix applications.

GraphBuilder [52] is a framework which aims to create a scalable system that has all the features from graph formation, tabulation, transformation, partitioning, output formatting and serialisation. GraphBuilder has been implemented in the MapReduce environment.

Graph Processing System (GPS) [93] is an open source distributed graph processing system. GPS aimed to support large-scale graphs to achieve high scalability and more fault tolerance. GPS has flexibility in the execution of algorithms on the big-graph dataset. GPS has additional dynamic partitioning feature on top of Pregel [72]. Dynamic partitioning plays a very significant role in dealing with dynamic graphs, when graphs have a tendency to change structure by inserting or deleting vertices

Another scalable graph partitioning framework called Spinner [74], was implemented in the public cloud environment. Spinner is flexible in handling massive dynamic graphs, and provides a better partitioning quality with minimum edge-cut. Spinner is based on well-known Giraph platform and aims to speed up the processing of different applications.

A well-performed single-machine system to compute large-scale graph is GraphChi [67]. This system aimed to provide a large-scale computation system on a single PC in order to overcome the difficulty of designing a distributed system for the non-expert. This system has come up to the attention of industry and academia. GraphChi has been implemented in C++ by making use of secondary storage to store huge graph data to process. GraphChi introduces two new techniques to process large graphs in a single PC. They are i) out-of-core computation and ii) selective scheduling. Out-of-core is a new data structure that minimises the accessing to secondary storage while graph computation is being done.

A light weight parallel graph processing framework called Ligra [106] proposed to make it easy for graph traversal algorithm to process. Ligra is able to process a large-scale graph efficiently with either the DFS or BFS algorithm in a single machine. However, it has a disadvantage; if the graph grows over-time, Liagra will have a memory bottleneck issue. Table 2.1 shows that some well-known graph partitioning framework and their partitioning strategy.

2.4 Streaming Graph Partitioning

In this section, we present an overview of graph stream and its partitioning technique. I also discuss some previous works on streaming graph partitioning.

Framework	Strategy	Type of Balance	Streaming	Type of Graph
Deterministic Greedy[110]	Edge-cut	Vertex Balance	Yes	Static
FENNEL[113]	Edge-cut	Vertex Balance	Yes	Static
PowerGraph[38]	Vertex-cut	Edge Balance	Yes	Static
PowerLyra[20]	Hybrid	Edge Balance	No	Static
GraphBuilder [52]	Vertex-cut	Edge Balance	No	Static
S-PowerGraph[126]	Vertex-cut	Edge Balance	Yes	Static
HDRF[83]	Vertex-cut	Edge Balance	Yes	Static
LEOPARD[48]	Edge-cut	Vertex Balance	Yes	Dynamic
HoVerCut[70]	Vertex-cut	Edge Balance	Yes	Static
DBH[70]	Vertex-cut	Edge Balance	No	Static
GraphChi[67]	Edge-cut	Vertex Balance	No	Static
Spinner[74]	Edge-cut	Vertex Balance	No	Static
GPS[93]	Edge-cut	Vertex Balance	Yes	Dynamic
GraSP[70]	Edge-cut	Vertex Balance	Yes	Static
Ligra[106]	Edge-cut	Vertex Balance	No	Static
Ja-Be-Ja[87]	Both	Both	No	Static
TAPER[34]	Edge-cut	Vertex Balance	No	Static
Sheep[73]	Vertex-cut	Edge Balance	No	Static
Hamma[107]	Both	Both	No	Static
Trinity[103]	Both	Both	No	Static
AsyncFENNEL [105]	Edge-cut	Vertex Balance	Yes	Dynamic
MLP[122]	Edge-cut	Vertex Balance	No	Static
xtraPULP[108]	Edge-cut	Vertex Balance	No	Static

TABLE 2.1: Summarisation of Graph Partitioning Algorithms and Frameworks

Stream processing is a data processing technique that is very convenient for processing low latency, incremental computations on graph-structured data. In this technique, the input graph comes in the form of streams, for example, social network data, bank transactions or weather report data, and so on. Almost any kind of data can be formed as a graph as long as those data have a relationship between them. A single machine is often unable to process emerging large-scale graphs due to memory latency and computational cost. Graph partitioning is the appropriate technique to distribute this large-scale graph data between distributed machines.

Traditional graph partitioning algorithms are not suitable for the following aspects of the "big data" era: i) a massive dataset is too big to store; ii) Even an $O(n^2)$ time algorithm is too slow; iii) recently most of the applications' data changes over time; iv) algorithms should have the capacity to handle the dynamic changes of graph data. Consequently, streaming, dynamic and distributed graph algorithms are a wise solution and much needed for analysing big graph data in today's data explosion trend. The most common example is network analysis; nowadays a website contains huge information and contents which is interrelated. For example, web pages and hyperlinks, neurons and synapses, papers and citations. Maintaining this interrelationship and processing such graph data are critical tasks.Partitioning a massive graph to process different applications without storing them in main memory is the data stream model. In this approach, the partitioning algorithm receives the input graph data in a stream order; the stream could be vertices of a graph and associated edges of that vertex. Data stream models are the most promising and most popular trends in big graph data processing. Stream based graph partitioning algorithms are much more effective than traditional partitioning algorithms.

Significant progress has been made in analysing the streaming data but the progress for analysing graph data is still limited [8]. Streaming graph analysis has gained attention because of recent real-time graph data infrastructure and its usability in different, promising applications. When a graph processing application processes a graph data as it arrives, it is considered as a streaming graph processing application and its datasets are known as streaming graph data. The vertex of a graph arrives along with its associated edges, whereas offline graph processing systems receive the entire graph data and store them in main memory before processing them. This creates a huge memory bottleneck and consumes enormous data storage, if we process large-scale datasets [36]. Streaming graph data is also known as online data that leverage the storage issue. However, this has brought some challenges in handling one pass data. In this process, data appears only once when it arrives; thus, immediately, it needs to be allocated and sent to the proper location in a distributed system. An efficient and optimised streaming graph partitioning algorithm is required to handle such graph data. Streaming graph partitioning is a state-of-art technique, which decentralises the graph processing system. The system has a chance to read vertices and associated edges only once. Consequently, it leads to less overhead of memory access, storage and runtime which can be defined by: $O(|V| + |E|)$ [36].

An experimental comparison study and proposed a streaming graph partitioning algorithm in [43]. The authors address the issues in run time by analysing the run time process. This study also takes into account the characteristics of the graph

Algorithms	**Findings**
Natural Graph Factorization [7]	A novel factorization technique proposed that reduces the vertices rather than edges across the partition. It uses the local information to perform the vertex replication process which reduces the communication cost
METIS[62]	A minimum cost partitioning approach is proposed, which find the optimal partitions. A fast partitioning process is proposed.
LOOM[33]	This streaming partitioning uses sub-graph pattern machine queries, which reduces inter-partitioning communication. A streaming window uses to process the query.
STINGER[88]	This algorithm aims to handle constant stream data from different domain (e.g health care, security, business, and social network). A new graph data structure proposed which provides trade-off with partitioning algorithm. This leads to achieve high performance parallelism of massive graph processing.
Planted Partition[112]	A higher length walk has been introduced which significantly reduces the computational cost
HDRF[83]	A streaming partitioning algorithm, which effectively exploits skewed distribution, graphs and it considers the vertex degree while placing a vertex to a partition. This algorithm replicates the high-degree vertex first in order to make a balanced partitioning system.
HoVerCut[91]	This algorithm is horizontally and vertically scalable. This algorithm can receive multiple streaming sources.

TABLE 2.2: Key Findings of Steaming Graph Partitioning

in order to achieve a better quality of partitioning, depending on graph applications. Another graph analysis model [130] proposed for streaming data to analyse the graph data in real time for various applications.

A framework [31] proposed using Condensed Spanning Tree (CST) structure in order to address the memory bottleneck issue in streaming graph partitioning. It also capable of adapting, based on the demands of the requirements of different graph applications, minimising the inter-machine communication and reducing the load imbalance.

2.4.1 Greedy Algorithm

A well-known streaming partitioning was proposed in order to compute large-scale of streaming data in [110]. This is greedy heuristic, which is linear. This algorithm is also known as Linear Deterministic Greedy (LDG). It has a central graph loader, which loads the data and distributes them among the available workers. This heuristic assigns a vertex to the partition with which it shares the most edges. 16 partitioning heuristics are evaluated with 21 different datasets. Graph datasets have been used from several domains: The World Wide Web, social networks, finite-element meshes, and synthetic datasets. The authors achieved different results from different datasets. This streaming partitioning method makes heuristics scalable in the size and the number of partitions of graphs. It has significantly sped up PageRank computations on Spark [131], by 18% to 39% for large social networks. Another greedy heuristic algorithm proposed in [7], uses an unweighted, deterministic greedy algorithm instead of using a weighted penalty function, in order to partition vertices. This algorithm uses also a factorisation technique that aims to reduce the neighbouring vertices rather than edges across partition. In other words, a vertex-cut partitioning technique is employed here and is well-suited for large-scale natural graph.

Microsoft research implemented a partitioning framework called FENNEL [113] based on greedy heuristics. This is one of the top performing graph-partitioning frameworks. This partitioning framework aimed to overcome the issue of computational complexity using the traditional balanced graph-partitioning technique. FENNEL leverages modularity maximisation [78] to deploy a greedy strategy for maintaining balanced partitions. It has also improved performance regarding communication cost and runtime, while computing graph data iteratively in a distributed system. The FENNEL algorithm can be defined by the following equation:

$$\arg\max_{1 \le i \le k} \{ |N(v) \cap P_i| - \alpha \frac{\gamma}{2} (|P_i|)^{\gamma} - 1 \} \tag{2.6}$$

P_i refers to the vertices in ith partition. v refers to the vertex to be assigned and $N(v)$ refers to the set of neighbours of v. α and γ are parameters. This is the scoring heuristics in FENNEL to decide a partition for a vertex to allocate. This heuristic assigns v in a partition with the highest score.

In some cases, same datasets generate streamed data repeatedly in a routine manner. A typical streaming algorithm is not possible in handling such situations. A re-streaming technique is needed in order to address this issue [81]. This algorithm is proposed over Linear Deterministic Algorithm [110] and FENNEL [113].

A re-streaming algorithm [81] was proposed by Nishimura et al. The authors considered the scenario in which the same datasets were routinely streamed. This re-streaming technique performed well when the same datasets appeared repeatedly in an application. However, the drawback of this study is that it is not suitable for the graphs in which changes occur very frequently. In such cases, data do not arrive in a routine manner or do not repeat their stream. Consequently, re-streaming technique has less impact in such scenario. A partial re-streaming was proposed [28] called hybrid streaming model. The proposed model allows the restreaming of portions of the graph when the rest of the graph uses single pass manner to complete the partition of the graph.

A comparison study[3] was completed with the several streaming partitioning algorithms [110, 113, 83]. This study compares the online graph partitioning performance with different applications and datasets.

2.4.2 Large-scale Distributed

Large-scale datasets have recently been gaining more attention, bringing with them many challenges due to their inclusion by several applications. Few studies have been undertaken to address large-scale data computation complexity using distributed systems. One of the most significant and successful distributed frameworks is MapReduce [24], however there are limitations in processing graph data using this framework. MapReduce is not suitable for computing iterative processes which are required for any graph data. A significant distributed framework named Spark was proposed in [131] to overcome the issues of MapReduce. Unlike MapReduce, the Spark framework can perform iterative computation. A workload streaming partitioning technique [32] was proposed to reduce inter-partition network communication. This technique is based on a well-known streaming graph partitioning heuristic [110] that allocates vertices according to the maximum number of edges of a vertex in a partition. The technique overcame a few issues faced by previous algorithms, for example, inter-partition traversals when executing and pattern matching queries.

STINGER [88], a framework to analyse streaming graph structured data, was proposed to facilitate portability, productivity, and performance for the research and development of big data. Its motivation was based on contemporary issues which can be formulated on the basis of storage and the changes of dynamic datasets over time, also known as the dynamic spatio-temporal graph problem. STINGER supports insertion and deletion of edges from scale-free graphs. Consequently, it allows fast query processing. Another streaming partitioning algorithm was proposed by Tsourakakis [112] and is called the planted partition model. This model uses higher length walks for graph partitioning. As a result, it achieved negligible computational cost and it significantly improved the partition quality. Wang and Chiu proposed a scalable streaming partitioning approach, aiming to achieve a low complexity system. This partitioning technique [123] aims to reduce the edges between partitions, and thus reduce the communication cost for query processing.

A scalable streaming partitioning approach has been proposed [123] aiming to provide a low complexity system. This partitioning technique aims to reduce edges between partitions; consequently, it reduces the communication cost of query processing. A streaming vertex-cut partitioning algorithm, High Degree, Replicated First (HDRF) [83] was proposed with the aim of utilising the vertices' characteristics. They use the greedy vertex-cut approach in which high degree (a vertex has more edges) vertices replicate first to minimise and avoid unnecessary vertex replication. This algorithm achieves a significant improvement for a stream-based partitioning algorithm over any other previous algorithms [71]. HDRF achieves nearly twice the speed up of traditional greedy placement and almost three times the speed of a constrained solution.

In the recent past, there has been considerable interest in designing algorithms and frameworks to handle massive graph data from streams of data. The ever-increasing stream of graph data can be partitioned into a cluster of nodes; the graph access pattern could be online or offline processing. This streaming partitioning is very efficient because graph loader or partitioner does the partitioning task. The partitioner loads the graph from the stream into the cluster [75]. A scalable streaming graph partitioning approach [91] provides horizontal and vertical scalability in a graph partitioning system.

Another streaming partitioning study called S-PowerGraph [126] over PowerGraph algorithm. S-PowerGraph uses vertex-cut partitioning rather using edge-cut.

Algorithms	Vertex-cut	Edge-cut	Distributed
Natural Graph Factorization [7]	Yes	No	Yes
METIS[62]	No	Yes	Yes
LOOM[33]	No	Yes	Yes
STINGER[88]	No	Yes	No
Planted Partition[112]	No	Yes	No
HDRF[83]	Yes	No	Yes
HoVerCut[91]	Yes	No	Yes
AKIN [132]	No	Yes	Yes

TABLE 2.3: Summary of Streaming Graph Partitioning

This method is suitable for partitioning skewed natural graphs and it outperformed previous studies in terms of acceptable imbalance factor.

Recently, a distributed graph processing algorithm called AKIN [132] in streaming manner was proposed. The algorithm measures the similarity of the degree of the vertices. Based on the statistics of similarity, the algorithm partitions the vertices. The similarity of statistics helps reduce the inter-machine communication to a greater extent.

In a traditional graph partitioning, the graphs must be loaded in order to partition and process them for any application purpose. That leads to the requirement of a huge storage capacity due to the large-scale of graph-structured data. Consequently, it requires high computation and communication costs during partitioning and processing of any application (for example, PageRank, shortest Path) after partitioning graph data. That is the main motivation for using streaming graph partitioning.

Additionally, there are a few other motivations in handling big streaming graphs; they are: i) Most of the applications generate large-scale dynamic data, which is impossible to store in the main memory of a single machine. ii) Big graph data makes an impression in analysing the complexity of streaming graph computation. iii) In many applications, emerging fastest growing data requires analysis in turn-around time. Table 2.2 shows the key findings of some existing streaming algorithms.Table 2.3 also shows a summary of some of well-known streaming graph partitioning algorithms.

2.5 Dynamic Graph Partitioning

In this section, we will discuss the behaviour of the dynamic graph and its partitioning technique. Table 2.4, shows some algorithms and their key findings. We also summarise some of the dynamic graph partitioning in Table 2.5.

Partitioning a large dynamic graph in a distributed system is an NP-hard problem [9]. Many applications such as social networks, communications networks, VLSI design and graphics have the dynamism in their graph structures from deleting or inserting new vertices and edges. Insertions and deletions of vertices or edges occur in this kind of graph as required by applications. For example, Twitter is one of the social networks that has a large dynamic data set; each new post in Twitter provides new information and is added to the graph. There are two dynamics in a dynamic graph: i) partially dynamic, if only insertions or deletions occur in the graph and ii) fully dynamic, when both insertions and deletions happen. This is subject to a graph's demands and structure. It is a way of changing the graph structure and dynamicity that makes the partitions unbalanced in a distributed graph-computing system. A dynamic graph is said to be incremental if insertion of a new vertex is allowed in the existing graph. On the other hand, a dynamic graph problem is called decremental if the deletion of a vertex or edge is allowed from an existing graph. A dynamic graph creates a new dimension in graph partitioning. Due to the updating of a dynamic graph, a vertex imbalance occurs among the partitions in a graph processing system. Maintaining a dynamic graph and a balanced dynamic graph partitioning is much needed to develop an efficient graph-structured data processing system. The process by which a dynamic graph data is distributed has not been well-researched. Data could arrive in real time, during the processing of an application. Distributing and maintaining the connectivity between nodes with newly arriving data is challenging. It is also challenging to make the system scalable. A study [69] has been completed which explored that the number of edges increases over time and the average distance between vertices shrinks over time. In addition, a graph generator was also proposed in this study, which requires fewer parameters to generate the full range of a graph.

A well-known dynamic load balancing system studied in [63] is called Mizan based on the Pregel framework. Mizan finds the load imbalance by using the run-time statistics of a graph. It uses the vertex migration method in a distributed manner without coordinating any central machine.

An adaptive dynamic partitioning, called xDGP [118] was developed to improve the graph partitioning performance. It uses an iterative vertex migration algorithm that relies on local information only. It has been demonstrated that a significant improvement was achieved in graph partitioning, reducing execution time by more than 50%. It also adapts the graph partitioning structure by balancing load with a large number of changes.

To compute PageRank in a parallel manner, a site-based graph partitioning and repartitioning technique [17] was proposed. Sparse matrix-vector multiplication is responsible for incrementally growing web matrices data, which are stored in a distributed manner. PageRank computation requires high-efficiency and low processing of overhead calculations because PagRank [12] computation has frequently repeated iterations. An algorithm [17] was proposed with a sparse-matrix multiplication technique to achieve high efficiency and parallelism, in order to focus on reducing th pre-processing overhead in a PageRank computation. Repartitioning techniques were used in this algorithm. A common problem in dynamic web data is the addition and deletion of new pages. A large graphs partition management system was proposed [129], in order to facilitate searching and mining a large graph. Minimising the inter-machine communication was the aim of developing this system. To perform the graph partitioning task, a two-level (static partition and dynamic partitions) structure was introduced which helped to improve the query response time and throughput. This two-level partitioning structure is effective as it adapts in real time when query workload is changing over time.

2.5.1 Adaptive Partitioning

Adaptive partitioning of a dynamic graph is a way to handle the changes of a graph over time. Parallel computing has been utilised to partition a large dynamic graph since the inception of large graphs. A dynamic partitioning technique [120] was proposed for adaptive unstructured meshes with parallel computing technology. This

Algorithms	Findings
Aggressive replication algorithm [77]	In-memory based dynamic partitioning. Achieved low-latency communication.
SPAR[84]	Achieved better data locality while minimizing the replication
xDGP[118]	Dynamically repartition a graph by adapting to structural changes. It adopts an iterative vertex migration with local information only.
Relative Optimisation [119]	A relative gain optimization technique used in vertex migration among different partition, while minimizing the degradation of partitioning quality.
Kineograph[21]	..
LEOPARD[48]	A replication algorithm is combined with a partitioning algorithm in aiming to reduce edge-cut. Local information has been considered in reassigning the vertices
Sedge[129]	A dynamic partitioning policy that supports large scale fasts queries processing
LogGP[128]	A log based partitioning technique that stores and uses that historical information for better partitioning result.
Sharding Networks[27]	Distributes large social network and evaluate several distributing strategies
Continual and Cost Effective[4]	A cost function and incremental partitioning technique proposed to allocate properly vertex from a dynamic graph
Multi-phase[97]	Multi-constraint and multi-objective partitioning technique to meet scientific simulation demand.

TABLE 2.4: Key Findings of Dynamic Graph Partitioning

algorithm uses a relative gain optimisation technique, which aims to balances workload and reduces the inter-partition communication overhead. A few series of adaptively refined meshes are applied for the purpose of the experiment and the results indicate that this provides better partitioning than a static partitioner. A distributed system, Kineograph [21], was proposed to handle the graph that changes rapidly; it is also able to capture the relationship between vertices. Kineograph also supports graph-mining algorithms to extract real time information from a fast changing graph. However, this system does not support dynamic partitioning.

Another adaptive partitioning method [117] has been developed for large-scale dynamic graphs. This method has been developed in addition to Kineograph. Additionally, this method was designed by using three different techniques to partition a graph. They are i) a most popular high scalability partitioning technique which is called modulo hash that was implemented in Pregel [72]; ii) another state-of-the-art streaming partitioning deterministic greedy heuristic [110], which is widely used in streaming graph partitioning; iii) an adaptive repartitioning heuristic. Repartitioning degrades the partitioning quality over time. To resist performance degradation the current method requires repartitioning the full graph; this can be cost and time effective with large-scale graphs. An adaptive approach was proposed in this algorithm to optimise the graph in every change and computation execution.

With consideration of connectivity and vertex degree, an online graph partitioning algorithm was proposed in [23], in order to achieve proper vertex locality. However, this algorithm does not support a streaming partitioning technique.

2.5.2 Vertex Replication

Dynamic graphs sometimes require a repartitioning job to keep graph-partitioned data balanced, in order to improve system performance. Good partitioning algorithms are featured with repartitioning techniques to handle huge dynamic graph data. Research has been undertaken on repartitioning online social network data [84]. The authors aim to improve scalability by reducing inter-partitioning communication. A replication method was used to reduce the communication among nodes. An in-memory based dynamic partitioning technique was proposed [77] to handle the large dynamic graph. This algorithm achieved significant low-latency communication in query processing. The authors provide a vertex replication policy that monitors the incoming vertices and decides what data to replicate. It has been evaluated on a social network graph, the result showing that this technique reduced the network bandwidth significantly. It also handles very large graphs efficiently.

Recently an algorithm called Lightweight Edge-oriented Partitioning and Replication for Dynamic Graphs (LEOPARD) [48], has been proposed for partitioning a dynamic graph. Two aspects of this algorithm are: i) a partitioning algorithm and ii) vertex replication algorithm. Replication was formed to provide the fault tolerance of the system by replicating the vertex in the case of any vertices being lost. This replication algorithm also helps to access the locality of a vertex; better access locality has greater performance in a partitioned system in processing any application. However, this system does not support the streaming graph partitioning technique.

A few approaches have been proposed on balancing the workload of a distributed system adaptively. Of them, dynamic replication based partitioning, proposed in

[129], is adaptively based on the change of workload. Another adaptive balancing workload is proposed in [102]. These approaches have been very effective in term of balancing the workload of a system. These systems have been designed for the BSP graph processing system that continuously updates vertex information with temporary data associated with running computations. However, on arrival of the new workload from a dynamic graph, these methods need to run for another partitioning results. Thus, there is no improvement in the partitioning results. With the aim of improving performance when workload changes frequently, a historical log-based portioning technique, called LogGP [128] was proposed. The LogGP framework analyses and reuses the historical statistical information to refine the partitioning result. It has great advantages of utilising the historical partitioning results to generate a hyper-graph. The authors claim that running statistic of historical partitioning logs could provide a greater improvement on partitioning results.

Alleviation load skew at query time is another benefit of vertices replication after distributing the large graph-structured network. In [27] this replication feature is presented. If there is no replication, popular nodes become overwhelmed by requests, in a partition for the value of those nodes.

Proper placement of a newly added vertex in a dynamic graph by using the cost-effective method, was proposed in a cost-effective partitioning method [4]. Several heuristics were also proposed to handle deletion or addition of edges. A vertex migration technique was also added here in the case of deleting a vertex, balancing all the partitions. These sets of heuristics also aim to reduce communication cost.

2.5.3 Multi-constraint Technique

Dynamic graphs applications can also be seen widely in scientific simulations to solve scientific real-world solutions. A multi-constraint and multi-objective [99] partitioning technique was proposed to meet some scientific simulation demand. Many scientific simulations, for example, multi-phase mesh-based computations, are not suitable for traditional single-constraint and single-objective partitioning [56]. Another example is multi-physics simulation in which several processes are simulated together, and which require a multi-constraint balanced technique to optimise the partitioning and load balancing [60]. Day by day numerical scientific simulation technologies are becoming more sophisticated because the number of processors is increasing. Consequently, partitioning techniques are required to optimise different types of objective functions in order to ensure good partitioning efficiencies.

Another multi-constraint graph partitioning technique was presented [98] to achieve optimisation from multi-objectives. This algorithm also meets the challenges of static and dynamic load balancing in multi-phase simulations. They propose both static and dynamic graph partitioners. They have two approaches in order to handle the dynamic graph partitioning; they are: i) start a new partitioning from scratch based on the new arrival of vertices and edges; ii) by using a diffusion-based method to balance the partition. In this case, the original partitioning needs to agitate enough to make all the partition balanced.

There are a few more techniques that we can use to handle dynamic graphs in a large-scale graph-partitioning problem. They are Clustering, Sparsification, and Randomisation. These are very useful techniques for the undirected graph. However, a few more techniques and data structure can be useful for a directed graph. They are Kleene Closure, Locality, Long Paths and Matrices.

There have been many works proposed in order to solve dynamic graph partitioning related issues in different perspectives. However, there are still some potential spaces in dynamic graph partitioning. Whenever new vertex insertion or deletion happens, a big graph does require repartitioning. The standard dynamic partitioning algorithm performs full graph partitioning, which is time intensive in terms of computation. We argue that repartitioning can be performed in optimising the new arrival data only, rather than repartitioning the whole graph. Partitioning quality degrades over time in a dynamic graph; we must perform the repartitioning task by minimising the edge-cut in such a way that will restrict the degradation of partitioning quality.

2.5.4 Scalable Dynamic Partitioning

Existing parallel graph partitioners (such as ParMetis) produce good quality partitioning but their scalability is poor. A parallel scalable graph partitioner was proposed [65] to scale the large-scale graph data. The authors proposed a method using a lattice-based multilevel technique for the coordination. However, the proposed

Algorithms	Vertex Replication	Adaptive Partitioning	Scalable	Dataset
Aggressive replication algorithm [77]	Yes	No	Yes	Static
SPAR[84]	Yes	Yes	Yes	Static
xDGP[118]	No	Yes	Yes	Dynamic
Relative Gain Optimisation [119]	No	Yes	No	Static
Kineograph[21]	Yes	No	Yes	Dynamic
LEOPARD[48]	Yes	No	Yes	Dynamic
Sedge[129]	Yes	Yes	Yes	Static
LogGP[128]	No	No	No	Static
Sharding Networks[27]	Yes	No	Yes	Static
Continual and Cost Effective[4]	Yes	No	No	Dynamic
Multi-phase[97]	No	Yes	No	Static

TABLE 2.5: Summary of Dynamic Graph Partitioning

method slightly compromised the graph quality to reach the maximum scalability. A scalable dynamic graph processing framework called GraphTau [50] was proposed on top of Apache Spark. GraphTau continues to create the snapshot of a graph over time to analyse it for partitioning or other analytical purposes. Processing a time-evolving graph in a distributed system requires efficient task management and fault tolerance; the correlation between snapshots is challenging. GraphTau provides an efficient technique in streaming graph processing and an incremental computational model that helps the coordination between graph snapshots to attain an efficient computation. A robust dynamic graph data management system was proposed [68]. The proposed dynamic data management method uses a replication technique with an updated graph snapshot. A scalable technique is used to adjust the number of servers and workers based on the updated graph snapshots in each time interval. All the worker or server machines may not be fully utilised in this method. In addition, cost optimisation was not also considered in this proposed study. For example: if the graph snapshot updates over-time in the large worker machines and 50% utilise it, this workload could have been processed by the small machine.

2.6 Resource Scalability

A number of resource scalable technique was proposed in cloud computing. In this section, some of the existing recent work on scaling resources in the cloud computing is discussed.

Cost is one of the important factors to consider in scaling the resources in the distributed computing. A cost-efficient auto-scaling for the cloud environment is proposed in [89]. This method is software container-based, and is lightweight and best suited to fine-grained computing. This work also offers a rescheduling algorithm to support the best use of resources by scaling in or out, as per the needs of the computational load.

A predictive auto-scaling method [90] was proposed which forecast the computational load in advance based on the limited horizon. This helps to allocate the resources and adjust the number of resources in order to minimise the cost. Jiyun et al. proposed a novel optimising algorithm [104] for resource provisioning at the virtual machine level. The method utilises queueing theory to decide the number of machines required to provide for each service.

A lightweight resource scaling method [44] was proposed to minimise the resource cost in cloud applications. It provides a fine-grained scaling of different resources such as CPUs, memory, and Input/Output and so on, while minimising the cloud provider's cost.

A power and migration cost optimisation method was proposed for cloud computing in [40]. It is based on a Service Level Agreement (SLA) and a convex optimisation heuristic is employed, in order to minimise the power and migration costs.

In [18] a resource provisioning method was proposed to optimise the cost for the cloud resources. It supports multiple and long-term provisioning. This is also known as reservation and an on-demand plan. The algorithm is able to trade-off between the reservation of resources and the allocation of on-demand resources.

2.7 Gap Analysis

In this section, we discuss the research graph in relation to graph partitioning, particularly a time-evolving graph. Based on literature studies, the resource utilisation in the cloud environment for the streaming graph partitioning has not yet been studied. There are a lot of auto-scaling algorithms was proposed to scale the cloud machine resources with the demand of requests to process in a distributed machine. However, there has not been any auto-scaling study done for the streaming graph partitioning algorithm. This dissertation will address the issue in resource utilisation and cost optimisation in the cloud environment for the streaming graph partitioning algorithm.

A number of streaming algorithms have been proposed to address the memory bottle-neck issue in a one-pass manner while minimising communication and the balancing load. However, no algorithms were evaluated in the cloud environment and most of the algorithm did not consider real-world application with a real-world dataset.

To the best of our knowledge, a dynamic graph partitioning algorithm in a streaming manner has not yet been considered. We provide a streaming graph partitioning for a dynamic graph dataset. In addition, a dynamic scaling method is also proposed to optimise the resources and cost.

2.8 Summary

In this chapter, graph structure, traditional graph partitioning and dynamic graph partitioning in a streaming manner techniques are reviewed thoroughly. This chapter also analyses the existing graph partitioning techniques and discusses their limitations in depth. A summary of key findings and contributions from existing work are presented in the tables. We categorise the different graph partitioning and frameworks in different sections to analyses the limitations of the existing research.

We also discuss the dynamic graph partitioning categorically in several sections such as i) Adaptive partitioning ii) Vertex Replication iii) Multi-constraint technique and iv) Scalable dynamic partitioning.

Resource scalability and cost optimisation in the cloud environment are also discussed and analysed in this chapter.

Finally, in this chapter we discuss the gap analysis based on the literature studies. Dynamic graph partitioning in a streaming manner is necessary at this time, in order to meet recent dynamic graph partitioning applications.

Chapter 3

Evaluating Distributed Streaming Graph Partitioning in the Cloud

This chapter describes the implementation of two existing streaming graph partitioning heuristics in the cloud. The PageRank application is also explored after partitioning a graph in a streaming manner. We observe how the streaming partitioning algorithm behaves in the cloud environment in terms of communication, time complexity and allocating resources with the different experimental scenarios.

3.1 Motivation

Recently graph-structured data has been growing exponentially. It is becoming challenging to query and process, and to manage continuously increasing large volumes of streaming graph-structured data arising over time in many applications, such as social networks, biological data, communication networks and so on. Partitioning a graph into the disjoint partition is the way to distribute graph computational load. Streaming graph partitioning is the variant of graph partitioning which takes the graph input in a one-pass manner. It does partition a graph using limited information of a graph rather than using whole graph information. Cloud computing has a significant advantage in accommodating large-scale graph datasets. However, the performance evaluation of streaming graph partitioning with a real application is yet to be done in the cloud environment. Moreover, suitability of the streaming graph partitioning algorithm in the cloud environment has not been evaluated.

This chapter evaluates the performance of two streaming graph partitioning heuristics with the PageRank application in the cloud environment with the different experimental scenario. This study observes the suitability of the streaming graph partitioning algorithm in the cloud environment and finds potential limitations in terms of scalability, minimising communication and load balancing.

3.2 Introduction

Today, graph data are huge. These huge datasets are: knowledge graph, web data, social network data and biological networks, to name few. These data can be formed as graph data and many of the real-world networks or graph data change continuously over time, generating large dynamic graphs. Example of these are online activity and interactions formed from electronic communication, social media, and content sharing. These processes produce huge amount of continuous, interactive data, which is represented as a dynamic graph. A recent statistic shows that Twitter has over 43 million users and there are more than 1.5 billion social relationship over this network. This tells us that the trend of the exponential growth data is significant. In graph streams, individual edges of the underlying graphs arrive sequentially in a stream, unlike traditional graph-data which has a fixed number of vertices and edges. The rapid growth of data in many emerging applications (for example, online social networks, web graphs, health informatics, financial analyses and monitoring, public policies and monitoring, protein-protein interactions) needs streaming analysis in real time as data arrives.

In order to meet the huge graph data processing demand, efficient systems are a necessity. A distributed system is one of the best solutions to allocate an enormous amount of streaming graph data before processing. Graph partitioning is the technique, which distributes numerous graph data between distributed systems. This partitioning technique distributes the graph-structured data to distributed machines with the aim of minimising graph node communication. This partitioning is called $k-$way partitioning [9] which is an NP hard problem, meaning that best partitioning solution is difficult but an optimal solution is possible. Much research has been undertaken in order to compute large-scale graph data; of them the most popular are: [4, 110, 103]. Google proposed a vertex-centric distributed graph processing framework which is known as Pregel [72]. It aims to process large-scale graph data by using a message passing technique from one node to another. It uses a Bulk Synchronous Processing system to pass the message among nodes and process them in parallel.

Streaming data has had great usage in recently and state-of-art applications such as social networks, communication networks, weather forecasts, biological networks and web data where edges of underlying graph are received and updated sequentially. For example, new accounts are created and deleted every day in online services such as Facebook, Skype and Twitter. Stream data arrives in real time in many real-world applications. It is necessary to partition these streams of graph data in a single pass manner because there is only one chance to read the data. It is also known as streaming graph partitioning. This is very necessary to provide an effective solution in dealing with enormous streaming data and to meet the future demand. An offline graph partitioner requires the entire graph information to be presented in memory, whereas a streaming graph partitioning technique is able assign vertices as they arrive. It is extremely necessary to have efficient graph partitioner of dynamic graphs. The one-pass algorithm has a chance to read input data only once. Thus, it is challenging to receive them and allocate them to the properly partitioned location by balancing the load. It is also important to maintain good locality over a stream of a vertex in order to achieve optimal and good partitioning. A number of works [110, 109] have been produced in the recent past to meet this challenge.

Cloud computing is a well-known distributed computing environment which is comprised of interconnected and virtual computers that could be dynamically increased or decreased on demand [15]. It has much more flexibility than any other shared nothing cluster system. Given the success of distributed computing, cloud storage has become one of the most popular distributed systems due to its cost effectiveness, quick deployment facility and easy access to information. Single machine storage approaches do not scale because of their limited capacity. Due to an ever increasing size of the graph, application deployments are moving from small-scale cluster servers towards the cloud which provides massive storage and significant parallelism. A cloud consists of tens of thousands of inter-connected machines, which provide much more flexibility in deploying graph data. On top of that, cloud computing has tremendous data recovery options. To the best of our knowledge, streaming graph partitioning performance has not been evaluated in the cloud distributing environment. This study evaluated the performance of two streaming graph partitioning algorithms in the cloud computing environment, with a combination of different resources and locations.

An evaluation study [3] was completed on LDG algorithm[110] and presented some insights of this streaming graph partitioning algorithm in terms of its edge-cut and load balancing performance with other similar algorithms. However, in our experimental studies, we explored the algorithm to find the potential research problem in resource optimization in a cloud environment in order to evaluate the performance of edge-cut and balancing load. Particularly we also observed how different type of memory has an impact on partitioning time.

3.3 Streaming Graph Algorithms

Two state-of-the-art streaming graph algorithms have been used in this evaluation study which was completed by Stanton[110]. Stanton et al. proposed some heuristics to address memory bottleneck in graph partitioning. Technically, these algorithms are established in streaming graph partitioning techniques. They are the most popular and most cited streaming graph algorithms. The notation P^t refers to the set of partitions at time t. $E(v)$ refers to the set vertices that v is connected with. These algorithms are explained below:

3.3.1 Balanced

In balanced partitioning, the algorithm assigns an arriving vertex v to a partition which contains a minimum number of vertices.

$$partitionIndex = \arg\min_{i \in [k]} |P_i^t| \tag{3.1}$$

3.3.2 Linear Deterministic Greedy Algorithm

Streaming graph partitioning makes decision based on incomplete graph information. However, in *t* time, the Linear Deterministic Greedy (LDG) algorithm uses a greedy approach to assign a vertex to a particular partition. It tends to assign a vertex to a partition which has the most edges. To balance the load of the partitions, it weighs the partition by a penalty function based on the capacity of the partition.

$$partitionIndex = \underset{i \in [k]}{\arg\max} |P_i^t \cap E(v)| \left(1 - \frac{|P_i^t|}{C_i}\right) \tag{3.2}$$

Where, C_i is the capacity of *ith* partition, P_i^t is the set of vertices in *ith* partition over time *t* .

3.4 Graph Applications: PageRank

There are several real applications available [80], which generate massive datasets and those can be formed as graph datasets. Thus, graphs have become key components of a wide range of applications, such as PageRank, Connected Component, protein interactions, semi-supervised learning based on random graphs walks, web searches based on link analysis, scene reconstructions based on Markov random fields and social community detections based on label propagation, to name just a few examples. However, we use two existing streaming graph partitioning algorithms to evaluate performance on the PageRank application, because they are widely used and very popular for large-scale graph-structured applications.

The most well-known application for graph processing is ranking web pages, also known as PageRank, which was proposed by Google [82]. It calculates the linking relationships between web pages. The web can be represented as a giant graph, in which the web pages are nodes and the links from one page to another are edges. For example, if many people follow a Twitter user, the user will be ranked highly. Every hyper link carries a vote for the page to which it is linked. The more links a page has, the more rank it gains and it becomes defined as a well-ranked page. The PageRank algorithm does not give a rank for a whole website; it gives a rank for an individual page of a website. PageRank is defined by the following formula:

$$PR(A) = (1-d) + d * \frac{PR(T_1)}{C(T_1)} + ... + PR(T_n/C(T_n)) \tag{3.3}$$

Where, $PR(A)$ =rank of a page A, $PR(T_1)$ = rank of a page T having a link to page A, $C(T_1)$ = number of links to another page from page T and d is the damping factor.

3.4.1 Distributed PageRank

We implement a distributed PageRank calculation. Our implementation takes the static graph data as input in a stream manner. The streaming technique takes one vertex with its associated edges and the partitioning algorithm decides the respective partition. The partitioning algorithms in the server machine assign the vertex to the proper machine. After completing the partitioning and distributing all the vertices of a graph, the client starts calculating the PageRank. To facilitate calculation, we set an initial PageRank value for each vertex at 1.0 in its first iteration. After completing the calculation on each client side and in every iteration, the client updates their PageRank value to the server side. During calculation, a client machine might need a ranking value of a vertex which does not belong to the same machine. In such a case, the client sends a request to the server side for that respective rank of that particular vertex. The Server sends the particular client's requested PageRank of a vertex. The computation continues until the last two iterations values become equal.

3.5 Experimental Design

We evaluate two well-performed streaming graph partitioning heuristics in the cloud environment and apply them to a widely used application PageRank. We use two graph datasets from a graph data archive[121]. Table 3.1 shows the characteristics of datasets.

Name of Dataset	$\lvert V\rvert$	$\lvert E\rvert$	Type
3elt (Synthetic) [121]	4200	13722	Finite-element mashes
4elt (Synthetic)[121]	15,606	45,878	Finite-element mashes

TABLE 3.1: Characteristics of Datasets

Figure 3.1 illustrates the whole processing framework of this study. We receive data in a stream order one by one before assigning a vertex to the respective partition. The partitioning algorithm assigns the vertex as it arrives and sends it to a cloud machine. The moment the cloud machine receives the vertices, it immediately starts processing the calculation of PageRank.

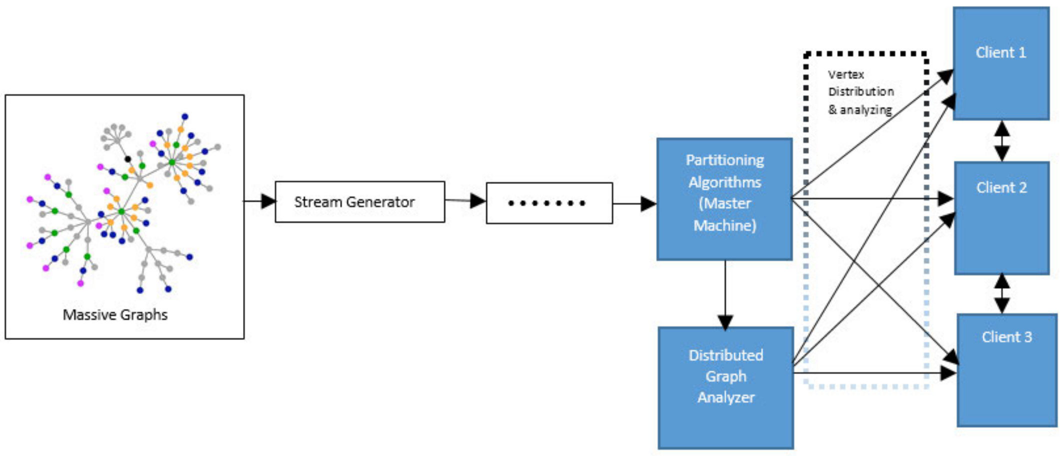

Massive Graphs
Stream Generator
Partitioning Algorithms (Master Machine)
Distributed Graph Analyzer
Vertex Distribution & analyzing
Client 1
Client 2
Client 3

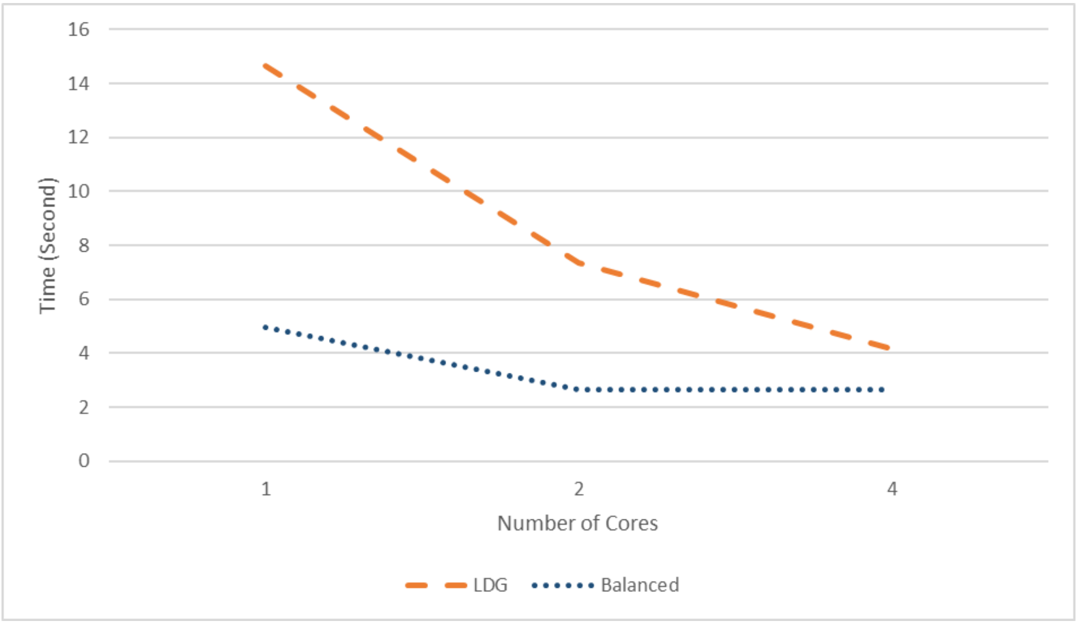
FIGURE 3.1: Experimental method

We used the Nectar Cloud virtual machine [2] to evaluate the implemented algorithms. The following experimental scenarios have been used to evaluate the performance of those algorithms in the cloud machine.

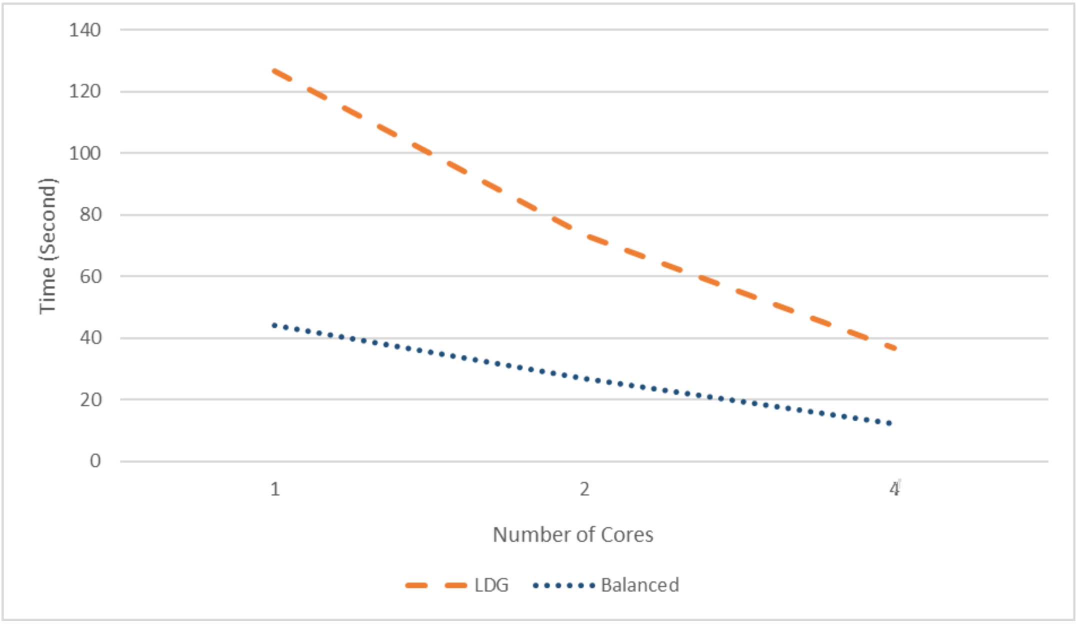
1. Different number of partitions (2, 4, 8) with the 4 cores in master machine, whereas number of cores in client is 2.
2. Different number of cores (e.g 1, 2, 4, 8) on master machine with the fixed 2 cores on client machines, and number clients are 8.
3. Different location for master machine to observe the efficiency in terms of distance. Number of cores in client machine was 2 and with the 4 cores in master machine in this configuration.

3.6 Result Discussion

We considered different scenarios to observe the performance of those algorithms with the different resources in the public cloud. A different number of cores in the master machine, a different number of clients and master machines residing in different locations have significant impact on execution time and the number of messages exchanged between client and server.

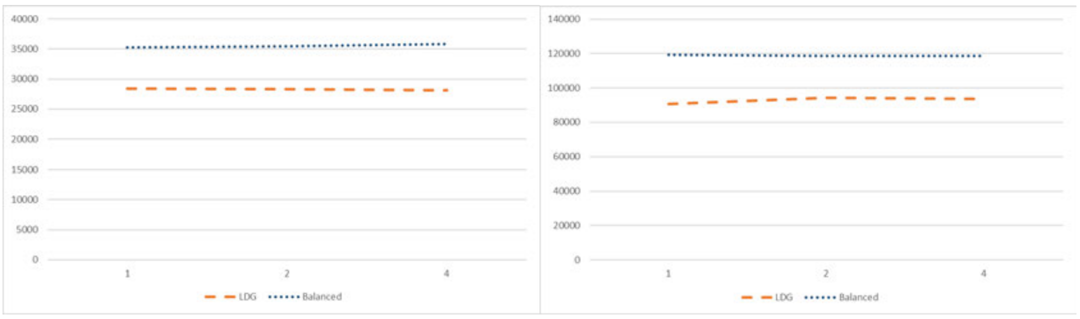


(A) 3elt Dataset



(B) 4elt Dataset

FIGURE 3.2: Execution time with different cores in Master Machine

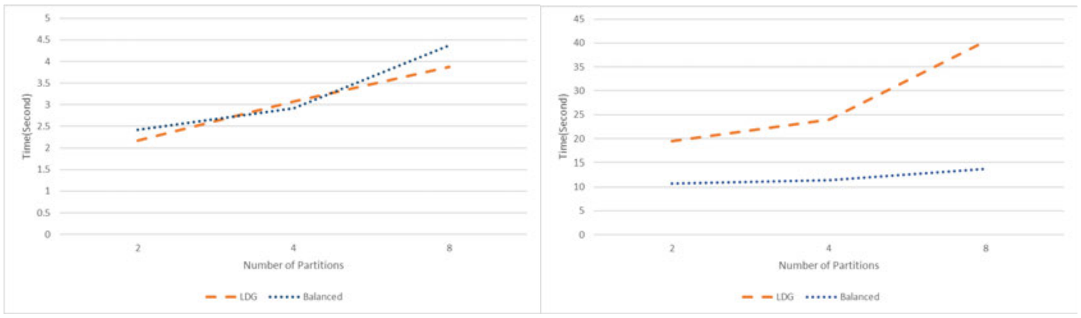


(A) 3elt Dataset (B) 4elt Dataset

FIGURE 3.3: Number of messages with different cores in master machine

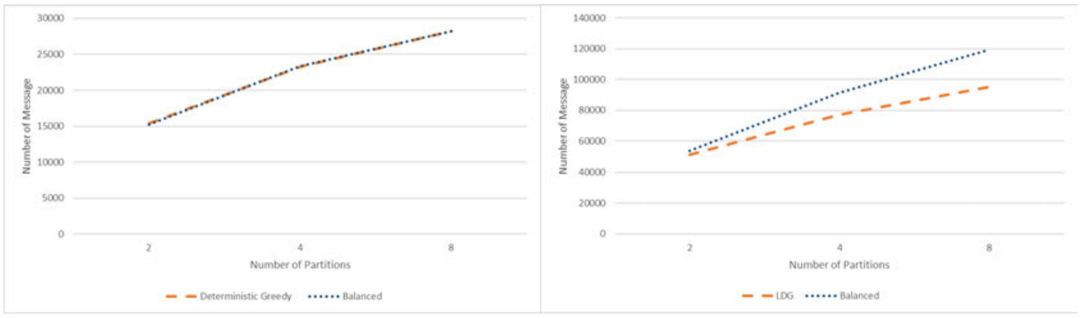
Figure 3.2 and 3.3 illustrated the execution time and number of messages in calculating PageRank of LDG and Balanced algorithm with different number cores (1,2,4) in the master machine. It is seen clearly that the execution time decreases as the number of cores increases, because more number cores in a machine allow more messages to be transferred on the network at the same time. Figure 3.3a and 3.3b shows that the number of messages exchanged among master machines and client machines decreases slightly as the number of cores increases for the LDG and Balanced algorithm, respectively, in both datasets, which is to be expected.

We also observe the execution time and number of messages exchanged between client and server for both algorithms by varying the number of partitions. As shown in Figure 3.4a and 3.4b, PageRank execution time increases as the number of partitions increases, which is expected. This tell us that, by increasing the number of partitions, it creates more communication messages on the network as shown in Figure 3.5a and 3.5b because the connected vertices are residing in an increased number of partitions which creates more edge-cut rates among partitions. However, it is noticeable in Figure 3.5a that the number of messages exchanged for both algorithms are the same as for the 3elt dataset. On the other hand, for the 4elt dataset, the LDG algorithm performed better in terms of communication. In conclusion, it is noticed that the LDG algorithm performed better compared with the Balanced algorithm in different aspects, such as communication and execution time for both datasets.



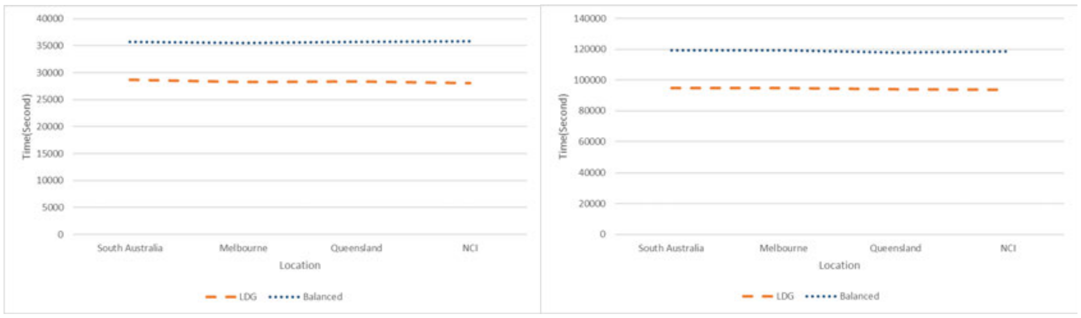
(A) 3elt Dataset (B) 4elt Dataset

FIGURE 3.4: Execution time with four cores in master machine



(A) 3elt Dataset (B) 4elt Dataset

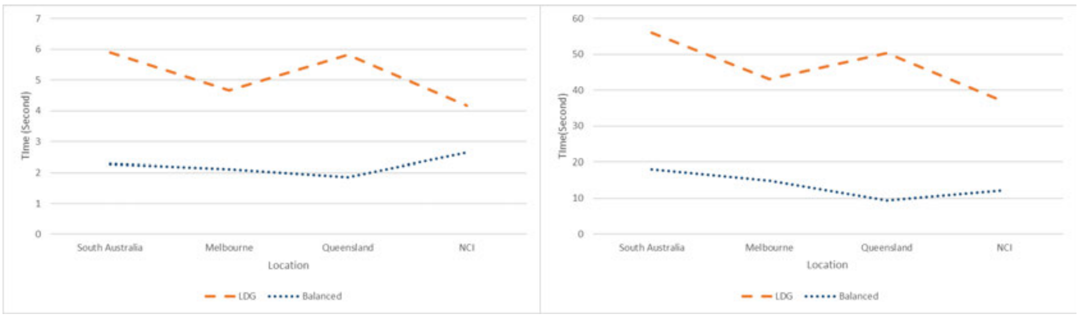
FIGURE 3.5: Number of Messages With Four Cores in Master Machine



(A) 3elt Dataset (B) 4elt Dataset

FIGURE 3.6: Number of messages with the master machine in different location

We also evaluate these algorithms by varying the location of the master machine in the cloud. We used four locations: i) South Australia, ii) Melbourne iii) Queensland iv) NCI (Canberra). The number of cores in the master machine was 4. Client machines were always located in NCI (Canberra) in this scenario. Figure 3.7 and



(A) 3elt Dataset (B) 4elt Dataset

FIGURE 3.7: Execution time with the master machine in different location

3.6 depicts the execution time and the number of messages exchanged for both algorithms, for calculating the PageRank, by allocating master and client machines in different locations. It is noticeable that execution time varies in terms of distance among server machines and client machines. The number of messages slightly varies in changing the location as Figure 3.6a and 3.6b show.

It is obvious in Figure 3.7a and 3.7b that for both datasets for the LDG algorithm, the location of the master machine (NCI) and the client machine (NCI) in the same area reduces the execution time. It is also observed that the highest execution time occurs if the master machine and client machine reside in South Australia and NCI (Canberra) respectively.

3.7 Related Work

Researches have been undertaken on graph partitioning in the cloud environment. In this section, we review some related work on streaming graph partitioning and gap analysis.

In the recent past, there has been considerable interest in designing algorithms and frameworks in order to handle massive graph data from streaming data. Steaming graph-data can be partitioned into a cluster of nodes; the graph access pattern could be online or offline processing. This partitioning is very efficient because the partitioner assign the graph data as they arrive. Partitioner loads the graph from the stream order into the cluster [110]. There are some motivations in handling big streaming graphs. They are: i) most of the applications generate large-scale dynamic data, which is impossible to store in the main memory of a single machine. ii) Big graph data makes an impression in analysing the complexity of streaming graph computation.

Another well-known streaming partitioning algorithm called FENNEL [113] was proposed. FENNEL has made great improvements in reducing the computational complexity and reducing communication near to the METIS algorithm. However, these algorithms were not evaluated in the cloud environment.

Microsoft research implemented a partitioning framework, a top performing graph-partitioning framework called FENNEL [113]. This partitioning framework aimed to overcome computational complexity issues in the traditional balanced graph-partitioning problem. It has also improved performance regarding communication cost and runtime. In some cases, the same datasets generate streamed data again and again in a routine manner. A typical streaming algorithm can not possibly handle this situation. A restreaming technique is needed in order to resolve this issue [81]. This algorithm is proposed over Linear Deterministic Algorithm (LDG) [110]and FENNEL [113].

A Scalable cloud based graph partitioner was proposed in [74]. The authors aim to scale the size of graphs and compute cores adaptively. Although the proposed algorithm is able to adjust with changes in the graphs, it is unable to partition a graph in a one-pass manner. Another cloud based graph partitioning framework called Trinity [103] was proposed to provide a memory infrastructure and a data structure. Trinity is able to partition online and offline graph for several real-world applications.

The evaluation of streaming graph partitioning algorithms in the cloud system and observing their performance with different real applications are yet to be undertaken. There is a fundamental diversity in the different graph applications domain and in partitioning them. For example, social graphs and web graphs are very different in their structure [22, 115]. Ever increasing growing data and dynamism in the graph creates more challenges to find an optimal solution in big graph processing applications. Some applications fit well into a good locality, regardless of how good the minimum edge-cuts are and vice versa. This is important in deciding a suitable streaming graph partitioning algorithm for some specific real applications.

Cloud computing in the recent past was paid great attention in the area of distributed systems and parallel processing. Some reasons which could make computing a very important solution for many applications are worth mentioning here. They are: high throughput, capacity to compute the exponential growth of data, greater capability of scale out, up, or down, and dynamically load balancing adaptability. This work also observes the above performance matrices of the cloud system with the various graph partitioning algorithms and with real-life applications.

3.8 Summary

This chapter evaluated the performance of two streaming graph partitioning algorithms in the cloud environment. It observed how cloud resources and locations have an impact in stream graph partitioning. In huge graph partitioning in a one-pass manner, the locality of the cloud machine and different resources has a great impact on partitioning performance. We observed that allocating different machine in different locations in the same cloud environment consumes more computational time in exchanging messages between machines. Increasing resources in a cloud machine also has another impact in streaming graph partitioning in the cloud environment. We observed that, as we increase the resources in a machine, performance gets better in terms of reducing the execution time and the number of communications.

This chapter motivated the need for streaming graph partitioning for the scalable dynamic graph in the cloud environment. We identified a few graph partitioning problems in streaming manner after evaluating these existing algorithms. Utilising the cloud machine's memory, optimising the resource cost, minimising communication, reducing time complexity and load balancing are key challenges in streaming partitioning in the cloud environment. To address the identified research problem, we designed algorithms and models in Chapters 4 to 6.

Chapter 4

ASP: Auto-Scaling for Streaming Graph Partitioning in Cloud

This chapter introduces a auto-scaling algorithm for a streaming graph partitioning in the cloud environment. A provision/de-provision of a type of machine method is also introduced in this chapter to optimise the resource cost. We present an existing graph partitioning algorithm to evaluate our proposed auto-scaling algorithm in the cloud environment.

4.1 Motivation

Streaming graph data is ever growing in recent applications, such as PageRank, the Web and social networks. Due to time constraints and memory bottlenecks, a single machine is unable to process this large, ever increasing data. An efficient distributed application is necessary in order to process a graph-oriented application efficiently. The number of machines has to be determined in advance, based on the upcoming streaming, in order to better utilise machine resources. In addition, to cater for the ever-increasing computational load from streaming graph data and to avoid consuming unnecessary resources, an auto-scaling distribution system is required. Moreover, graph computation workload continues to change over time, thus machine provision/de-provision is necessary to utilise the machine resources and optimise the cost. The public cloud has an elasticity feature to handle provision and de-provision as per demand, without any human interaction. Most of the real-world graph data applications are migrating to cloud services in order to cope with ever-growing data and removal of data. This study proposes a cost-based auto-scaling strategy in order to best use a number of machines and cloud resources in graph -partitioning. The following contributions are from this study: 1) Determine the number of machines based on predicted data stream; 2) A cost-efficient auto-scaler which scales up/down the machine resources as per the demands of work-load. Experiment results shows that the proposed auto-scaler in streaming graph partitioning has significant improvement of memory utilisation of resources while minimising the resource cost.

4.2 Introduction

In recent years, streaming graph data has been increasing in an unprecedented manner in graph-oriented, real applications. Moreover, graph-oriented applications tend to change their structure over time. Therefore, a dynamic auto scaling cloud service is required to cope with data changing over time. Due to the ever-increasing graph data, a distributed graph processing is the solution to handle the unprecedented graph data. Scalability is one of the solutions in utilising the cloud resources as per user demand.

Currently, graph data are huge. These large datasets are: knowledge graphs, web data, social network data and biological networks, to name a few. These data can be formed as graph data. Many of the real-world networks or graph data are continuously changing over time and this generates large, dynamic graphs, for example, online activity and interactions formed from electronic communication, social media and content sharing. These processes produce a huge amount of continuous, interactive data, which is represented as a dynamic graph. A recent statistic shows that Twitter has over 43 million users and there are more than 1.5 billion social relationships over this network. This tells us that the trend of the exponential growth in data is significant. In graph streams, individual edges of the underlying graphs arrive sequentially in a stream, unlike traditional graph-data which have a fixed number of vertices and edges.Fast growth of data in many emerging applications (for example, online social networks, web graphs, health informatics, financial analysis and monitoring, public policy and monitoring, protein-protein interactions) need streaming analysis in real time as data arrives.

In order to meet the huge graph data processing demand, efficient systems are a necessity. A distributed system is one of the best solutions for the allocation of an enormous amount of streaming graph data before processing. Graph partitioning is the technique which distributes numerous graph data among distributed systems. This partitioning technique distributes the graph-structured data among distributed machines, with the aim of minimising graph node communication among machines. This partitioning is called k-way graph partitioning [9], which is an NP hard problem. There has been much research in order to compute large-scale graph data, the most popular of them being: [4, 110, 103]. Google proposed a vertex-centric distributed graph processing framework which is known as Pregel [72]. It aims to process large-scale graph data by using a message passing technique from one node to another. It uses the Bulk Synchronous Processing system to pass the message among nodes and process them in parallel.

Cloud computing is a well-known, distributed computing environment which is comprised of interconnected and virtual computers that could be dynamically increased or decreased on demand [15]. It has much more flexibility than any other shared nothing cluster system. Given the success of distributed computing, cloud storage has become one of the most popular distributed systems, due to its cost effectiveness, quick deployment facility and ease of accessing information. Single machine storage approaches do not scale because of their limited capacity. Due to the ever-increasing size of the graph, application deployments are moving from small-scale cluster servers towards the cloud, which provides massive storage and significant parallelism. A cloud consists of tens of thousands of inter-connected machines, which provide much more flexibility in deploying graph data. In addition to that, cloud computing has tremendous data recovery options.

Due to a highly increasing amount of graph data in a graph-orientated application, it is convenient to predict the number of task requests by users or upcoming data in advance. Predicting the data helps in utilising the cloud resources and optimising the resources cost as well. Additionally, it is also important to provision or de-provision the currently used machines, according to the computational load, as the data increases or decreases over time in dynamic graph-oriented applications. This study addresses the above research problem by providing an auto-scaling mechanism. The proposed study utilises the cloud resource and optimises the cost as per the demand of the ever-increasing graph data. It uses a heterogeneous distributed cloud environment for the evaluation of an existing streaming graph data partitioning algorithm.

The proposed novel auto-scaling algorithm has the following contributions:

- A prediction model to predict the number of user requests and upcoming graph data in advance. Based on the predicted data, the proposed model decides the number of Virtual Machines(VM) in the cloud.
- The proposed auto-scaling algorithm, provision/de-provision the type of machines being used, based on the current computational load during the partitioning.

4.3 System Work-flow

In this section, we discuss the overall system architecture for cost-based resource utilisation method proposed in this study. Figure 4.1 shows the work flow of our proposed auto-scaling method.

Stream Data: Stream of data are sent to the queue for the prediction of required number of machine. This stream of data also sends the vertices and edges to the partitioning algorithm also receives the vertices and edges from the stream for the partitioning purpose after deciding the required number of machines.

Queue: The queue receives the graph data from the stream of graph. The queue is used to put the vertices for purpose of prediction.

Prediction Algorithm: In this work-flow, the prediction algorithm is responsible for the prediction of the upcoming data for a unit of time from the buffer. Based on the predicted data, the algorithm is also responsible for determining the number of machines is required to complete the computation.

Partitioning Algorithm: In this work-flow, the partitioning algorithm is responsible for the partitioning of the graph input from the stream to the appropriate machine.

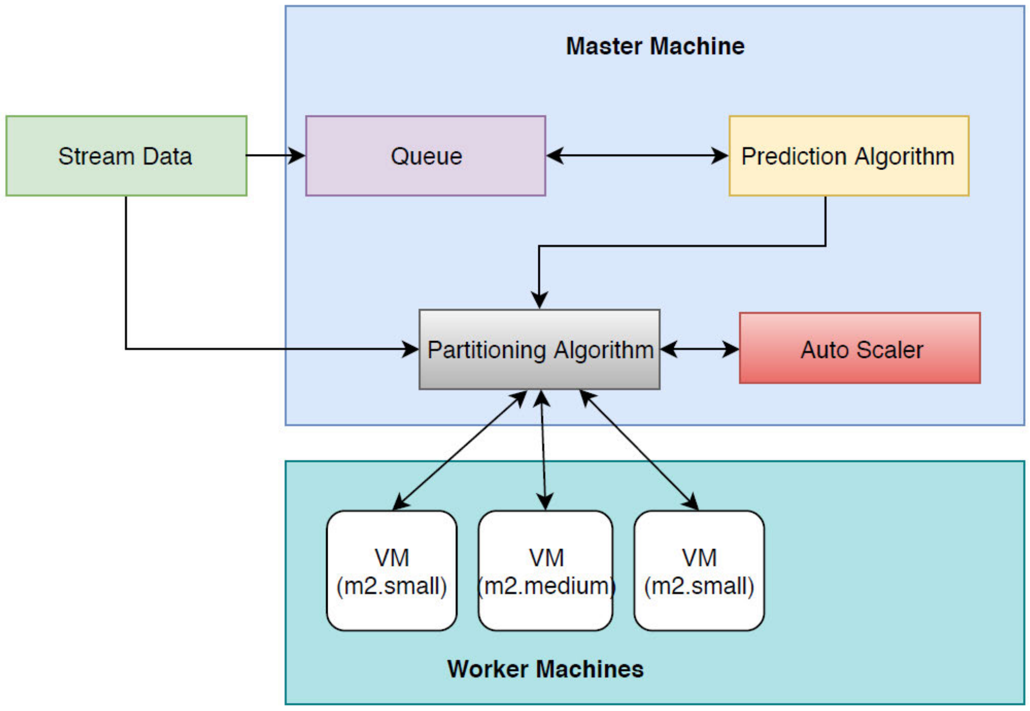


FIGURE 4.1: Auto-scaling work flow

Auto Scaler: The auto-scaler checks the partitioned data in real-time, in order to provision/de-provision the machine as per the computational demand.

4.4 Linear Deterministic Greedy Algorithm

In this section, we discuss the streaming partitioning algorithm we use to evaluate our cost-based graph computation method in the cloud. This algorithm is a state-of-the-art algorithm for streaming graph partitioning as proposed by Stanton [110].

Streaming graph partitioning makes decisions based on incomplete graph information. However, in *t* time this increasing amount Linear Deterministic Greedy(LDG) algorithm uses a greedy approach to assign a vertex to a particular partition. It tends to assigns a vertex to a partition where it has the most edges. To balance the load of the partitions, it weighs the partition with a penalty function based on the capacity of the partition.

$$partitionIndex = \arg\max_{i \in [k]} |P_i^t \cap E(v)| \left(1 - \frac{|P_i^t|}{C_i}\right) \tag{4.1}$$

where, C_i is the capacity of *ith* partition, P_i^t is the set of vertices in *ith* partition over time t.

4.5 ASP: Auto-scaling method

We use a prediction model to predict the stream of data for the next unit of time. We take the following into consideration to design the prediction model: 1)the speed of the streaming data coming in; 2) the size of the data in the buffer; 3) the capacity of worker machines; 4) the resources of a worker machine; 5) waiting time

We use a time scale T; the data arrives in a time series each interval data arrived in a series s_i and $s_i + 1$. The stream of data comes in a buffer and is stored in a queue. The model takes the data from the last unit of time and measures their sizes and arrival speed to determine the next unit of time.

We use the queueing theory model $M/M/\alpha$ [124], where $M =$ the input process, we use the Poisson distribution model to process the input. In the input process the arrival rate is $\lambda = arrivedTuple/sec$. Another M in the second part of this model is the application service completing rate $\mu = taskComplete/sec$, and s is the number of worker machines. The utilisation of each worker machine is calculated with the following equation:

$$\rho = \frac{\lambda}{s\mu} \tag{4.2}$$

The predicted stream data waiting in the buffer for a time unit can be defined with the following equation:

$$predictedData = \frac{\rho \alpha^s p_0}{s!(1-\rho)^2} \tag{4.3}$$

p_0 is the probability of having no stream of data in the buffer. p_0 can be defined with the following formula:

$$p_0 = \left[\sum_{r=0}^{s-1} \frac{\alpha^r}{r!} + \frac{\alpha^s}{s!(1-\rho)} \right]^1 \tag{4.4}$$

The expected waiting time of the stream of data can be calculated with the following formula:

$$waitingTime = \frac{\alpha^s p_0}{s! s\mu(1-\rho)^2} \tag{4.5}$$

Based on the predicted data in the stream (*predictedData*) from the Equation 4.3, our predicted model determines the number of worker machines that need to be allocated to accommodate the upcoming stream of data. The following thing also need to be considered: 1) the current capacity of the workers; 2) the size of the stream waiting in the buffer.

We can obtain the number of estimated machines (s) with the following equation:

$$s = \lceil \mu + \sqrt{\mu} \rceil \tag{4.6}$$

We use a capacity threshold to determine the number of machines we need to accommodate the predicted data in the stream. The capacity threshold can be determined with the following equation.

$$capacityThreshold = predictedData/M_f \tag{4.7}$$

where, M_f denotes the type of machine(e.g Large, Small), the *capacityThreshold* is rounded to the next higher integer and it is equal to s. The predicted number of machines depends on the data arrival rate (λ) and the service time μ in a machine.

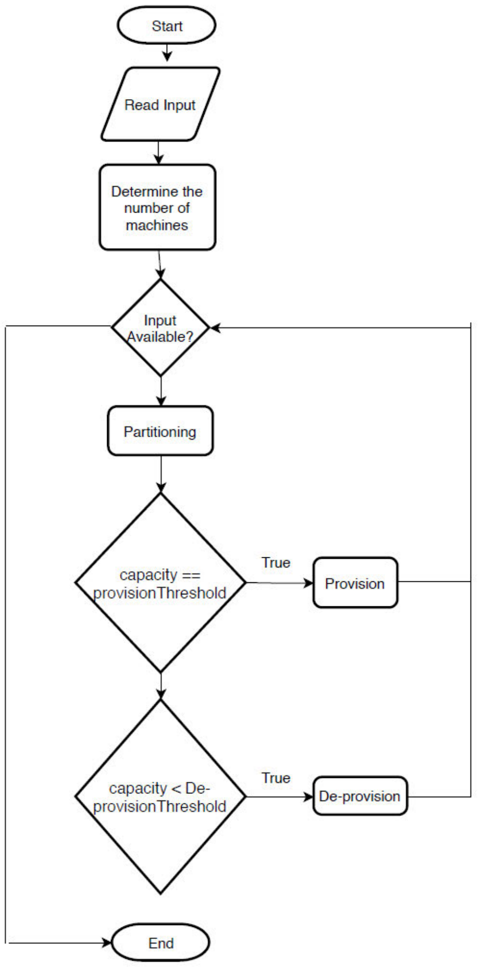

Start
Read Input
Determine the number of machines
Input Available?
Partitioning
capacity == provisionThreshold
True
Provision
capacity < De-provisionThreshold
True
De-provision
End

FIGURE 4.2: Auto-scaling work flow

4.5.1 Provision and De-provision

We use the provision or de-provision of resources with the modified Linear Deterministic Greedy (LDG) algorithm [110]. In addition to that algorithm, we use a scaling technique to provision/de-provision the type of machines based on workload in the upcoming stream. To scale up/down the resources, we check the capacity of the partition after assigning every vertex from the stream.

We check the worker availability in order to distribute the graph data and provision/de-provision the machine. The current available capacity (C_t) of worker availability needs to be checked before assigning any data to that worker machine. The worker availability is checked one unit of time in advance, to ensure the machine is ready for the next predicted stream of data.

Provision: Whenever a partition reaches near the maximum capacity($MAXCAP$)

of that partition, then our scaling algorithm, provisions from a small machine to a medium/large machine. We use a tolerance level of space which allows sometime to provision the machine from a lower capacity to a higher capacity, as some time is required to start a new machine and shut down current machine, once the newly machine is ready to serve. We use a threshold value(l) which determines launch of the machine. If a partition's capacity (C_t) reaches equal or more than the *provisionSmall* value, the auto-scaler will start launching a new machine with higher memory capacity before shutting down the current machine. The work flow of our complete auto-scaling algorithm shows in Figure 4.2

$$average = numberOfvertices/numberOfMachines \tag{4.8}$$

$$provisionSmall = (toleranceParameter * average)/100 \tag{4.9}$$

$$provisionMedium = (toleranceParameter * average)/100 \tag{4.10}$$

toleranceParameter is the tolerance percentage of the maximum capacity of a machine, which allows the initialisation of a new machine before shutting down the current machine. It helps to continue graph computation without any interruption.

De-provision: Whenever a partition is occupied less than the *deprovisionThreshold* capacity, the auto scaling algorithm de-provisions the type of machine, for example from a large to a small type of machine. We use the following equation to find the capacity of a machine.

$$deprovisionThreshold = (reduceParam * MAXCAP)/100 \tag{4.11}$$

reduceParam is the percentage level of the partition to de-provision the type of machine.

Algorithm 2 shows the overview of our complete auto-scaling algorithm, where numberOfMachine(λ) is the function to decide the number of machines based on the arrival rate. The detail of this function is explained in Section 4.5 and other parameters and variables are also explained in the same section.

Algorithm 2 Auto-scaling Algorithm

```
INPUT: $\lambda$ = arrival rate, $V$ = Vertices, $E$ = Edges, $capacity$, $threshold$,
$deprovisionThreshold$ , $currentVM$

  $k \leftarrow$ numberOfMachine($\lambda$)
  $partitionIndex \leftarrow$partition($V, E, k$)
  $i \leftarrow$ getpartitionIndex($k$);
  for ($i = 0$ to $k$ ) do
    if ($capacity_i == threshold$) then
      if $currentVM == Small$ then
        Provision the $currentVM$ to Medium for the partition $i$
      else if $currentVM == Medium$ then
        Provision the $currentVM$ to Large for the partition $i$
      end if
    end if
    if ($capacity_i < deprovisionThreshold$) then
      if $currentVM == Large$ then
        De-provision the $currentVM$ to Medium for the partition $i$
      else if $currentVM == Medium$ then
        De-provision the $currentVM$ to Small for the partition $i$
      end if
    end if
  end for
```

4.6 Experimental Setting

This study proposes an auto-scaling model based on predicted data in stream by using a queueing theory model. In this section, we discuss the evolution setup and different experimental scenarios.

The real time graph partitioner receives the graph data in a stream manner and allocates it as it arrives. We use a state-of-the-art streaming graph partitioner [110] to partition the data with a balancing strategy. While partitioning the data the auto-scaler will utilise the best of the cloud resources by using this auto-scaler algorithm.

The auto-scaler algorithm always maintains the number of Virtual Machines (VM) as per the demand of computational loads. It automatically allocates or de-allocates the VMs based on the upcoming stream and the allocated streaming data.

The auto-scaler algorithm uses a prediction model to predict the arrival data and their size. The prediction is explained in the Section 4.5.

4.6.1 Cloud Environment

We use Nectar Cloud distributed virtual machine(VM) to evaluate our proposed auto-scaling method. We used a heterogeneous cloud environment with the different types of virtual machines. Table 4.1, shows some typical VM in cloud environment.

Machine Type	**Number of Cores**	**RAM**	**Disk**	**Price/hour**
m2.large	4	12GB	110GB	$0.24
m1.medium	2	8GB	70GB	$0.12
m1.small	1	4GB	40GB	$0.0292

TABLE 4.1: Some Typical Virtual Machine

4.6.2 Dataset

The algorithm receives data in a stream manner at a certain arrival rate. The algorithm receives the data and sends it to the respective partition as it arrives. We use an existing streaming graph partitioning algorithm to perform the graph partitioning task. A number of real and synthetic datasets is used in this study. Table 4.2 shows the characteristics of several synthetic and real datasets.

Name of Dataset	$\|V\|$	$\|E\|$	**Type**
AstroPh [70]	18,772	198,110	Citation
Email-enron[70]	36,692	183,831	Communication

TABLE 4.2: Characteristics of Datasets

4.6.3 Prediction buffer

We use a prediction model to predict the data in order to determine the number of worker machines required to complete the computation. The arrived data are stored in buffer for a unit of time. We use the Poisson distribution model to distribute the data.

4.6.4 Performance Metrics

We use the following performance matrices to evaluate the performance of the auto-scaler.

Processing Cost: The resource cost to cater for the upcoming data and processing the requests from the stream.

Number of Machines: The number of machines we require to complete the computation for graph data. This is highly dependent on the data arrival rate and processing time of a request.

Type of Machines: The type of machines are utilised for a certain type of input and there size.

4.6.5 Evaluation Scenario

In this evaluation we observe how the number of machines determines the various data arrival rate for each dataset. The data arrival rate depends on the types of datasets and their characteristics because the changing rate of data makes an impact on deciding the number of machines and their utilisation. After deciding the number of machines, we use the streaming graph partitioning algorithm to partition the graph.

In considering the cost optimisation, we also observe changing the type of machine based on the arrival rate of data and the current computational load. We start the experiment with small machines, then according to the scaling threshold, the type of machine changes, for example, from small to medium, or vice versa.

We use a 95% tolerance level during provisioning; in this way if the machine capacity fills to 95%, then the provisioning threshold starts launching a new type of machine with a higher capacity (for example: small to medium) and migrates the data from the old machine.

We compare our algorithm with a well-performed existing auto-scaling algorithm [61] to evaluate the performance. The existing algorithm is well-established in resource scaling optimisation in the cloud environment.

4.7 Result Discussion

In this section, we discuss the results of machine utilisation and the optimised cost of cloud machines in a distributed system for the graph-oriented applications. Figure 4.3 and 4.4 shows the optimised number of machines as determined by the optimised scaling algorithm 2. The algorithm decides the number of machines based on the data arrival rate and the service time of a task.

4.7.1 Machine Utilisation

Utilising resources in distributed computing is an important factor to consider. As shown in Figure 4.3 to 4.6, our auto-scaling algorithm optimize the number of machine according to the load and upcoming data.

As shown in Figure 4.3 to 4.6, represent the scalability of proposed and existing algorithms. Both algorithms are able scale the machines even if the arrival rate is higher. However, the existing algorithm uses more machines even with the slower data arrival rate.

In Figure 4.3 we observe that the number of machines is used for both the algorithms is the same at the rate of 4, 7 and 9 for Email-Enron data set. On an average the proposed approach is utilising about 11% fewer machines over arrival rates. Also, the proposed ASGP method using less number of machines while data arrival is high. In some cases (at the rate of 5 and 8) proposed approach utilising more number of machines compared with the existing one while using Email-Enron data set. Figure 4.4 represents the number of machine utilisation while using AstroPh dataset. Similar to Figure 4.3, the proposed ASGP method utilise less number of machines while the arrival rate is high. The proposed approach is utilising more than 10% machines in the case of AstroPh data set on an average. The proposed and existing methods utilising the same number of machines at the rate of 9 and 12. However, for all other rate proposed approach utilising a fewer number of machines. Because each vertex arrives in the stream with different numbers of associated edges, that makes the different load of a machine over time and utilises the different number of machines over time.

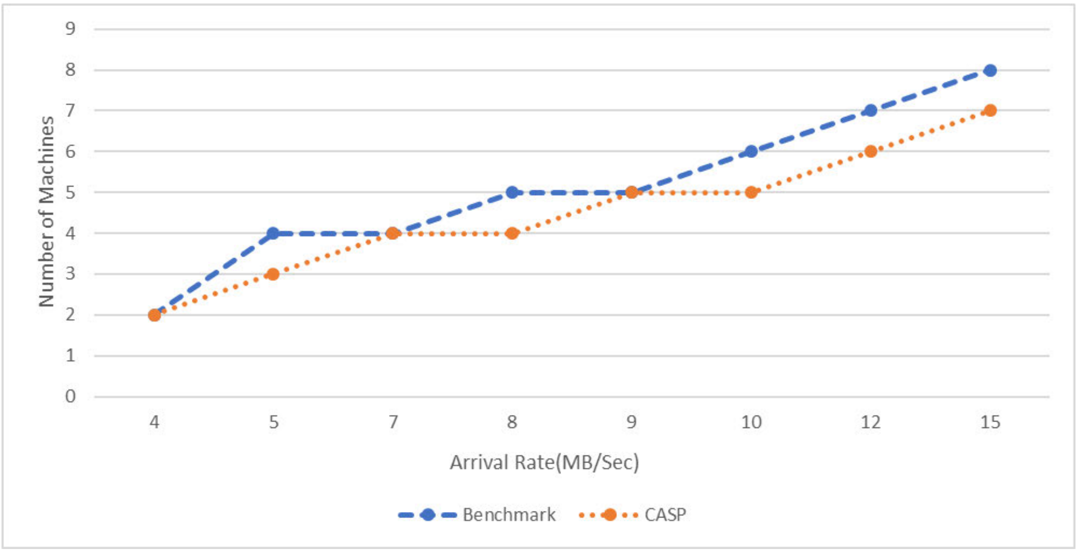


FIGURE 4.3: Number of machine for Email-Enron dataset

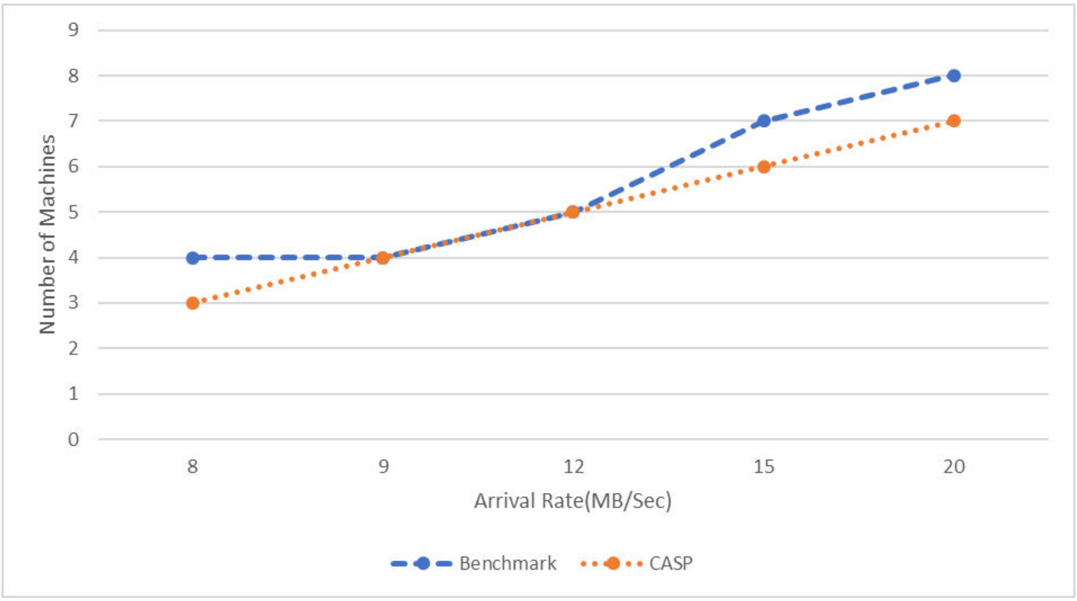


FIGURE 4.4: Number of machine for AstroPh dataset

Figure 4.5 and Figure 4.6 shows the utilisation of different flavour machines for proposed and previous algorithms while using Email-enron and AstroPh data set. For both data set proposed algorithm is using less number of medium flavour machine which is more cost-efficient. Small flavour machines are not been used in most arrival rate cases. However, while the arrival rate is increased the system is started using small machines which proves the efficiency of the proposed algorithm.

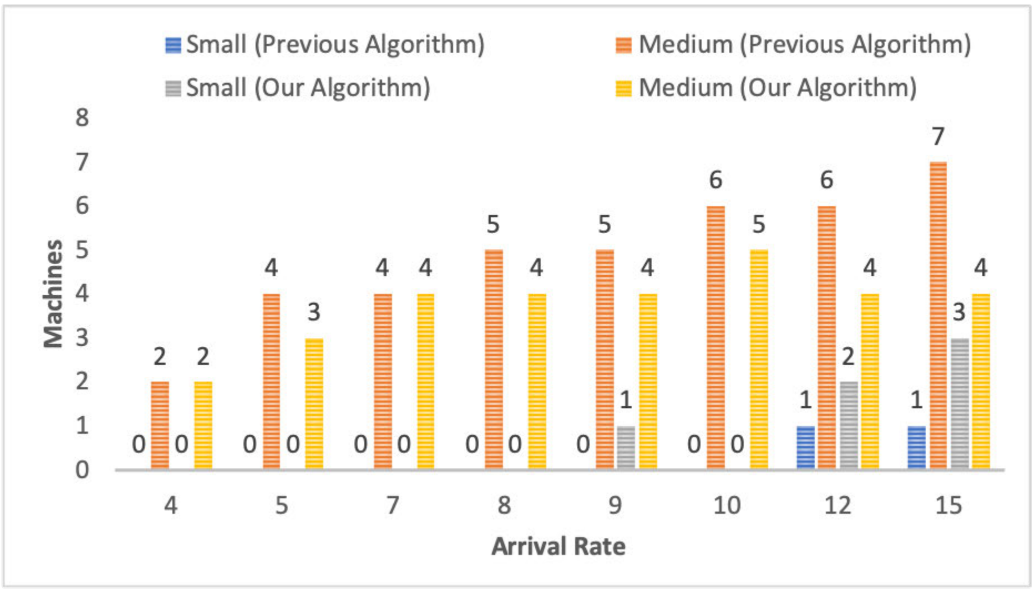


FIGURE 4.5: Different type machine for Email-enron dataset

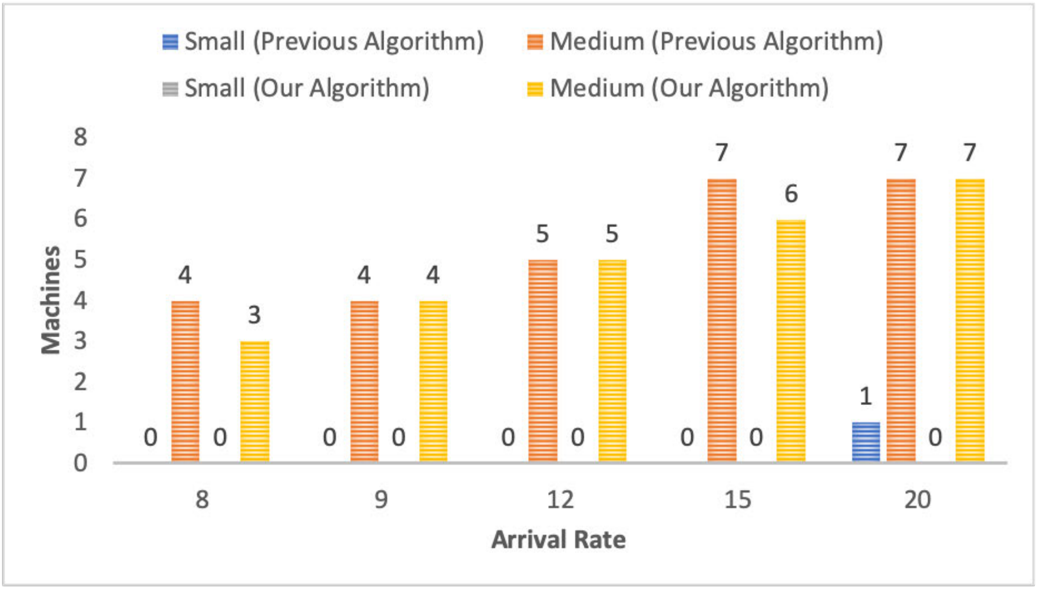


FIGURE 4.6: Different type machine for AstroPh dataset

In Figure 4.7, shows the memory utilisation of currently utilised machines that is the amount of memory being occupied by the user requests. It is seen clearly that our algorithm has better memory utilisation than the existing algorithm. The memory utilisation by our algorithm reaches close to 100% for the data arrival rate of 15 MB/sec.

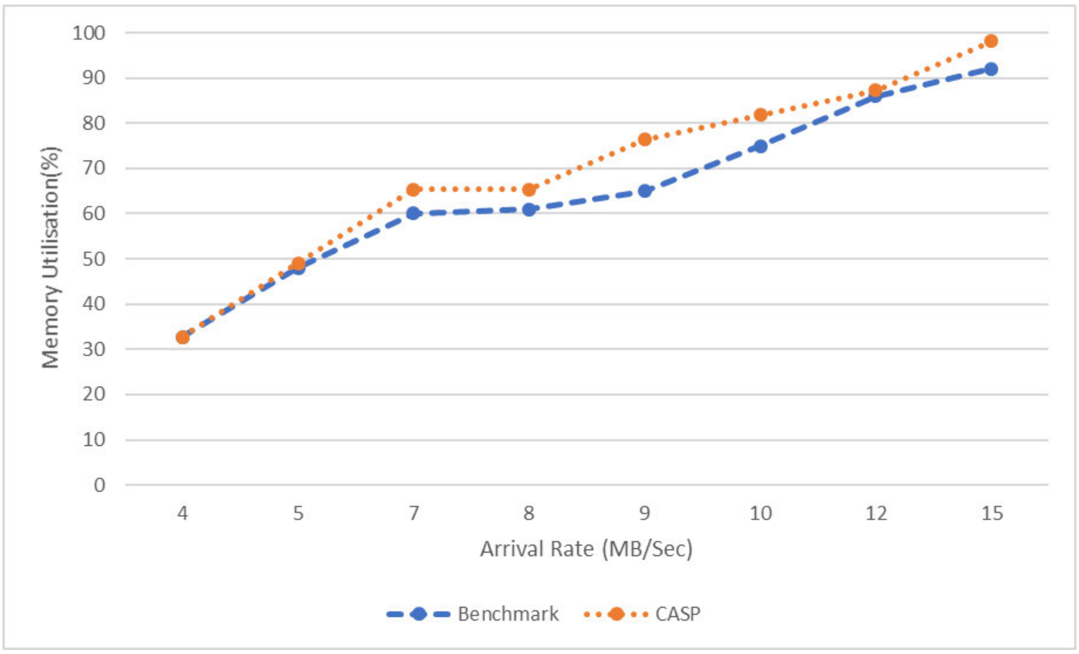


FIGURE 4.7: Memory utilisation with different arrival rate

4.7.2 Cost Optimisation

In stream graph partitioning, the selection of the type of the cloud machine has a great impact on managing the resource cost. Unused or unnecessary resources lead to a wasting resources. Figure 4.8 and 4.9 shows the comparison of two resource provisioning auto-scaling algorithms. It is clearly seen that our proposed algorithm reduces the cost of the cloud resource by selecting the type of machine as required. We compared the cost optimisation with an existing algorithm; our algorithm outperforms the existing algorithm.

In Figure 4.8 shows that at the rate of 7 and 4MB/sec the cost of resources is almost the same for both algorithms. In the case of power-law graph the data arrival rate of a vertex of the graph stream makes the difference in resource utilisation. In graph stream If a vertex arrive with high degree, then the graph requires more resources to accommodate the arrived vertex and their edges.

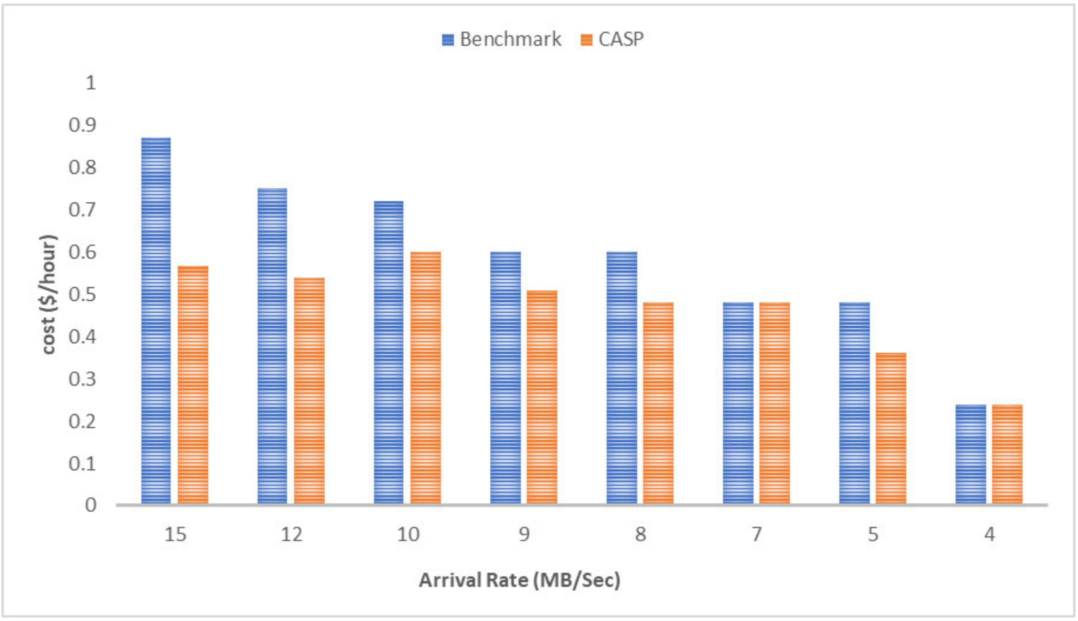


FIGURE 4.8: Cost comparison for Email-enron dataset

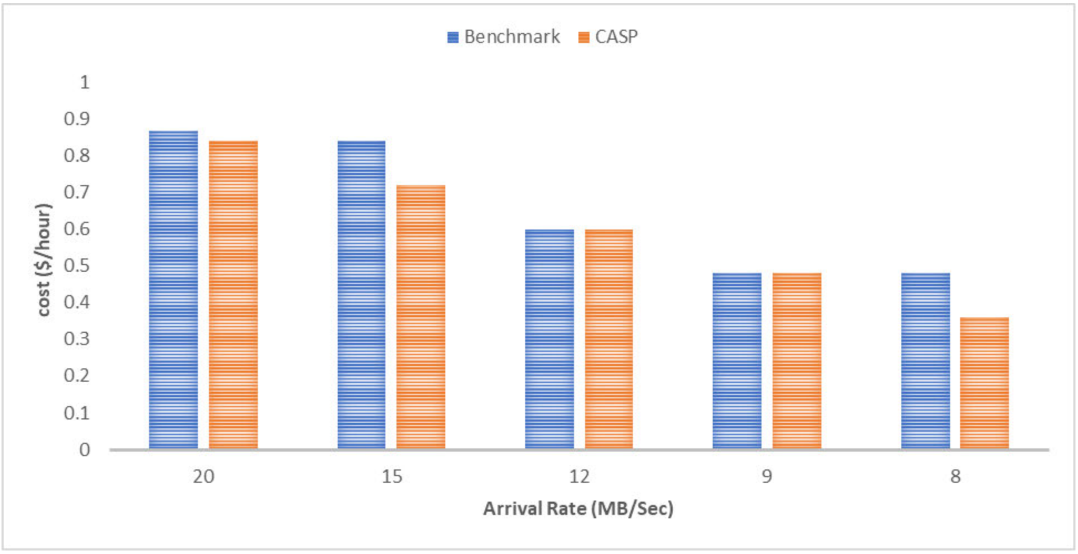


FIGURE 4.9: Cost comparison for AstroPh dataset

4.8 Related Study

One of the state-of-the-art proposed models based on the queueing model proposed [53] by using queuing theory. Based on the queuing theory in another study a number of models proposed [1] to determine the number of servers. It uses the infinity server model to decide the number of machines, based on arrival rate and service rate.

An optimal method [54] was proposed to scale the cloud resource in an automatic manner for web applications. This algorithm predicts the number of web requests automatically and allocates the number of virtual cloud machines required, according to the prediction request. Another optimal cloud resource was proposed [61] based on response time. This proposed optimal algorithm uses a Pareto trade-off between the number of used workers and the resulting response time.

An automatic cloud resource provisioning technique was proposed in [11]. This technique uses a hybrid method with a combination of reactive and proactive approaches to scale resources according to the demands of the user's request.

These proposed work did not deal with the dynamic graph dataset and predicting the dynamic graph data is different from the batch processing dataset, when the graph node is connected to each other.

4.9 Summary

This chapter studied the auto-scaling algorithm for the cloud environment for a streaming graph partitioning algorithm. Initially, the prediction algorithm determined the number of machines required to accommodate the computational load, based on the data arrival time and the service rate. The auto-scaling algorithm scales the type of machine currently being used according to computational load over time, while partitioning a graph in stream manner. We observed that our auto-scaling algorithm outperformed the existing algorithm in utilising the memory and minimising the cost of resources. We used an existing algorithm for the partitioning purpose to evaluate our scaling mechanism in order to observe the machine utilisation and cost optimisation. We have not developed any mechanism to minimise the inter-machine communication or to balance the computational load. In the next chapter we will develop a streaming partitioning algorithm, based on a sliding stream window, to minimise inter-machine communication. We also propose a parameter-based balancing strategy to keep the partitions balanced while minimising communication.

Chapter 5

WStream: Window-based Streaming Graph Partitioning

This chapter proposes a novel streaming graph partitioning algorithm based on a sliding stream window and makes the best use of vertices and edges information from the sliding window to partition a graph in real-time. A load balancing strategy is also proposed to keep load imbalance as low as possible, while minimising communication between machines.

5.1 Motivation

In recent years, the scale of graph datasets has increased to such a degree that a single machine is not capable of efficiently processing large graphs. Thereby, efficient graph partitioning is necessary for those large graph applications. Traditional graph partitioning generally loads the whole graph data into the memory before performing partitioning; this is not only a time-consuming task but it also creates memory bottlenecks. These issues of memory limitation and enormous time complexity can be resolved using stream-based graph partitioning. A streaming graph partitioning algorithm reads each vertex once and assigns that vertex to a partition accordingly. This is also called a one-pass algorithm. This chapter proposes an efficient window-based streaming graph partitioning algorithm called WStream. The WStream algorithm is an edge-cut partitioning algorithm which distributes a vertex among the partitions. Our results suggest that the WStream algorithm is able to partition large graph data efficiently, while keeping the load balanced across different partitions, and communication to a minimum. Evaluation results with real workloads also prove the effectiveness of our proposed algorithm, and it achieves a significant reduction in load imbalance and edge-cut with different ranges of dataset.

5.2 Introduction

The scale of graph data is becoming larger and the trend will continue to grow rapidly with the emergence of different applications (for example, web-graphs, social networks, road networks and biological networks) that deal with massive interconnectivity [133]. Consequently, most real-world applications require distributed computation due to the emergence of these large graphs. To complete the distributed computation of any application, we need to partition an entire graph across machines in a cluster for faster localised processing. This is a vital process in distributing the load.

Graph partitioning cuts a graph into several disjoint sub-graphs with the aim of minimising the edges between these sub-graphs while retaining almost the same number of vertices in every partition. Imbalance among computational load in a distributed environment produces inefficient applications. Besides minimisation of communication, the load balancing also must be considered in graph partitioning. These two aspects make graph partitioning an essential pre-processing task for efficient computational speed in different real-world graph applications.

In traditional graph partitioning[72, 10], the entire graph must be loaded into memory for partitioning and processing. Potentially, this requires huge storage/memory capacities due to the large-scale of the data which, in some cases, may increase continuously over time. Consequently, the traditional partitioning algorithms require high computational costs and memory for the partitioning tasks. In addition, this also affects the graph data processing for different applications (for example PageRank, Shortest Path). These are the main motivations for this study which proposes a streaming graph partitioning algorithm to partition large graphs efficiently. Streaming graph partitioning is a new variant of the graph partitioning problem, which aims to deal with time-evolving graph datasets. The streaming partitioning technique is also known as a single pass algorithm, as the data can be seen only once in this partitioning algorithm. The streaming graph partitioning algorithm was introduced by Stanton [110]. It aims to provide efficient graph processing by reducing memory bottlenecks, allocating the graph data as it arrives rather than loading the entire graph into memory. It was a welcomed approach, as graph datasets are rapidly growing day by day. As a result, streaming graph-partitioning is now playing a vital role in overcoming the issues that traditional partitioning cannot.

Several studies have been conducted on streaming graph partitioning [110], [109], [7], [113], [4]. However, many of the studies use a synthetic dataset to evaluate algorithm performance rather than real-world graphs. These studies also assume that the graphs are already localised on the disk and the stream of data will be in a particular order (for example, Breadth First Search, Depth First Search). In real-world scenarios, graph data do not come in a certain order. Consequently, there is no scope to use any particular stream ordering in real-world graph application scenarios.

In this chapter, we propose a window-based streaming graph partitioning technique to obtain better partitioning performance and to reduce edge-cut, whilst keeping load imbalance as low as possible. The key idea of this algorithm is that the window-based stream of graph data has more information on vertex allocation because the usual single-pass graph partitioning only uses the presence of a vertex to determine the partition for that vertex. The window-based algorithm does not consider any stream order when receiving the stream of data input. We argue that this technique improves the partitioning performance with regard to the following aspects: i) It balances computational loads among machines. ii) It addresses scalability, as it accepts any range of datasets. iii) It reduces the communication between machines (by reducing the number of edge-cuts). The contributions of this chapter are as follows:

- A window-based streaming graph partitioning technique that aims to reduce the number of edge-cuts by maintaining a balanced partition.
- A streaming window that helps to obtain more information associated with a vertex before a vertex is assigned to a partition.

- An algorithm which checks the number of edges of a buffered vertex in the window and helps to achieve better partitioning performance when assigning the vertex.

5.3 Related Work

Recently, there has been considerable interest in the design of an algorithm and a framework to handle massive graph data in a streaming manner. Steaming graph-data can be partitioned into a cluster of nodes; the graph access pattern could be done via online or offline processing. Streaming graph partitioning is very efficient because the graph loader or partitioner does the partitioning task while receiving the graph data in a streaming manner. A near-optimal traditional graph partitioning algorithm called METIS was proposed in the early graph-partitioning era [62]. METIS is the de facto standard for near-optimal partitioning in distributed graph partitioning. METIS can reduce the communication costs among distributed machines despite having a lengthy processing time for small graphs. However, METIS is not suitable for processing medium or large graph datasets [62].

Graph partitioning can be categorised into two types: Vertex-Cut-based and Edge-Cut-based. In Table 5.1, we summarise and compare the most recent stream-based partitioning algorithms. In the following sections, we provide details of related work.

Algorithm	Vertex-cut	Edge-cut	Distributed	Window-based
Linear Deterministic Greedy(LDG)[110]	No	Yes	Yes	No
Natural Graph Factorization[7]	Yes	No	Yes	No
LOOM[32]	No	Yes	Yes	No
STINGER[88]	No	Yes	No	No
Planted Partition[112]	No	Yes	No	No
HDRF[83]	Yes	No	Yes	No
HoVerCut[91]	Yes	No	Yes	Yes
Vertex Migration[4]	No	Yes	Yes	No

TABLE 5.1: Summary of stream based graph partitioning

5.3.1 Vertex-cut Partitioning

A scalable streaming partitioning approach was proposed by Wang and Chiu [123] with the aim of achieving a low complexity system. This partitioning technique aims to reduce the number of edges between partitions, and consequently reduces the communication cost of query processing. A streaming vertex-cut partitioning algorithm, High Degree Replicated First (HDRF), was proposed by Petroni et al. [83] to utilise the vertex characteristics. The study used a greedy vertex-cut approach, in which the high-degree (number of edges of a vertex) vertices replicate first, in order to minimise and avoid unnecessary vertex replication. This algorithm achieved a significant improvement in stream-based partitioning compared with previous algorithms [71]. HDRF achieves nearly twice the speed of traditional greedy placement and is almost three times faster than using a constrained solution. Sajjad et al. proposed a scalable streaming graph partitioning technique called HoVerCut [91], which provided horizontal and vertical scalability for the graph partitioning system. HoVerCut used multi-threading with a windowing technique to share incoming edges among the threads. However, in that the window that contains the edges does not update over time. This may create performance degradation and it is not suitable for dynamic datasets.

Real-world graphs, for example, social networks, typically follow a power-law degree distribution. Partitioning power-law graphs is very difficult. PowerGraph [38] aims to reduce inter-partition communication by computing edges over vertices of power-law graphs. It follows the GAS (Gather, Apply, and Scatter) model and uses a vertex-cut partitioning technique. It distributes replicas of vertices into multiple machines to parallelise the computation.

Another variant of PowerGraph streaming partitioning was proposed by Xie et al [126] called S-PowerGraph. S-PowerGraph also used vertex-cut partitioning. This method is suitable for partitioning skewed natural graphs and was found to outperform algorithms in previous studies with regard to an acceptable imbalance factor.

5.3.2 Edge-cut Partitioning

A distributed vertex swapping technique called Ja-be-ja [86] was proposed by Rahiman et al. this vertices swapping technique made uses to reduce the communication. Ja-be-ja was built based on a local search and Simulated Annealing(SA) method. The SA method uses the statistical mechanism which is not suitable for the sparse network [125]

Stanton [110] proposed a few heuristics for partitioning a large-scale graph in a streaming manner. Linear Deterministic Greedy (LDG) was the best performing heuristic of these. This algorithm is a greedy heuristic, which is linear. It has a central graph loader, which loads and distributes data among the available workers. The heuristic assigns a vertex to the partition with which it shares the most edges. The algorithm was evaluated using 21 different static datasets and up to 16 partitions. It makes heuristics scale with the size and number of graph partitions. Based on PageRank computations, the method yielded a significant speed up achievement for large social networks by 18%-39% when compared with Spark [131]. However, there are drawbacks in this study which we have addressed in our study. LDG receives the input data in a certain order, which is not suitable for any real-world streaming graph application whereas, the WStream algorithm receives the graph input sequentially, as it arrives regardless of any particular order. Moreover, the LDG algorithm uses the entire subgraph information from all vertices previously partitioned. They also used a distributed look up table to access the graph information. Moreover, we used a stream window which creates more opportunities to get better partitioning results by using a greedy strategy.

The LDG algorithm is a well-established streaming graph partitioning algorithm and is a state-of-the-art one-pass edge-cut partitioning algorithm. Therefore, in this study, we compared our one-pass edge-cut partitioning algorithm with the LDG algorithm [110].

We propose a window-based streaming processing algorithm; the window can contain more information about a candidate vertex which is ready to be assigned to a particular partition. While partitioning graph data from a window, the first vertex and its adjacent vertices of the window are assigned to a partition by the algorithm. The first entry checks for the presence of any connected vertices in the current window as well as the vertex with the most edges from the partitioned data. This stream window helps in deciding an appropriate partition to assign for the first entry and its associated vertices to a respective partition of the candidate vertex with the most edges, or any of the connected vertices to reduce communication. This technique achieves significant improvement in reducing edge-cut because the decision to assign a vertex has some impact on its future connected vertex from the window. To the best of our knowledge, this technique has not yet been applied in any studies.

5.4 Proposed Algorithm

5.4.1 Preliminaries

We consider an undirected graph G, with a set of edges E and vertices V, such that $G = (V, E)$. A balanced k-way partitioning divides the graph into almost equal subsets. The graph partitioning algorithm uses a balancing constraint to keep all the partitions balanced. The balancing constraint can be defined by Equation 5.1:

$$\forall_i \in \{1..k\} : |V_i| \leq L_{max} := (1+\alpha)\lceil |V|/k \rceil \tag{5.1}$$

where, α is the unbalanced parameter, and is a non-negative real number. The vertex v is adjacent to vertex u given there is an edge $\{u, v\} \in E$. If vertex v and vertex u reside in different partitions, this is called the *cut edge*. Thus, $E_{ij} := \{\{u, v\} \in E : u \in V_i, v \in V_j\}$ is the set of edge-cuts between partitions. Edge-cut graph partitioning always aims at reducing this cut.

5.4.2 System Architecture

In our system, we used master machines which are responsible for reading input and assigning the vertices to the clients, as such partitioning algorithm resides in the master machine. We used a Stream Generator which creates the stream data after receiving the input and then forms a stream window. We used a distributed meta data file which has been used to store the information of the vertices which are already seen in the stream. This information was used to assign the future vertices from the stream of data. Figure 5.1 shows the architecture of the system.

- **Master Machine:** The master machine receives the input graph data from the input file. In the master machine the Stream Generator generates the stream data and maintains the stream window before assigning each vertex to the respective partition. It also stores the partitioned vertex information and later uses this for future vertex partition.

- **Partition:** Each partition, also known as a worker machine, communicates with the master machine to receive the assigned vertex from the master machine. The worker machines also communicate with each other to maintain the computation of a domain application.

5.4.3 The Streaming Model and Window

We consider that the graph data comes in a stream of tuples $V < vertex; edges >$. The proposed algorithm utilises a sliding stream window of size W. We define two different vertices in the stream window: 1) ***Candidate vertex*** is the one, which is the front of the stream window and is available for partitioning. 2) Neighbors of the candidate vertex in the window are known as a ***Buffered vertex*** in this study. As shown in Figure 5.2, $V1$ is the candidate vertex that resides at the front of the window which contains three more vertices. Vertices $V3$ and $V4$ are the neighbours of candidate vertex $V1$. Thus, these two vertices are defined as a buffered vertex in a stream window.

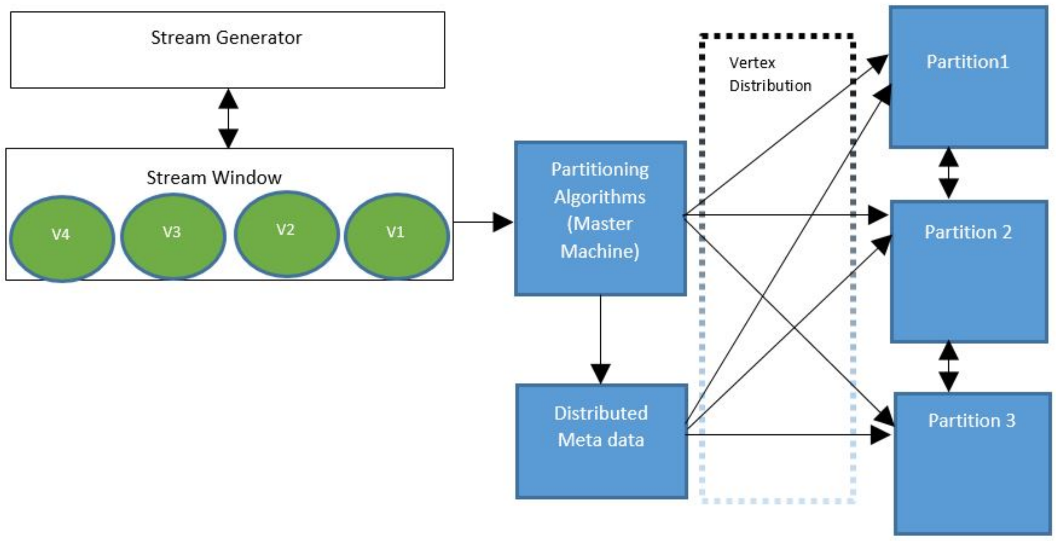


FIGURE 5.1: System Architecture

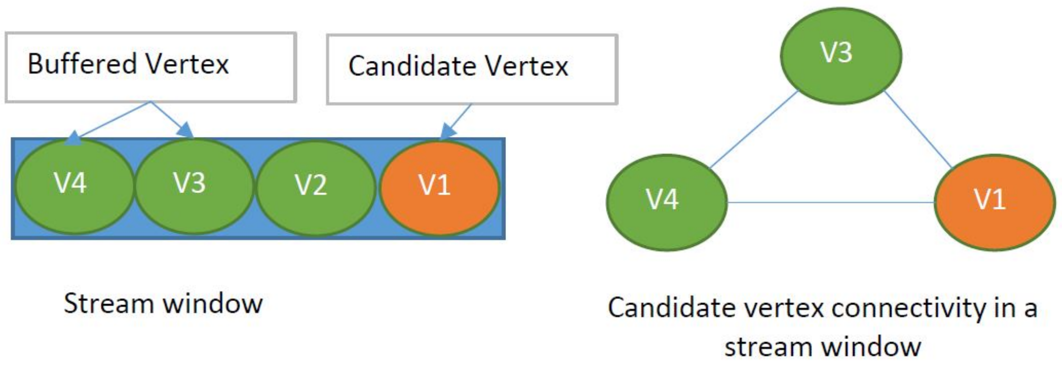


FIGURE 5.2: Candidate and Buffered Vertex in a Stream Window

When a candidate vertex is allocated to a partition, it leaves a space for another vertex to come into the stream window to maintain the window size. The stream window contains more than one vertex such that it gives more information about a candidate vertex and other connected vertices of the candidate vertex. Consequently, this window-based partitioning produces better partitioning quality. The size of the window is $W > 1$ and the size of the stream window depends on the graph structure and the type of graph. In this study, the minimum and maximum window sizes are 100 and 800, respectively.

Stream order is another aspect to consider while performing streaming graph partitioning, as it has a major influence on the performance of graph partitioning. The input order of a graph makes a significant difference to the performance of a partitioning method. In a real-world graph with streaming settings, the order of a stream is not predictable. In this study, we consider a uniformly random order while receiving the graph input to the stream window. Algorithm 3 presents the pseudocode for our proposed WStream algorithm.

Algorithm 3 WStream Algorithm

```
if all the partitions are empty then
    assign $V$ randomly(uniform)
else
    if $loadImbalance \geq B$ then
        perform greedy strategy except for the partition with the highest load
    else
        perform greedy strategy
    end if
end if
```

Algorithm 4 Greedy Strategy

```
for $i = 0$ to $k$ do
    if $(|P_k \cap E(V_c)| \geq |P_k \cap E(V_b)|)$ then
        $k \leftarrow V_c$
    else
        if $(|P_k \cap E(V_c)| == |P_k \cap E(V_b)|)$ then
            $V_c$ to a partition randomly
        else
            $V_c$ to a partition that has minimum load
        end if
    end if
end for
```

5.4.4 WStream Algorithm

The algorithm starts with three inputs, a number of vertices, their associated edges, and a balancing parameter. We also specified the number of partitions. The algorithm finds the total number of vertices of each partition and identifies the partition with the maximum number of vertices. The algorithm must keep track of differences among partitions to keep them balanced. Algorithm 3 shows the pseudocode for the WStream algorithm.

The balancing parameter p checks the level of imbalance of the partitions with each other. We used the parameter p=[0, α] and a range of α values to check the balancing performance of the algorithm. We observed that a higher α value reduces the number of edge-cuts, and the load difference among partitions did not exceed the parameter α. In the case of α=0 the partitions were perfectly balanced. The balancing technique checks the load of the partitions after assigning a vertex to a partition. This is obtained by finding the load differences between partitions. Finding load differences means finding the comparison between the total numbers of allocated vertices among all the partitions.

During partitioning, if the difference between the load at any partition and the maximum load exceeds the value of α, the algorithm decides to assign vertices to other partitions using Greedy Strategy except for the partition with maximum load. After completing the partitioning task, we calculate the load imbalance by calculating the standard deviation of the number of vertices in each partition.

This algorithm checks the balancing parameter after assigning each vertex to any partition. While performing the partitioning task, whichever partition exceeds the value of the parameter, the partitioning algorithm stops sending vertices to that partition and applies a greedy strategy (discussed in the next section) to assign the following vertices to other partitions. The following section discusses in detail the mechanism of the vertex assignation technique, using the Greedy Strategy for partition decisions.

5.4.5 Greedy Strategy

The WStream algorithm exploits the Greedy Strategy to find the partition, which has the most edges of the candidate vertex or the buffered vertex from the window. The master machine is responsible for the partitioning task and distributes vertices to the clients. The master machine also stores the summary of vertex information to be used for assigning future vertices to an appropriate partition. At the beginning of the partitioning task, the algorithm assigns the candidate vertex to a partition by using a uniform random distribution. Algorithm 2 shows the pseudocode of this window-based Greedy technique.

The major aim of this method is to find a partition for the candidate vertex and buffered vertex in the window. To decide this, neighbours of the candidate vertex (based on the graph summary) are also taken into account. The algorithm assigns the candidate vertex along with its neighbours to the most weighted partition. This technique helps minimise the communication between partitions. The greedy technique tends to assign the vertices where they or their associated connections have the most connections. However, if the algorithm finds the same number of edges from two or more partitions then the algorithm assigns that candidate vertex and its connected vertex from the window to the partition, which has fewer loads among the tied partitions. In any case, if the algorithm does not find any edges for candidate vertices and buffer vertices from a partition, it decides to assign the candidate vertex randomly in a uniform manner to any of the partitions.

$$\arg\max_{k \in P} \{|E(V_c) \cap P_k|\} \tag{5.2}$$

$$\arg\max_{k \in P, b \in B} \{|E(V_b) \cap P_k|\} \tag{5.3}$$

Equations 5.2 and 5.3 show the formula to determine the partition which has the maximum number of edges of the candidate vertex and buffered vertex, respectively. $E(V_c)$ is the number of edges of the candidate vertex, $E(V_b)$ is the number of edges of the buffered vertex, P_k is the set of vertices of the kth partition, and $b \in B$ is the set of buffered vertices of the stream window.

5.5 Performance Evaluation

This section discusses the evaluation criteria, dataset, performance metrics and experimental environment used in this study.

5.5.1 Experimental Settings

We implemented our proposed WStream algorithm and the LDG algorithm [110] by using JAVA programming language. We then compared these two algorithms by using two synthetic and five real-world graph datasets. Three performance metrics are used in this comparison. We also use the METIS graph partitioning algorithm to compare the partitioning performance. We consider different experimental scenarios to evaluate the WStream algorithm. We run our experiments on the Linux operating system using virtual machines from the Nectar cloud [2] service. The device has 12 GB of RAM and 4 VCPUs.

5.5.2 Dataset

We evaluated our partitioning performance using several static undirected real and synthetic graph datasets from different graph data archives. Table 5.2 summarises the basic characteristics of the datasets used in our experiments. We chose different sizes and a variety of graphs to observe the partitioning performance of the algorithm in the context of scalability. Different structure and volumes of data make differences in partitioning behaviour and performance. For example, the degree of adjacent vertices of social networks data is more positively correlated than in another dataset [79].

Name of Dataset	$\|V\|$	$\|E\|$	Type	Source
3elt (Synthetic)	4200	13722	Finite-element mashes	[121]
GrQc	5242	14496	Collaboration Network	[70]
Wiki-vote	7,115	99,291	Social	[70]
4elt (Synthetic)	15,606	45,878	Finite-element mashes	[121]
AstroPh	18,772	198,110	Citation	[70]
Email-enron	36,692	183,831	Communication	[70]
Twitter	81,306	1,768,149	Social	[70]
com-DBLP	317,080	1,049,866	Citation	[70]

TABLE 5.2: Characteristics of dataset

5.5.3 Performance Metrics

We observed the performance of our proposed algorithm using the following performance metrics: i) fraction of edge-cut; ii) load imbalance; iii) execution time. We observed the number of external connections of a vertex from one partition to another partition. We calculated the fraction of the edge-cut using Equation 5.4:

$$edgecutratio = \frac{|E(u,v)|}{|E|} \tag{5.4}$$

where, $|E|$ is the total number of edges of a graph and $|E(u,v)|$ the total number of edges between u and v across partitions. We calculated the standard deviation of the number of vertices in a partition to observe the imbalance from one partition to another. Equation 5.5 was used to calculate the load imbalance:

$$loadImbalance = \sqrt{\frac{\Sigma|v-\bar{v}|^2}{n}} \tag{5.5}$$

where, v is the total number of vertices of a partition and n is the total number of partitions.

We measured the execution time from the start of partitioning until the end of partitioning. Input receiving time is also calculated during this execution time, as the streaming partitioning algorithm executes as stream data arrives.

5.5.4 Evaluation Scenario

We used an unrestricted stream model to receive our graph input. In this model, the algorithm accepts a static graph input in sequential order and in a one-pass manner.

We used a different combination of partitions (e.g. 2, 4, 8, and 16) with different ranges of datasets.

To evaluate the effectiveness of our algorithm, we evaluated how the variation of stream window sizes affects performance; we conducted experiments with the following window sizes: 100, 200, 300, 400, 500, 600, 700 and 800. The size of a window refers to the number of vertices in a window with its associated edges. Figure 1 depicts the impact of window sizes on the WStream algorithm performance.

Variations in the balancing factor with different balancing parameters were also evaluated in this study. We conducted experiments with different balancing parameters such as 50, 100, and 150.

In this evaluation, we also compared the performance of our algorithm with the best streaming graph partitioning heuristic currently in the field, the LDG algorithm [110]. This algorithm employs a state-of-the-art streaming graph partitioning technique. A few studies have been undertaken on streaming graph partitioning. However, most have not used one-pass streaming in their implementation. The LDG algorithm is similar to our method and it is also one of the best streaming graph partitioning techniques in this field. This is the main reason for the selection of this algorithm for comparison purposes.

METIS is one of the most powerful algorithms for offline graph partitioning and it is a near-optimal algorithm. Therefore, we also compared our WStream algorithm with METIS to observe the edge-cut performance. That gives us an insight as to how close WStream is to an optimal algorithm regarding the minimisation of the number of edge-cuts.

5.6 Results Analysis

This section discusses the results of the different datasets using different scenarios, such as a different number of stream windows, a different number of partitions and different balancing parameters. We also compare our evaluated results with a state-of-the-art algorithm [110] for streaming graph partitioning.

5.6.1 Impact of Stream Window Sizes

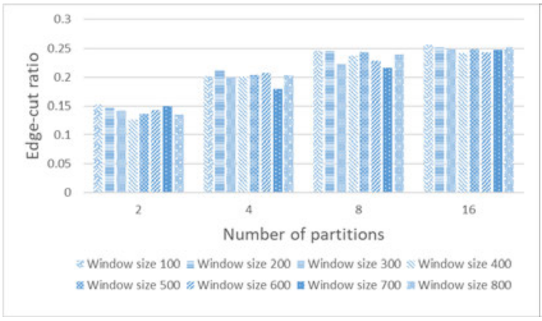
The WStream algorithm generates a stream window while receiving a stream of graph data. In this evaluation, we use a different number of window sizes to observe the partitioning performance (for example, a reduction in communication cost) of the algorithm.

Figure 5.3 depicts the edge-cut ratio for different partitions with varying window sizes. The WStream algorithm aims to utilise the stream window to provide efficient partitioning performance. The results show the impact of different window sizes on the partitioning performance of the WStream algorithm. It is expected that the edge-cut decreases with a larger window size, as a larger window contains more information of a candidate vertex. However, as shown in Figure 5.3, in some cases, a higher edge-cut ratio is obtained for the larger stream window sizes compared with the smaller window sizes. This is because our balancing parameter checks the imbalance of all the partitions every time stream data is received before allocating them to a partition. To keep the partitions balanced according to our balancing parameter, the algorithm defies the Greedy strategy and allocates the vertices to the partition with the minimum load. In this case, the WStream algorithm does not obtain the expected edge-cut reduction.

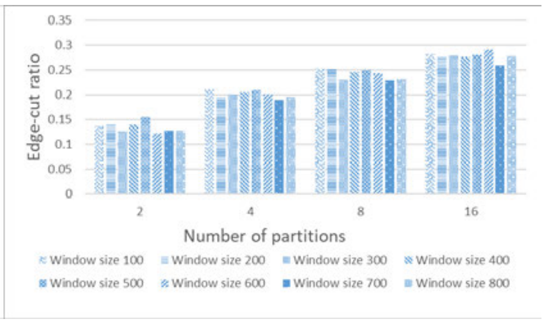
Figure 5.3(g) demonstrates the edge-cut ratio of the Twitter dataset with a different number of stream window sizes and different partitions. We observe that the 4 partitions setting performs well and achieves the expected outcome for the window-based streaming algorithm, except for window size 200 and window size 600, which obtains a slightly higher edge-cut; otherwise it reduces the edge-cut as we increase the window size.

Figure 5.3(e) shows that the WStream algorithm performs well for the Email-enron dataset for 2 partitions and 16 partitions in reducing the number of edge-cuts as the window size increases.

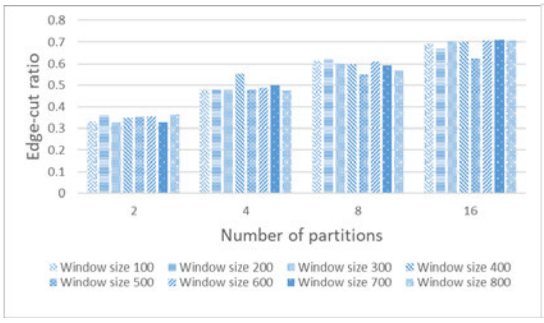
As depicted in Figure 5.3(g) and 5.3(h) for com-DBLP and Twitter dataset which contains billion edges. WStream performs well in reducing the edge-cut as the window size increases. We observe that the 16 partitions com-DBLP dataset produces better performance than other partition settings and the Twitter datasets performs



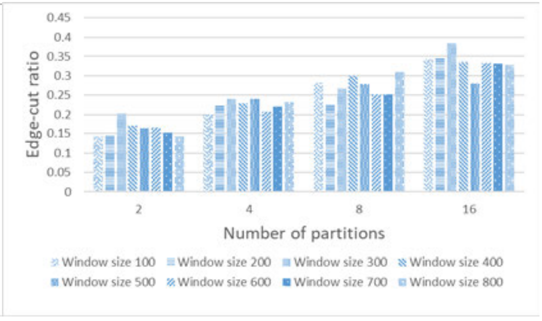
(A) 3elt Dataset



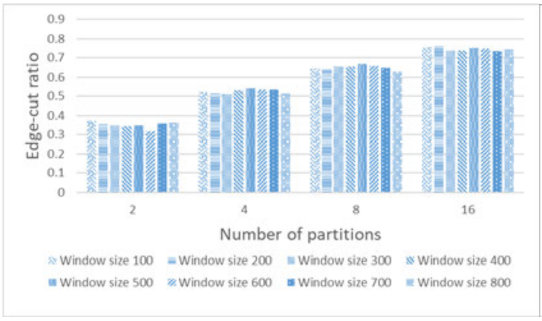
(B) 4elt Dataset



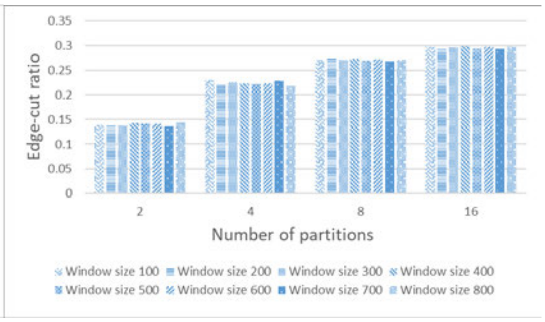
(C) AstroPh Dataset



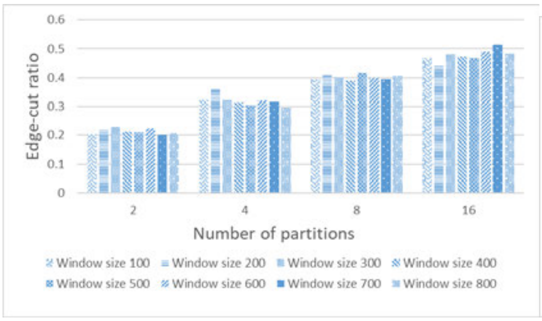
(D) GrQc Dataset



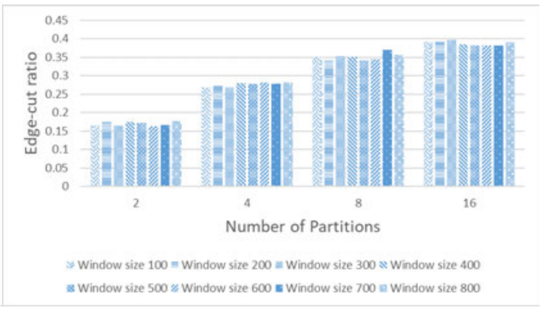
(E) Email-enron Dataset



(F) Wiki-vote Dataset



(G) Twitter Dataset



(H) com-DBLP Dataset

FIGURE 5.3: Impact of different window sizes on different datasets

well for the 2 partition settings.

5.6.2 Impact of Balancing Factor

Figure 5.4 shows the impact of the balancing parameter in reducing the communication cost of our WStream algorithm for the Twitter dataset. It can clearly be seen that, as expected, a larger balancing parameter reduces the inter-partition communication across partitions, because the smaller balancing parameter provides a lesser load imbalance. To make partitions more balanced in a cluster, vertices have to be distributed to the machines, thus resulting in increased communication cost. In the WStream algorithm, the applied balancing parameter provides greater reduction in communication cost. The more partitions are balanced the more communication is created. As depicted in Figure 5.4, the balancing parameter at the value of 50 produces about 17%- 19% more edge-cuts compared with the value of 150 for all partition settings. The proposed WStream algorithm checks the workload of each partition prior to assigning a vertex to a partition, where the difference in workload among partitions never exceeds the value of balancing parameter. Consequently, it keeps the imbalance as low as possible across partitions.

Figure 5.5 shows the load imbalance in the Twitter dataset with different balancing parameters for partitions 2, 4, 8, and 16. It is clearly seen that the smaller balancing parameter has a lower imbalance for any number of partition settings.

5.6.3 Performance Comparison

Edge-cut comparison

Figure 5.6 shows the comparison of the fraction of the edge-cut between the LDG, METIS and the WStream algorithms for different scales of datasets. Our algorithm outperforms the LDG algorithm in reducing the edge-cut for all datasets except for the Wiki-Vote dataset. Edge-cut ratio differences between METIS and WStream are quite promising as METIS is a static and near optimal graph partitioning algorithm.

In this evaluation, we use a different number of partitions (for example, 2, 4, 8, and 16) to test the partitioning performance. As expected, the communication cost increased for a larger number of partitions, as shown in Figure 5.6, for all the algorithms. We compare the edge cut of different datasets. In this comparison, we use a window size of 100 for the WStream algorithm. A window size of 100 provides the worst performance of the WSteam algorithm as expected. That is why, we choose this worst performance of WStream algorithm to make a comparison with LDG algorithm. From Figure 5.6, it is clearly seen that the WStream algorithm outperforms the LDG algorithm in reducing the edge-cut for the 3elt dataset. The result indicates that the WStream algorithm is able to reduce the edge-cut ratio by 56% for 16 partitions for the 3elt dataset. As demonstrated in Figure 5.6(a), our WStream algorithm shows significant improvement in reducing edge-cuts by about 40%-56% for all partition settings. The WStream algorithm has significant improvement in reducing edge-cut ratio compared with LDG algorithm for the large scale dataset like com-DBLP and Twitter. It is noticed that, it reduces 66%-75% edge-cut ratio for these two datasets.

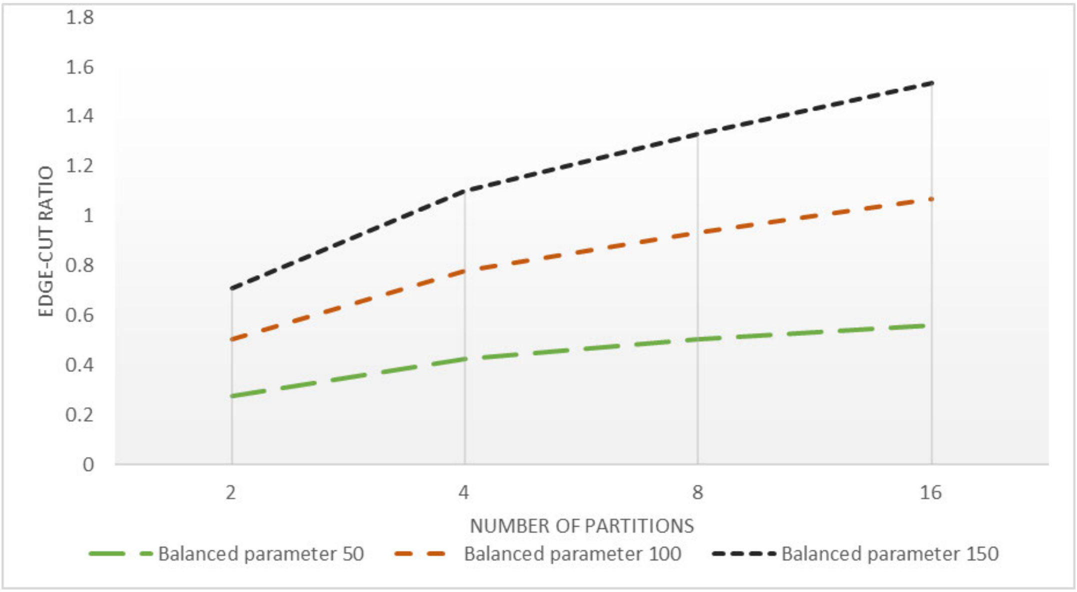


FIGURE 5.4: Variation of Balancing Parameter for Twitter dataset

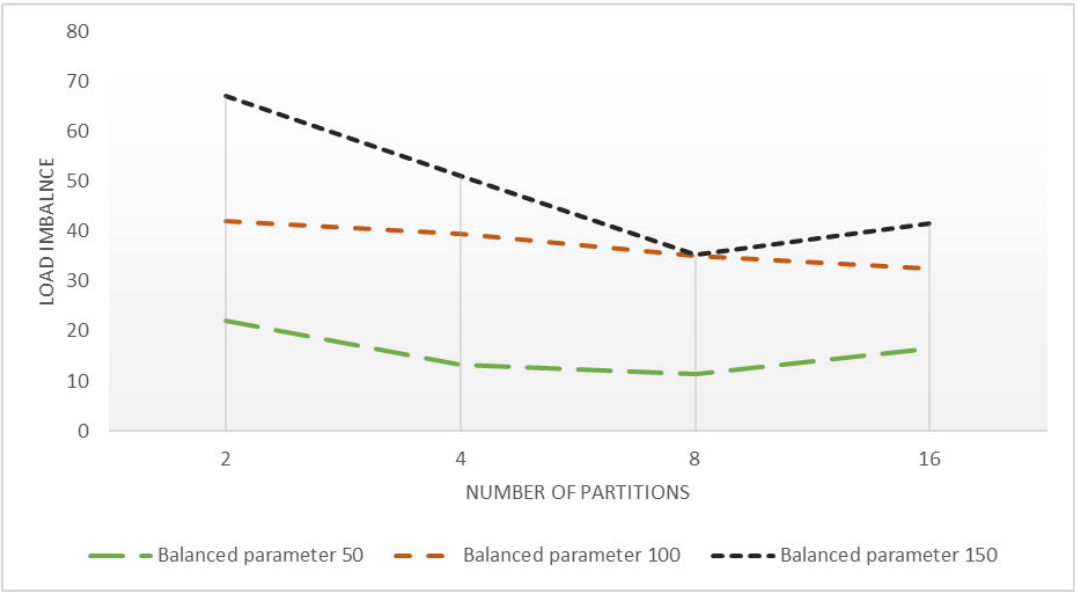
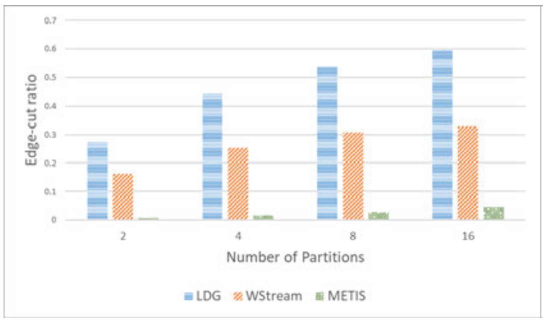


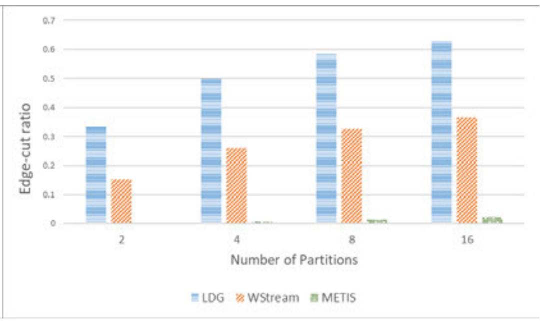
FIGURE 5.5: Load Imbalance for Twitter dataset

Our WStream algorithm is compatible with datasets of different scales and outperformed the LDG algorithm in most cases. However, there is slight performance degradation of the WStream algorithm for Wiki-Vote dataset. Figure 6(f) depicts the edge-cut comparison between the WStream and LDG algorithms for the Wiki-Vote dataset. The results indicate that the performance of the WStream algorithm drops slightly compared with the LDG algorithm for this dataset. This is due to the behaviour of the dataset. This dataset is structured with a higher degree of vertex distribution compared with other datasets in this study. While partitioning a graph, the algorithm tends to make partitions as balanced as possible according to the balancing parameter. Consequently, the adjacent vertices of a vertex might have been allocated to other partitions. Thus, this causes more edge-cuts in the Wiki-Vote dataset.

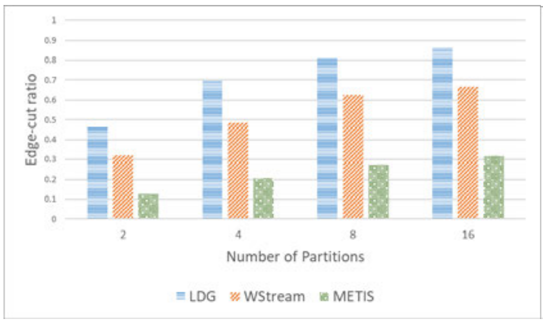
Figure 5.6 also depicts the edge-cut performance of the METIS algorithm along with WStream and LDG algorithms. As expected, METIS outperforms the LDG and WStream algorithms because it is an offline graph-partitioning algorithm in which all the information about the graph is known prior to partitioning. METIS provides optimal partitioning but for the AstroPh, Email-enron and Wiki-vote datasets overall edge-cut difference is 20%-25% between METIS and WStream algorithms. This is a quite promising performance of the WStream algorithm when compared with a static graph partitioning algorithm. The worst performance was for WStream for



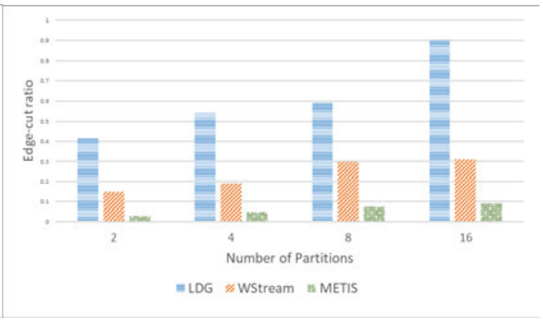
(A) 3elt Dataset



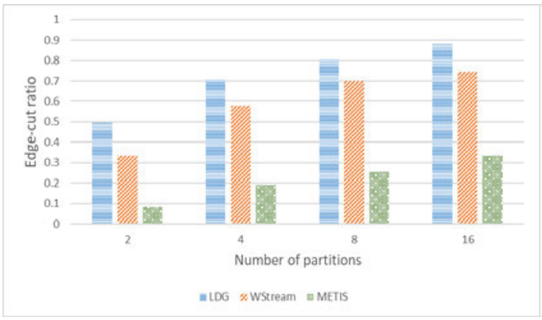
(B) 4elt Dataset



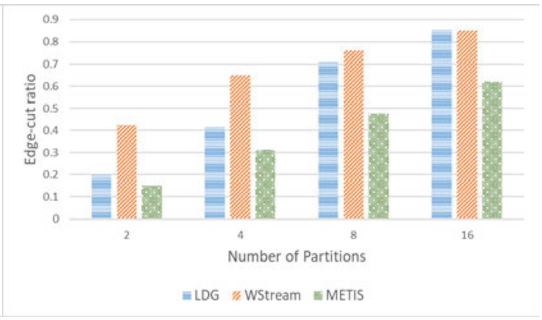
(C) AstroPh Dataset



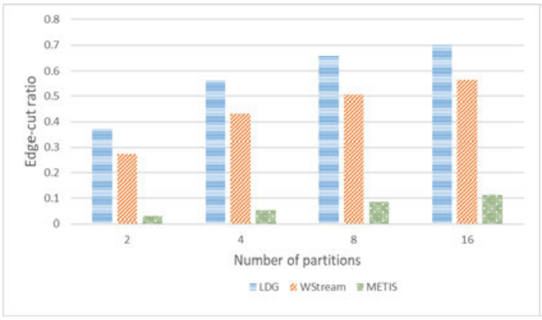
(D) GrQc Dataset



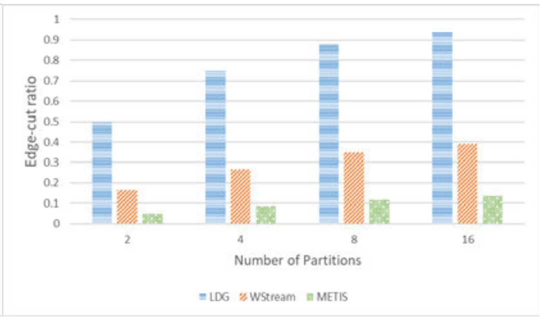
(E) Email-enron Dataset



(F) Wiki-vote Dataset



(G) Twitter Dataset



(H) com-DBLP Dataset

FIGURE 5.6: Edge-cut ratio comparison

the Twitter, 3elt, and 4elt dataset compared with METIS.

In conclusion, our WStream algorithm was able to significantly reduce edge-cut for different graphs ranging from 3000 vertices to 317080 vertices.

Load Imbalance Comparison

We compare the load imbalance of our proposed algorithm with the LDG algorithm and METIS for every partition setting. Table 5.3 shows the load imbalance between the LDG algorithm, WStream and METIS for different datasets. We use the WStream algorithm partitioning result with a window size of 100 for this comparison.

The WStream algorithm achieves a completely balanced allocation for the 3elt dataset with the 2 partitions setting. The LDG algorithm achieves a completely balanced partitioning for most of the datasets, except for the Wiki-Vote in the 2 partitions setting. However, for the 16 partitions, the WStream algorithm demonstrates an 85% load imbalance reduction compared with the LDG algorithm for the 4elt and WikiVote datasets. For all partitions, the Wiki-vote was better balanced using the WStream algorithm compared with the LDG algorithm, and the load imbalance in the 3elt and GrQc datasets is significantly reduced. It is quite balanced for the AstroPh dataset for 16 partitions with the WStream algorithm. However, WStream does not perform well for the com-DBLP, Twitter and Email-enron datasets in balancing the load for any number of partitions, but it is much more efficient in reducing communication than the LDG algorithm.

The LDG algorithm uses a capacity constraint to keep the partitions balanced. However, our WStream algorithm uses a balancing parameter to keep all the partitions balanced at a certain threshold. In reality, different real-world applications behave differently and some graph partitioning applications require that the computation load be balanced. On the other hand, some applications require communications to be reduced across partitions. Based on this reality, the graph partitioning objective should be whether to minimise the load imbalance or minimise communication. This is very subjective and application-dependent. Our WStream algorithm performs well in general, and it is application independent. It also offers a trade-off between load imbalance and edge-cut minimisation.

The WStream algorithm achieves load imbalance reduction for every dataset except for 3elt dataset, compared with METIS. WStream algorithm outperforms METIS for the big datasets such as com-DBLP, Twitter, Email-enron and AstroPh for every partition setting. WStream achieves 99% reduction for AstroPh and Email-enron for 2 partitions setting and more than a 77% load imbalance reduction for the AstroPh, Email-Enron, and Twitter datasets for every partition setting. It is clearly seen that, WStream achieves 97%-99.5% load imbalance reduction for the big dataset with billion edges like com-DBLP. However, except for the 2 partitions setting, WStream performs better than METIS.

TABLE 5.3: Load Imbalance Comparison

Dataset	Load Imbalance (standard deviation)											
	Number of Partitions											
	2			4			8			16		
	LDG	WStream	METIS	LDG	WStream	METIS	LDG	WStream	METIS	LDG	WStream	METIS
3elt	0.0	7.0	4	19.01	17.54	6.04	56.71	7.63	4.69	22.62	7.56	3.16
GrQc	0.0	25.0	74	258.94	18.95	22.01	324.20	15.16	14.79	65.94	9.96	6.44
4elt	2.5	5.5	2	3.42	6.98	9.2	0.33	14.27	32.32	94.23	11.68	11.69
Wiki-Vote	213.5	24.5	100.5	285.93	18.95	38.05	210.85	15.93	21.32	92.89	11.73	13.22
AstroPh	0.0	1.0	274	0	9.72	137.03	0.5	14.19	66.84	272.08	8.86	30.88
Email-Enron	0.0	1.0	551	0	24.50	268.01	0.5	19.46	137.28	0.43	13.99	59.89
Twitter	0.0	25.0	254	0.5	18.20	591.1	0.43	15.56	252.74	0.48	15.29	115.45
com-DBLP	0.0	22.0	4261	0.0	9.20	1531.16	0.0	16.80	659.09	0.5	11.95	418.99

Execution Time Comparison

As shown in Table 5.4, we compare the execution time for partitioning tasks between the WStream and LDG algorithms. We do not include the METIS algorithm in this comparison as we use two single-pass algorithms for the run time comparison. In this evaluation, it is clearly seen that WStream outperforms the LDG algorithm in reducing execution time for almost every experimental run except for the Wiki-Vote dataset. We also observe that, WStream can reduce the execution time remarkably for most of the datasets; for the 4elt dataset, it reduces the execution time by 35%-40% for all partition settings. As shown in the Table 5.4 the WStream algorithm performs well in reducing the execution time for large-scale datasets; for example, it reduces the CPU execution time for the Twitter, and Email-Enron datasets by 13%-37% and 31% respectively (except for two partitions). A significant time reduction occurs for billion edge, such as com-DBLP, with the WSstream algorithm compared with the LDG algorithm. The WStream algorithm achieves up to a 44% time reduction over the LDG algorithm. However, a performance drop is also observed for the WStream algorithm when processing the Email-Enron dataset for two partitions and for the Wiki-Vote dataset.

Dataset	Execution time (seconds)											
	Number of Partitions											
	2			4			8			16		
	LDG	WStream	METIS	LDG	WStream	METIS	LDG	WStream	METIS	LDG	WStream	METIS
3elt	0.89	0.77	0.004	0.97	0.83	0.004	0.99	0.94	0.004	1.02	0.85	0.008
CA-GrQc	1.30	1.12	0.004	1.31	1.11	0.008	1.29	1.09	0.012	1.41	1.15	0.012
4elt	10.75	7.01	0.008	10.32	7.80	0.008	9.99	7.60	0.012	10.32	8.75	0.012
Wiki-Vote	1.30	2.28	0.02	1.32	2.24	0.028	1.29	2.39	0.048	1.41	2.53	0.076
CA-Astro-Ph	21.51	20.85	0.036	27.25	19.67	0.048	22.28	13.13	0.068	27.69	13.10	0.088
Email-Enron	25.09	49.42	0.06	80.85	48.85	0.068	72.92	51.16	0.08	73.12	51.76	0.104
Twitter	802.21	557.02	0.18	784.53	455.97	0.2	794.96	480.55	0.25	750.09	524.39	0.28

TABLE 5.4: Execution Time Comparison

5.7 Theoretical Analysis

The time complexity for the WStream algorithm is $O(n + m + w + klogk)$, where n is the number of vertices and m is the number of edges, w is the number of traversals in the stream window for assigning each vertex to a partition, and k is the number of partitions. WStream takes graph input vertex by vertex, where n is the total number of executions to partition an entire graph and w is the number of operations required to traverse through the whole window. Furthermore, the m number of operations we need depends on the number of edges of a vertex in a window.

5.8 Application

PageRank is one of the well-known graph-oriented applications which ranks the pages of the web. It was invented by Page et al [82] for the giant search engine, Google. Until then, it was the most efficient web ranking algorithm being used by Google. The Graph Partitioning algorithm has a significant impact on the PageRank algorithm.

After evaluating the window-based partitioning algorithm, we applied a well-known and widely used graph application, PageRank, to our algorithm. We chose PageRank to evaluate this partitioning algorithm because of its popularity and widely used graph applications. PageRank covers a wide range of applications such as search, browsing, and traffic estimation. We used the MPJExpress MPI programming framework to evaluate this application and Nectar cloud virtual machines[2] were used for the execution. Each machine is installed with the Ubuntu 18.10 Linux operating system, and it has 6GB RAM and 4 VCPUs.

Table 5.5 shows the PageRank computation time for three datasets, that is Email-Enron, AstroPh and gplus. We compared the PageRank computation time with the LDG algorithm. It can be seen that WStream reduced the computation time by 30% for the Email-Enron dataset on four partitions compared with the LDG algorithm. However, it performed slightly better than the LDG algorithm for the two partitions of the Email-Enron dataset. Wstream significantly reduced the computation time for the AstroPh and gplus datasets compared with the LDG algorithm by 35%-50%. We can say that our Wstream algorithm undoubtedly outperformed the LDG algorithm in reducing PageRank computation time.

Dataset	Execution time (seconds)			
	Number of Partitions			
	2		4	
	LDG	WStream	LDG	WStream
Email-Enron	0.67	0.63	583.54	410.97
AstroPh	49.63	36.20	44.57	22.37
soc-gplus	136.67	105.49	131.03	73.80

TABLE 5.5: PageRank Computation Time Comparison

5.9 Summary

In this study, we have studied streaming graph partitioning by edge-cut. Stream-based partitioning has become prominent recently due to large-scale expansions of social media graphs, which require distributed processing. Several algorithms have been proposed to partition data in a stream and thus reduce the execution time for partitioning, while keeping load imbalance and edge-cut to a minimum.

In this chapter, we demonstrated that a stream window in streaming graph partitioning results in significantly higher quality partitioning. We proposed a streaming partitioning algorithm for large-scale graphs using a streaming window to minimise the edge-cut across partitions while reducing the imbalance among partitions. We compared our proposed WStream algorithm with the state-of-the-art LDG algorithm using real and synthetic graph data sets. The evaluation results clearly show that the WStream algorithm reduced the edge-cut by 40%-56% in comparison with the LDG algorithm for all datasets, except for the Wiki-Vote dataset. In terms of load balancing, our algorithm performed better than the LDG algorithm for 16 partitions except for the Email-Enron and Twitter datasets; it performed extremely well for the four partitions and eight partitions for most of the datasets. However, our proposed algorithm partitions the graphs faster. This makes the proposed algorithm quite suitable for cases in which graphs are growing at a rapid speed.

We considered static datasets in this study when the vertices and edges are continuously being added over time in a streaming manner. However, we did not consider the feature of deleting the vertices and edges from the partitioned data which is the characteristic of a fully dynamic dataset. In the next chapter, we will consider fully dynamic partitioning in which vertices and edges will be adding/deleting in real-time during partitioning. We will also use an auto-scaling mechanism to scale the number of machines based on computational load in real-time.

Chapter 6

Dynamic Graph Partitioning in Streaming Manner

In this chapter, we propose a novel dynamic graph partitioning algorithm in a streaming manner. Besides that, a scaling algorithm is also proposed to launch or shut down a machine according to the computational load in real-time.

6.1 Motivation

In recent years, large-scale graph-oriented applications have received attention due to their participation in real-world applications such as PageRank calculation, World Wide Web crawling and protein-protein interactions. These applications are communication-intensive and the computational load must be even between the partitions to maintain an efficient, distributed graph processing system. Moreover, these graph-oriented applications have dynamic behaviour, such that vertices and edges are continuously being added or removed over time. Time-evolving large graph-oriented applications have huge computational cost. Thus, they are incapable of being handled in a single machine due to memory bottleneck. Consequently, for any analysis purpose, large graphs need to be partitioned across a cluster in a distributed system. Reducing network communication and balancing the load between the partitions are the criteria required to achieve effective run-time performance in a distributed graph processing system. As the vertices and edges are frequently being removed or added in a large dynamic graph, it is necessary to partition the graph wisely in real-time by keeping the network communication and the load imbalance as low as possible during partitioning. A number of existing dynamic graph partitioning algorithms has been proposed to address the above problem. However, these partitioning methods are incapable of scaling the resources and handling the stream of data in real-time.

In this study, we propose a dynamic graph partitioning method called Scalable Dynamic Graph Partitioner(SDP), using a one-pass streaming technique with the following contributions: 1) An algorithm which can handle the dynamic changes of a large-graph when new vertices and edges are added or removed continuously over time, and which can assign vertices and edges to an appropriate machine. 2) A dynamic partitioning algorithm which accepts graph data in a streaming manner. 3) A vertex migration technique, in order to scale up or down the resources and reduce the imbalance as much as possible. 4) A communication balancing strategy used dynamically for edge-based balancing. Experiment results show that the proposed method achieves significant improvement in reducing communication cost and balancing the load dynamically, compared with previous algorithms. Moreover, the proposed algorithm significantly reduces the execution time during partitioning.

6.2 One-pass Dynamic Graph Partitioning

6.3 Introduction

In recent days most of the graph-oriented applications have a dynamic behaviour, which means that the vertex or edge might go off or gain a new vertex or new edges. For example, thousands of Twitter users update their tweets per second [117]. This behaviour of the dynamic graphs creates a computational load imbalance between partitions and increases the edge-cuts and communication cost as well.

Most real-world graph applications tend to receive graph data continuously as a stream of graph data in a real-time manner. It is necessary to have a graph partitioning algorithm that can distribute the stream data among the partitions in a one-pass manner, as the vertices arrive. A streaming one-pass graph partitioning algorithm receives the vertices one by one and decides the respective partitions with little connectivity information of a vertex. When a graph is updated over time, it is necessary to keep the computational load balanced, keeping the communication to a minimum. If the algorithm has to visit all the partitions again and revisit the whole graph to perform the repartitioning for an updated graph, it is very expensive in terms of computational complexity.

The computational load of a partition in a dynamic graph always changes over time, by adding or removing vertex elements from a partitioned graph. Since the incoming number of vertices is unknown and the number of vertices might be removed anytime from a partition, a huge imbalance between partitions is created. Consequently, this creates more cut edges and causes unbalanced partitions. Unbalanced partitions might also cause an unnecessary allocation of a partition which is a waste of computational resources.

In order to cater for the ever-increasing computational load as per the demands of an application, scalability becomes an important factor in dynamic graph partitioning. In this study, we also propose a dynamic machine allocation method to allocate a new machine, as per the demands of the computational load. Dynamic allocation of a machine is another important aspect in balanced graph partitioning, as over time the size of a graph continuously changes. It is important to consider a flexible allocation of a new partition according to the computational load. This study addresses these features by allowing for the decrease and increase of the number of partition allocations, according to the computational load. We use a capacity threshold to decide the allocation of a new machine or shutting down of an unused machine from the cloud.

A communication aware balancing strategy is also taken into account when assigning vertices to a corresponding partition. It trades-off with the number of communications, while minimising the load imbalance between partitions.

6.4 Related Work

In this section, we discuss the related work on dynamic partitioning. In order to compute the large-scale of streaming data in [110], a well-known streaming graph partitioning is proposed which is called Linear Deterministic Greedy (LDG). It has a central graph loader, which loads the data and distributes them among the available workers. This heuristic assigns a vertex to the partition with which it shares the most edges. 16 partitioning heuristics are evaluated with 21 different datasets. Graph datasets have been used from several domains: the World Wide Web, social networks, finite element meshes and synthetic datasets. The authors achieved different results from different datasets. This streaming partitioning method makes heuristics scalable in the size and the number of partitions of graphs. It has significant speed up achievement of PageRank computations on Spark [131] by 18% to 39% for large social networks. Another greedy heuristic algorithm proposed in [7] uses an unweighted, deterministic greedy algorithm, instead of using the weighted penalty function in order to partition vertices. This algorithm also uses a factorisation technique that aims to reduce the neighbouring vertices, rather than the edges across the partition. In other words, a vertex-cut partitioning technique is employed here, which is well-suited for large-scale natural graphs.

Adaptive partitioning is relevant in dealing with a dynamic graph. Few researchers have explored this technique to keep the partition balanced and minimise the communication. The main idea of this technique is to migrate the vertices/edges from one partition to another meeting some criteria towards reducing the load imbalance and communication cost. A greedy vertex migration technique [117] was proposed with the aim of partitioning a dynamic graph.

To compute PageRank in a parallel manner, a site-based graph partitioning and repartitioning technique [17] was proposed. Sparse matrix-vector multiplication is responsible for incrementally growing web matrices data, which are stored in a distributed manner. PageRank computation requires high-efficiency and low processing overhead calculations because PagRank [12] computation has frequently repeated iterations. An algorithm [17] was proposed with a sparse-matrix multiplication technique to achieve high efficiency and parallelism, in order to focus on reducing pre-processing overhead in PageRank computation. Repartitioning techniques were used in this algorithm. A common problem in dynamic web data is the addition and deletion of new pages. A large graphs partition management system was proposed [129], in order to facilitate searching and mining a large graph. Minimising the inter-machine communication was the aim of developing this system. To perform the graph partitioning task, a two-level (static partitions and dynamic partitions) structure was introduced which helped to improve the query response time and throughput. This two-level partitioning structure is effective as it adapts in real time when query workload is changing over time.

An adaptive unstructured meshes dynamic partitioning algorithm [120] was proposed with parallelisation. This algorithm uses a relative gain optimisation technique which aims to balance workload and reduce the inter-partition communication overhead. A few series of adaptively refined meshes were applied for the purpose of the experiment and the results indicate that they provide better partitioning than a static partitioner.

A distributed system, Kineograph [21], was proposed to handle the rapid changes in graphs and to capture the relationships. Kineograph also supports graph-mining algorithms to extract real-time information from a fast-changing graph.

Vertex replication is another technique to handle an ever-changing graph in a distributed environment. Vertex replication imitates the vertex in a partition to reduce the communication cost in a distributed graph processing system. A vertex replication algorithm [48] was proposed with the aim of attaining better access locality of a vertex, by replicating the vertex which resides in another partition. Eventually, it does minimise the communication cost across the network.

A few more researchers proposed the vertex replication method in graph partitioning, while minimising the workload imbalance and inter-machine communication in a distributed network. Of them, dynamic replication-based partitioning was proposed in [129] and this replicates the vertex adaptively, based on the change of workload. To improve performance during the frequent changes in workload, an historical log-based partitioning technique called LogGP was proposed [128]. LogGP framework analyses and reuses the historical statistical information to refine the partitioning result. It has great advantages in utilising the historical partitioning results to generate a hypergraph. The authors argue that running statistical analysis of historical partitioning logs can provide an improvement on partitioning results.

Dynamic graphs sometimes require a repartitioning process to maintain the balance of graph-partitioned data in order to improve system performance. Good partitioning algorithms with repartitioning features are in demand for handling huge dynamic graph data. A study was undertaken on repartitioning online social network data in [84]. The authors aimed to improve the scalability by reducing the inter-partitioning communication. A replication method was used to reduce the communication among nodes. An in-memory based dynamic partitioning technique was proposed in [77] to handle the large dynamic graph. This algorithm achieved significant low-latency communication in query processing. The authors provided a vertex replication policy that monitors the incoming vertices and decides what data to replicate. It was evaluated on a social network graph, and the result shows that this technique reduced the network bandwidth significantly. Moreover, the technique also handled a very large graph efficiently. Proper placement of a newly added vertex, in a dynamic graph, by using the cost-effective method, was proposed in [4]. A vertex migration technique was also added to this study in order to balance the partitions, due to deletion of vertices from a partition. The migration of the vertex depends on the latency and communication cost of the particular vertex being migrated. The authors proposed a set of heuristics to reduce communication cost, and to balance the partitions. However, these heuristics do not accept the graph data in a stream manner and do not make any decisions in real time.

6.5 System Architecture

This section describes the complete architecture and processing flow of our dynamic partitioning technique.

The graph data distributes to the number of machines in the distributed system in order to balance the computational load evenly between the machines. A Graph Loader also resides in the master machine which decides the nature of the input. The master machine takes the input, and a stream generator resides in the master machine to generate the stream of data from the Graph Loader. The stream generator forwards the input to the partitioner to perform the addition/deletion. Our dynamic partitioning method accepts three kinds of inputs(for example add, delete a vertex, and delete an edge). Figure 6.1 shows the architecture of this study.

The partitioning process starts with one worker machine and adds the partition dynamically, according to the load. The adding criteria of new partitions is explained in Section 6.7.3.

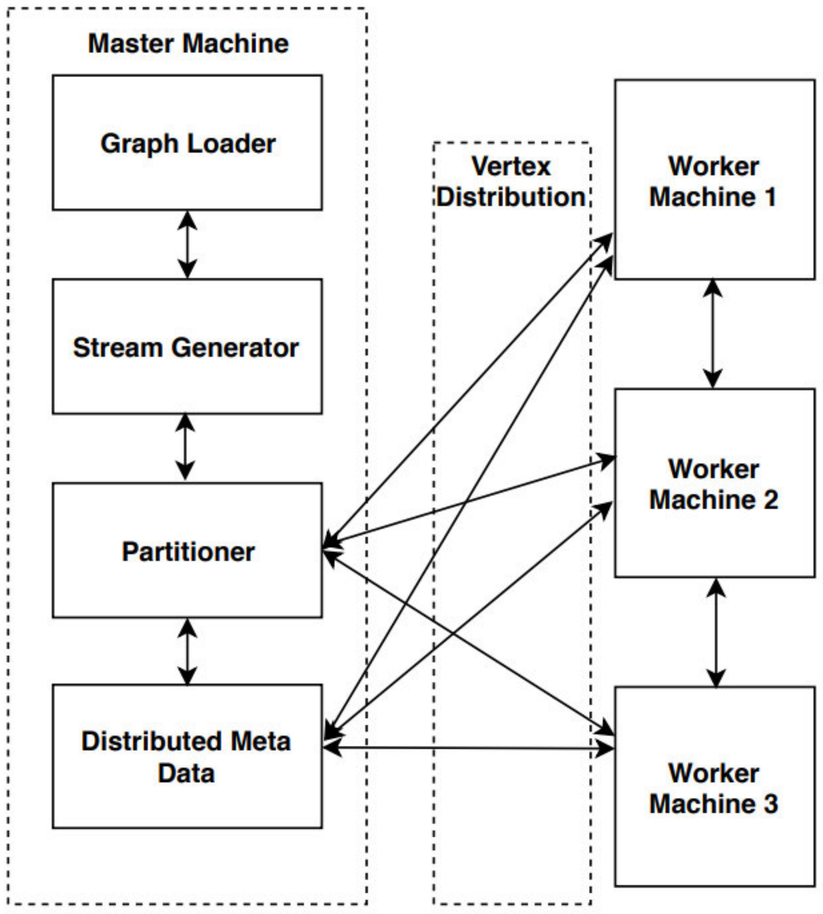


FIGURE 6.1: System Architecture

6.5.1 The components of system architecture

- **Graph Loader:** The Graph Loader loads the input from the disk memory. For example: add vertex, delete vertex and delete edge. The loader receives input from the disk uniformly and at random for this purpose and forwards to the Stream Generator in order to create the stream of data before forwarding to the partitioner.
- **Stream Generator:** A stream generator resides in the master machine to generate the stream of graph data from the input dataset. Each vertex arrives with its associated edges in the stream, sequentially from the Graph Loader. It is responsible for forwarding the graph input to the partitioning algorithm for the purpose of adding or deleting.

- **Distributed Meta Data:** It is located in the master machine to store the graph information, which can be used for partitioning purposes by the partitioner.
- **Data Receiver in Worker Machine:** A data receiver resides in the worker machines to receive the vertices and edges from the master machine and to send the acknowledgement to sender.

6.6 Problem Statement

Problem 1 *Partitioning of a dynamic graph $G = (V, E)$ into k number of subgraphs and allocate each subgraph to the P_k partition. The number of vertices V in each partition increases or decreases over time t, so the number of vertices of a partition after t time would be $|V_k(t)|$. The graph partitioning technique always aims to reduce the cut edges $E(u, v)$ between partitions, u and v. Two different end points (uandv) of an edge E reside in different partitions, such that, $min E(u, v) = \sum_{i=1}^{n} E(u, v)$*

Problem 2 *In k way graph partitioning, the algorithm always tends to divide the entire graph G into k number of sub-graphs. In a dynamic graph partitioning the size of the graph is continuously growing and the number of partitions is dynamic and unknown. The number of partitions P_k should be allocated according to the computational load over time t as follows: $|P_k(t)|$, such that the k value increases or decreases as per the computational load.*

6.6.1 Research Questions

We discussed the problems in the previous section regarding the dynamic graph partitioning in a one-pass manner. We are answering the following research questions in this study:

- How can we repartition a dynamic graph in a stream manner by reducing the cut edges and load imbalance as much as possible?
- How can we reduce the cut edges, and load imbalance by allocating the partitions dynamically, as per the demand of computational load?
- How can we trade-off between cut-edges and load imbalance efficiently, using the communication aware load balancing strategy?

6.6.2 Methodology

Based on the literature, the major issue in dynamic graph partitioning is to allocate the computational load as it arrives in real time, and to utilise the resources as needed, while, at the same time, minimising the communication and balancing the load as much as possible. This all happens in a real time manner. This study overcomes these issues and proposes a novel dynamic graph partitioning technique. Few studies have been undertaken on the streaming graph partitioning technique with static graph data. To the best of our knowledge, dealing with a dynamic graph at the same time partitioning in a streaming manner in a cloud environment, has not yet been studied. This research focuses on partitioning a dynamic graph in a streaming manner, in a cloud environment. We evaluated our partitioning algorithm with the number of Nectar cloud instances; every instance was equipped with the same resources.

6.7 SDP: Scalable Dynamic Graph Partitioner

This dynamic graph partitioning algorithm takes the stream of vertices and their associated edges as an input in a single-pass manner. This algorithm also accepts the input to remove vertices and edges at a certain point, in order to test the dynamism of our algorithm. The algorithm aims to minimise the edge-cut among partitions and to make the partition balanced as low as possible. Each time a vertex arrives for partitioning, the algorithm decides a suitable partition to allocate that vertex immediately. The algorithm stores the summary of partitioning results, in a distributed meta data file in the master machine. The meta data is used as a reference to allocate the future vertices. The summary includes vertex information and its allocated partition index. After allocating or deleting each vertex of edges, the graph summary will be updated. Algorithm 5 depicts the updating graph summary.

where, $parttionInfoMap$ is to store the summary of the graph and the partition index. The partitioning algorithm uses this information to assign a vertex to a proper location.

The algorithm starts by taking a tuple $V < vertex, edges >$ as an input, and the input comes in a single pass manner sequentially. It receives tuple at any point of time

Algorithm 5 Update the graph summary

INPUT: p = partition index, v = arrived vertex, $MAXCAP$ = maximum capacity of a partition, $averageLoad$ = the average load of the partitions

```
  $partitionInfoMap < p.List < v >>$
  if ($!partitionInfo.containsKey(p)$) then
    $List < Integer > list$
    $list.add(v)$
    $partitionInfoMap.put(p, list)$
  else
    $partitionInfoMap.get(p).add(v)$
  end if
```

as a stream of data is added to a machine and is removed over time. Based on the type of input it receives, the algorithm acts accordingly. The type of input is decided by the Graph Loader in the master machine. If the algorithm receives an input to add the vertices, the vertex allocation technique is employed to assign a vertex to a partition. The vertex allocation technique is described in Section 6.7.1 and the assigning algorithm is depicted in Algorithm 9. Before assigning every vertex to a partition, the balanced strategy checks the imbalance of computational load among the partitions. Moreover, if all partitions exceed maximum capacity, the algorithm adds a new partition to cater for the upcoming load and thus, maintains the scalability. We propose a communication-aware balancing strategy and also an adding/removing partitioning technique. These are explained in Section 6.7.2 and Section 6.7.3 respectively. The partitioning method receives the input from the Graph Loader in a sequential manner. If the algorithm receives an input to add/remove the vertices or edges, it adds/removes the vertices or edges. After removing vertices or edges from a partition, the partitions might become unbalanced. As a result, it is necessary to make the partition balanced. A communication aware balancing strategy is employed here before assigning a vertex to a partition. For the balanced partition we take the number of communications in the balancing method into account. This checks the number of communications over time while balancing the load. The key idea here is to trade-off the number of cut edges with the load imbalance. Algorithm 6 depicts the whole partitioning strategy dynamically in a one-pass manner.

Where, v is the vertex which has arrived in the stream and Graph Loader decides randomly what kind of input it is and α represent the type of input. The algorithm

Algorithm 6 Dynamic Partitioning

INPUT: V = set of partitioned vertices, P = number of partition indexes, v = vertex arrived in stream, $edge < v_1, v_2 >$= edge arrived in the stream, α= type of input, $E(v)$ is the associated edges arrived with vertex v, $partitionInfoMap < p < List >>, edgeInfoMap[] < vertex, List < edges >>, TH$ = balancing threshold. $averageLoad$ = average load of all the partitions.

```
$MAXCAP \leftarrow$ the maximum capacity of each partition
if ($\alpha = add$) then
  $thresHold \leftarrow addingThreshold(|E|, |P|)$
  if ($MAXCAP \leq thresHold$) then
    $updateSummery(P + 1, v, MAXCAP, averageLoad)$
  end if
  if ($P > 1$) then
    $\sigma \leftarrow findImbalance(P, partitionInfoMap < p, < List >>)$
  end if
  if ($\sigma > TH$) then
    $partitionIndex \leftarrow assignVertex(v, P, V, E(v))$
    $updateSummery(partitionIndex, v, MAXCAP, averageLoad)$
  else
    $partitionIndex \leftarrow findMinimum(partitionInfoMap < p < List >>, P)$
    $updateSummery(partitionIndex, v, MAXCAP, averageLoad)$
  end if
else
  if ($\alpha = deleteVertex$) then
    $deleteVertex(v, P, edgeInfoMap[] < vertex, List < edges >>$
    $, partitionInfoMap < p < List >>)$
  else
    if ($\alpha = deleteEdge$) then
      $deleteEdges(edge, < v_1, v_2 >, P, edgeInfoMap[] < vertex, List < edges >>)$
    end if
  end if
end if
```

also receives the edges to remove, which is $edge < v_1, v_2 >$. The imbalance parameter is obtained from the calculation of the standard deviation of the total number of edges in the partitions. Equation 6.10 shows the calculation of standard deviation.

$MAXCAP$ is the maximum capacity of a partition and $averageLoad$ is the average edge load of all partitions which can be calculated by $calculateAverageLoad(|V|, P)$ function. The $assignVertex(v, P, V, E(v))$ function is used to assign the vertex to the respective partition. The details of the vertex assigning technique are in Section 6.7.1. Algorithms 7 and 8 show how to delete the vertices and edges respectively. We use an $updateSummery(partitionIndex, v, MAXCAP, averageLoad)$ function to update the graph summary each time we partition or delete any vertices or edges. Algorithms 7 and 8 depict the method of deleting vertices and edges respectively.

Algorithm 7 Delete Vertices

```
INPUT: $v$ = vertex arrived in stream, $partitionInfoMap < p < List >>$,
$edgeInfoMap[] < vertex, List < edges >>$

  for $List < Integer > v : partitionInforMap.values()$ do
    $v.removeAll(vertex < v >)$
  end for
  for $i = 0$ to $k$ do
    $Iterator < Map.Entry < Integer, List < Integer >>> iter = edgeInfoMap[k].entrySet().iterator();$
    while ($iter.hasNext()$) do
      $Map.Entry < Integer, List < Integer >> entry = iter.next();$
      if ($v == entry.getKey()$) then
        $iter.remove()$
      end if
    end while
  end for
```

Algorithm 8 Delete Edges

INPUT: $edge < v_1, v_2 >=$ edge (source and destination vertices in the list) arrived in the stream, $edgeInfoMap[] < vertex, List < edges >>$

```
for $i = 0$ to $k$ do
  for $(List < Integer > edges : edgeInfoMap[k].values())$ do
    $edges.removeAll(edge < v_1, v_2 >)$
  end for
end for
```

6.7.1 Vertex Assigning Method

In general, the way to minimise the cut edges between partitions is to allocate the vertices to a partition which contains the greatest number of neighbouring vertices. It is always desirable to allocate the connected vertices to the same physical machine. From the stream of graph data, the candidate vertex arrives with its associated edges. The partitioning algorithm assigns the vertex with its associated edges to a respective machine. Algorithm 9 shows the vertex allocation strategy. Our algorithm aims to identify the best locality of the arrived vertex to minimise the edge-cut. This vertex allocation technique tends to send a vertex to the partition which has the most connected vertices. The algorithm checks all the partitions' information from the partition summary in order to decide which partition has the most connections of arrived vertices in the stream. The algorithm then allocates the vertex to that particular partition. However, if two or more partitions have the same number of connections of the candidate vertices, the algorithm assigns that candidate vertex to the partition with a minimum load. If it does not find a connection in any of the partitions, then the candidate vertex is allocated to the partition randomly and in a uniform manner. Finding the partition that has the most connections is calculated with the equation 6.1 and finding the minimum load of a partition is assessed with Algorithm 10.

$$\arg\max_{k \in P} \{|E(V_c) \cap P_k|\} \tag{6.1}$$

where, k is the number of partitions. $E(v)$ is the associated edges arrived with vertex v and $P(k)$ is the set of vertices in kth partition.

Algorithm 9 Vertex Assigning Method

INPUT: v= the candidate vertex that available for partition in the stream, k = number of partitions, $E(v)$ is the associated edges arrived with vertex v, P = set of partitioned vertices.

OUTPUT: partition index

```
for $i = 0$ to $k$ do
  $partitionInfoSet < Integer > (P(i))$
  $intersectSet \leftarrow partitionInfoSet.retainAll(E(v))$
  $size \leftarrow sizeoftheoftheintersectSet$
  if ($size > tempSize$) then
    $tempSize \leftarrow size$
    $partitionIndex \leftarrow i$
    $edgeInfoMap[partitionIndex].put(v, E(v))$
  else
    if ($size == tempSize$) then
      $paritionIndex \leftarrow i$
      $edgeInfoMap[partitionIndex].put(v, E(v))$
    else
      $partitionIndex \leftarrow random(k)$
      $edgeInfoMap[partitionIndex].put(v, E(v))$
    end if
  end if
end for
```

Algorithm 10 Finding minimum load

INPUT: $partitionInfoMap < p < List >>$, k = number of partitions

OUTPUT: partition index

```
$firstPartitionSize \leftarrow sizeofthefirstpartition$
for ($i = 0$ to $k$ ) do
    if ($firstPartitionSize > ithpartitionsize$) then
        $partitionIndex \leftarrow i$
    end if
end for
```

6.7.2 Communication-aware balancing strategy

The load of a partition is the number of external and internal connections of that partition. We propose a balancing strategy to keep the partitions as balanced as possible. The number of communications between the partitions is also taken into account in order to decide the imbalance of computational load among partitions. The number of communications has a great impact on balancing the load. We use average load difference and communication aware load deviation in order to decide the imbalance of the partitions. In this study, we assume that each machine in the distributed system has the same resources and computing power. We used the following variables to complete the load balancing task: The average load difference is AVG_d, the threshold is TH, and the weighted deviation is W_{dev}.

If $AVG_d > TH$ then the algorithm assigns the vertex v to partition P_l, otherwise the algorithm executes the vertex assigning method (Algorithm 9) to assign the vertices to a suitable partition where v is the vertex that has arrived in the stream to be allocated to a partition. The average load difference (AVG_d) can be calculated with the following formula :

$$AVG_d = (P_h - P_l)/n \tag{6.2}$$

where, P_h is the partition with the highest load, P_l is the partition which has the lowest load and n is the number of partitions. We obtain the balancing threshold using the following equation:

$$TH = W_{dev} - Load_{dev} \tag{6.3}$$

where, $Load_{dev}$ is the load deviation among the partitions. The calculation of the load deviation is the Standard Deviation of a load of the partitioning in a distributed system. The weighted deviation leverages the communication with the computational load. Because any partition in a distributed system has large number of communications, they carry more computational load than other partitions. Weighted deviation decides the imbalance among the partitions. This balancing strategy ensures good computational load distributions. Weighted deviation is denoted by W_{dev} which can be calculated by using the following equation:

$$W_{dev} = (edge^t / cut^t) * Load_{dev} \tag{6.4}$$

where, $edge^t$ is the edges arrived over time t, and cut^t is the cut edges in t time.

The communication aware balancing strategy ensures a well-balanced computational load while the number of cut edges is also taken into account in deciding the imbalance of a partition.

6.7.3 Scalability

Scaling Out: When the capacity of all the partitions exceeds the constraint C, then the additional partition needs to be included in the system in order to accommodate the increasing graph data. We use an adding threshold to add a new partition in the system, which can be defined by the following equation:

$$addingThreshold = \frac{|E^t|}{|P^t|} \tag{6.5}$$

where, $|E^t|$ is the total number of edges that has been assigned to all partitions in time t and $|P^t|$ is the total number of partitions over time t. The threshold decides when to add a new partition in the system. If the $C \leq addingThreshold$, then the system adds a new instance in the system. C is the capacity constraint of a partition which is the maximum computational load of a partition. In this study, we assume that the capacity of all the partitions is the same.

Scaling In: We use a vertex migration threshold to scale down the resources from the cloud. The idea is to shut down the unnecessary or unused machine from the system. The determination of shutting down a machine depends on the l value. If two machines have a computational load of less than the l, the algorithm migrates the vertices and their associated edges from the source machine to the destination machine. The source machine (*sourceMachine*) is the machine which has the minimum load of all the machines; *sourceMachine* can be defined with Algorithm 10.

$$l = (toleranceParameter * MAXCAP)/100 \tag{6.6}$$

The destination machine is the machine which is available to accept more load. We use *destinationThreshold* to decide the destination partition to migrate the computational load. To determine the availability of the machines to accept more load, we use the *destinationThreshold* threshold. A machine accepts computational load until the machine load is less than or equal to the *destinationThreshold*, which keeps some spaces for the upcoming data from the stream.

$$d = (param * MAXCAP)/100 \tag{6.7}$$

$$destinationThreshold = MAXCAP - d \tag{6.8}$$

6.8 Experimental Settings

In this section, we discuss the experimental setup, and performance metrics of this study. We use Java programming language to implement the algorithm. Java socket programming is used to implement the distributed environment to partition a graph dynamically. We use Nectar cloud machines to set up the experimental settings. We use a master machine to allocate the computational load to the worker machines with the partitioning algorithm. Each machine's characteristics are as follows:

Ubuntu 18.04 LTS operating system, m2.medium type machine with 30 VCPUs, 6 GB RAM and 30 GB Disk.

6.8.1 Dataset

We used a variety of synthetic and real graph datasets to evaluate the dynamic partitioning. Table 6.1 shows the lists and characteristics of the datasets used in this study.

Name of Dataset	$\|V\|$	$\|E\|$	Type
3elt (Synthetic) [121]	4200	13722	Finite-element mashes
GrQc[70]	5242	14496	Collaboration Network
Wiki-vote[70]	7,115	99,291	Social
4elt (Synthetic)[121]	15,606	45,878	Finite-element mashes
AstroPh [70]	18,772	198,110	Citation
Email-enron[70]	36,692	183,831	Communication
Twitter[70]	81,306	1,768,149	Social

TABLE 6.1: Characteristics of Datasets

6.8.2 Performance Metrics

We compared our algorithm with the most recent dynamic partitioning algorithm [4]. We observed the performance of our proposed algorithm using the following performance metrics: i) edge-cut ratio; ii) load imbalance; iii) execution time. We observed the number of external connections of a vertex from one partition to another partition as a cut edge. We calculated the ratio of the edge-cut by using the following equation:

$$edgecutratio = \frac{|E(u,v)|}{|E|} \tag{6.9}$$

where, $|E|$ is the total number of edges of a graph and $|E(u,v)|$ the total number of edges between u and v across partitions.

The load of a partition is the number of external and internal connections of a partition. Standard deviation of the total number of external and internal connections(edges) of all the partitions is the load imbalance. The following equation is used to calculate the standard deviation of the total number of external and internal edges of a partition:

$$loadImbalance = \sqrt{\frac{\Sigma|e - \bar{e}|^2}{n}} \tag{6.10}$$

where, e is the total number of external and internal edges of a partition and n is the total number of partitions.

We measured the execution time from the start of partitioning until the end of partitioning. The time taken to receive the input is also taken into account in the execution time, as the streaming partitioning algorithm executes as the data stream arrives.

6.8.3 Experimental scenario

In this sub-section we discuss some experimental scenarios of our dynamic partitioning algorithm.

Adding/Deleting Vertices

In a dynamic graph processing system, the addition/deletion of vertices or edges occurs over time as per the demand of a graph application. Consequently, the partitioned graph structure changes over time, which creates the unbalanced partitions and also increases the number of communications among partitions. The proposed dynamic graph partitioning algorithm accepts the graph input sequentially. It adds and deletes the vertices dynamically in a streaming manner. The algorithm assigns the vertex to the respective partition as it arrives.

In a regular interval of time, the algorithm adds and deletes the graph data from the input dataset. In each interval, we add 25% of the dataset and then delete 5% of the dataset from a respective partition. In each interval, the algorithm observes edge-cut and the number of partitions used. The number of vertices to add and delete in each time interval by using the following formulae is as follows:

$$numAddedVertex = (totalVertex * addPercentage)/100 \tag{6.11}$$

$$numDeleteVertex = (totalVertex * deletePercentage)/100 \tag{6.12}$$

where, $totalVertex$ is the total number vertices of the input dataset, $addPercentage$ and $deletePercentage$ are the percentages of the entire dataset of adding and deleting the vertices respectively.

Adding a partition dynamically

Initially, the partitioning starts with a master machine and a worker instance in the Nectar cloud environment. A capacity constraint C of each worker machine is used to check the maximum capacity of a worker machine. If all the running worker machines have reached the maximum capacity of C, the algorithm dynamically creates and launches another instance to accommodate the ever-increasing graph data load. Over time t, some vertices/edges may be deleted from a worker machine, making the worker machine available to receive more workload. According to the vertex assigning algorithm and balancing strategy, the master machine assigns the vertices to that available worker. Section 6.7.3 explains in detail the criteria for adding a new partition dynamically.

Deleting partition dynamically

As per the demands of the workload, the dynamic algorithm removes the unnecessary/unused instances from the system. Whenever any worker machine has the capacity to receive more load, it accepts the load until it has 5% capacity available.

6.9 Result Discussion

In this section, we discuss the results evaluated from the experiments with different types of datasets. The comparison of our algorithm with the existing algorithm[4] is also discussed here.

6.9.1 Edge-cut comparison

Edge-cut ratio indicates the performance of graph partitioning in terms of the communication overhead among the partitions. A higher edge-cut indicates higher communication overhead among machines in a distributed system.

We captured the edge-cut over time at an interval of every 25% of the whole dataset. We compared our algorithm with a recent well-performing, dynamic partitioning algorithm. Figure 6.2 illustrates the edge-cut comparison of a number of datasets from different ranges. It is clearly seen that our algorithm obtains a better edge-cut ratio than the existing algorithm. For the 3elt and 4elt dataset the algorithm has an 80%-90% reduction of edge-cut at the beginning of the partitioning. As shown in Figure 6.2(c-g), the edge-cut ratio decreases when the partitions have more added vertices. It is expected that when the partitions receive more information of a graph, the partitioning performance improves.

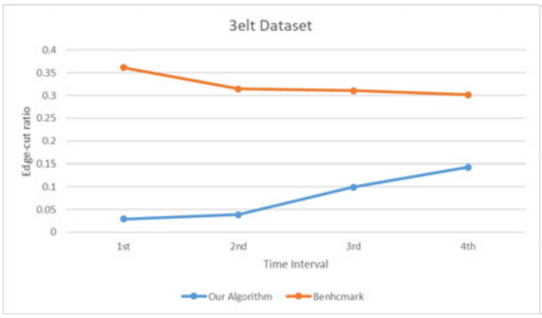
We also compare our algorithm with the METIS algorithm which is the best state-of-the-art graph partitioning algorithm of all time, to observe how closely our algorithm is to the static graph partitioning algorithm. As shown in Figure 6.3, it is obvious that our algorithm performed better for all the datasets than previous algorithms. However, for the rest of the datasets, our algorithm's performance is close to the METIS algorithm. This is understandable , as it is difficult for a streaming algorithm to achieve a better edge-cut ratio than an offline graph partitioning because offline graph partitioning algorithm has the entire graph information before the start of partitioning graph.

6.9.2 Load Imbalance Comparison

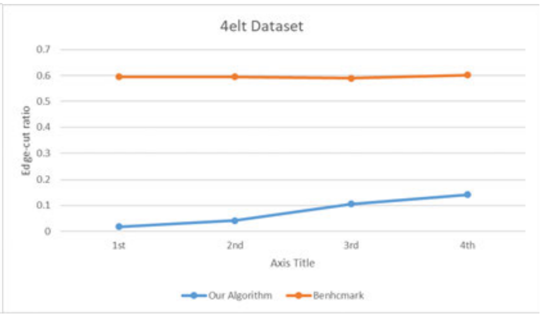
As shown in Figure 6.4, we illustrate the load imbalance comparison between our algorithm and previous algorithms. It is obvious that the reduction of load imbalance in our algorithm is better than the previous algorithms for all the datasets. Our streaming algorithm manages to reduce the 60% -70% load imbalance for all the datasets, except the GrQC dataset. The GrQC dataset performed almost similarly to previous algorithms. However, our algorithm performed well in reducing the edge-cut for the GrQC dataset.

6.9.3 Impact of Addition and Deletion

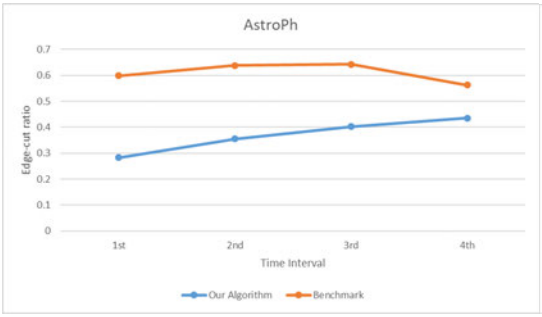
In this section, we examined the effect of dynamically adding and deleting the vertices or edges for a variety of datasets. Figure 6.5 shows the trend of the edge-cut



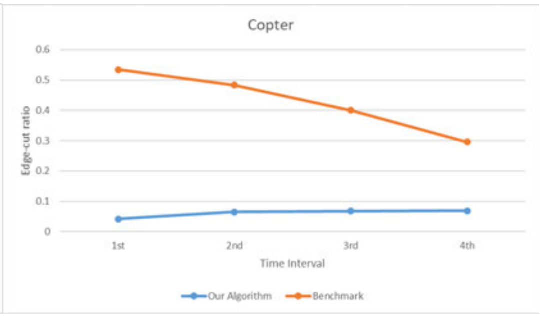
(A) 3elt Dataset



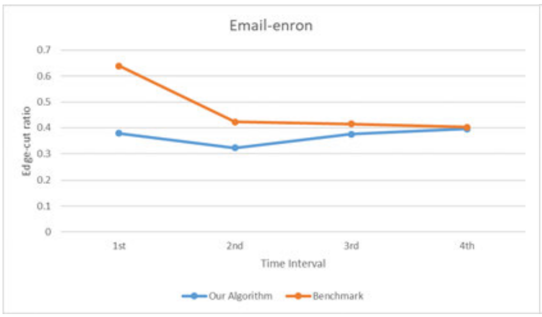
(B) 4elt Dataset



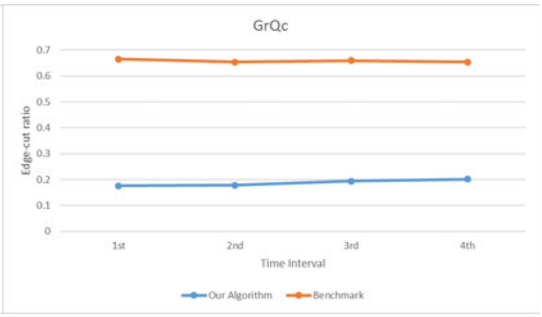
(C) AstroPh Dataset



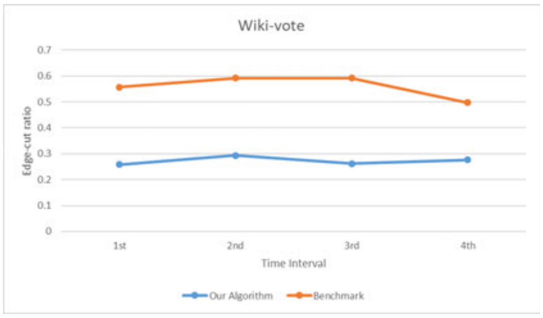
(D) Copter Dataset



(E) Email-Enron Dataset



(F) GrQc Dataset



(G) Wiki-Vote Dataset

FIGURE 6.2: Edge-cut comparison of different datasets

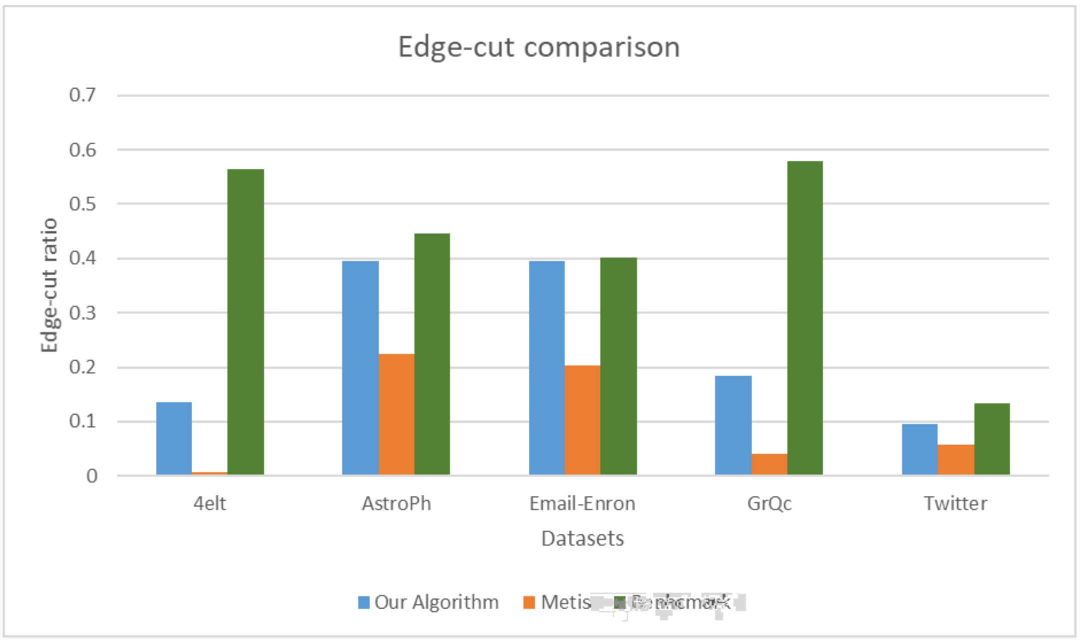


FIGURE 6.3: Edge-cut comparison

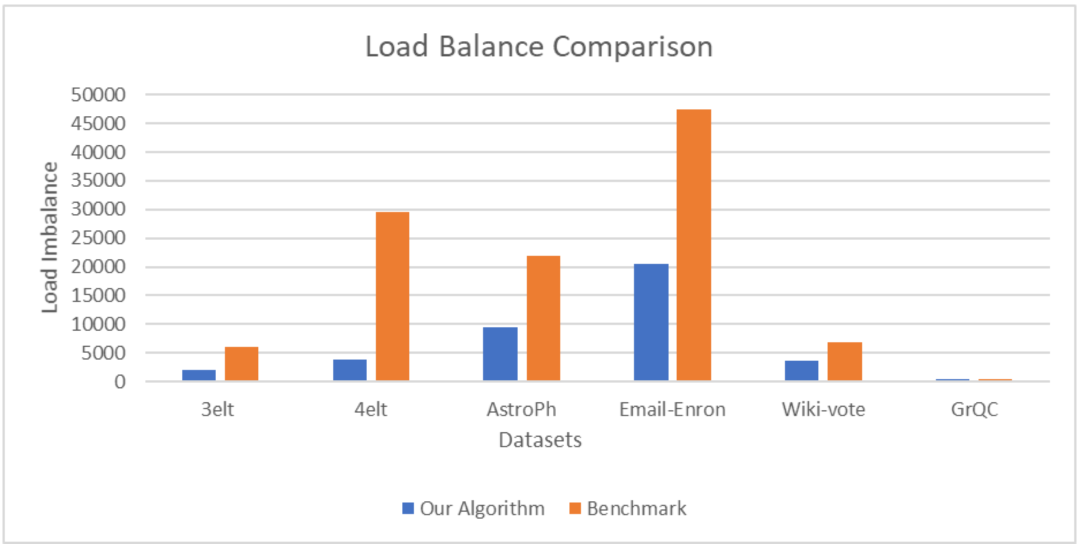


FIGURE 6.4: Load Balance Comparison

ratio over time *t*. In most cases, the edge-cut ratio decreases after deletion, while the graph changes in *t* time. Figure 6.5 shows the impact of the addition and deletion of vertices and edges over time. We capture the edge-cut performance in four intervals.

At each interval, after deleting vertices and edges from the partitions, the number of edge-cuts decreases as expected. However, as time goes by, the ratio of edge-cuts increases as the deletion percentage is less than the addition percentage. However, an exception happens with the Copter, 3elt and Wiki-Vote datasets, as shown in Figure 6.5(f), at the 3rd interval. The edge-cut ratio after the deletion is less than the 2nd interval. This is because the deleted vertices were connected with a large number of internal edges.

6.9.4 Impact of Number of Partitions

This section discusses the effect of the number of partitions in terms of communication cost. As shown in Figure 6.6, it is obvious that the communication cost increases as the number of partitions increases. However, in the Copter dataset there was a slight decrease of edge-cut after adding the third partition, as there was deletion of vertices happening at that stage of partitioning.

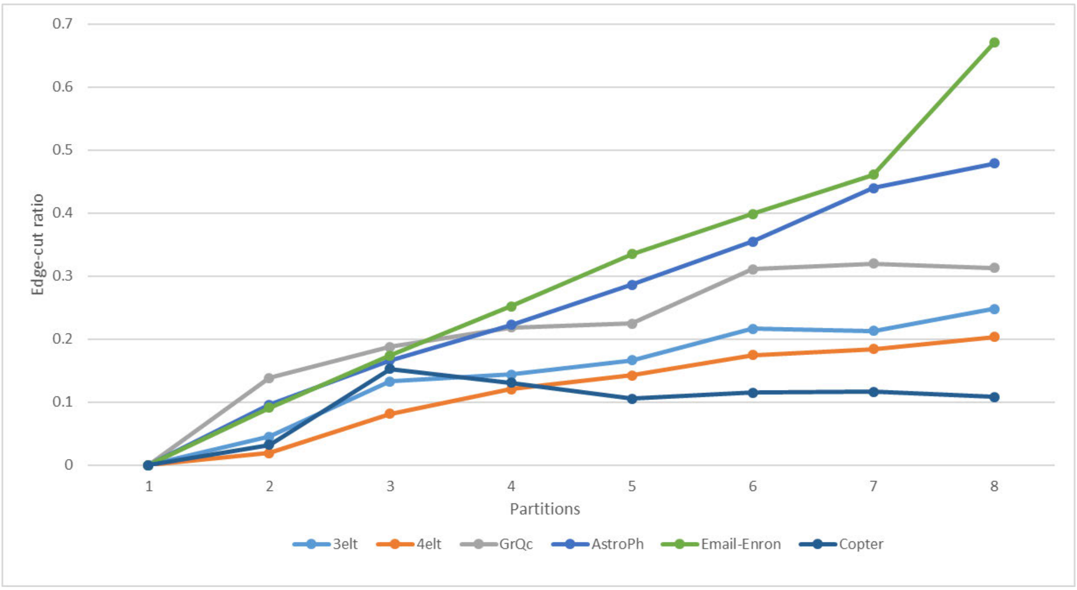
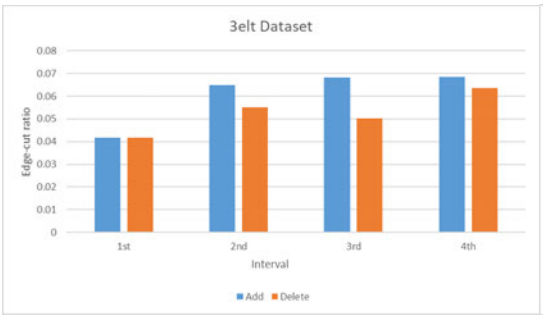
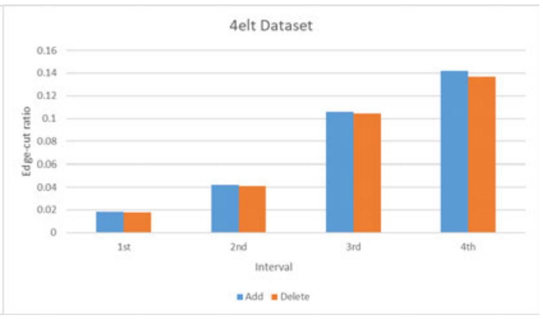


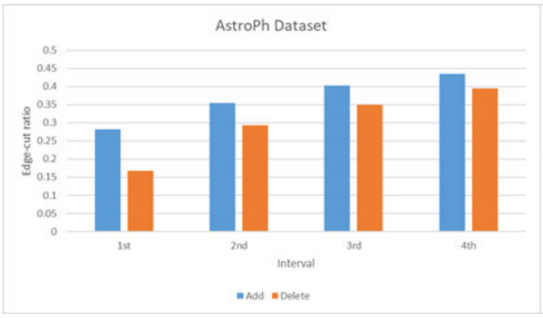
FIGURE 6.6: Impact of the Number of Partitions



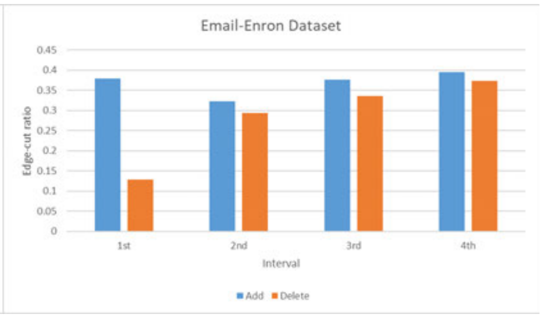
(A) 3elt Dataset



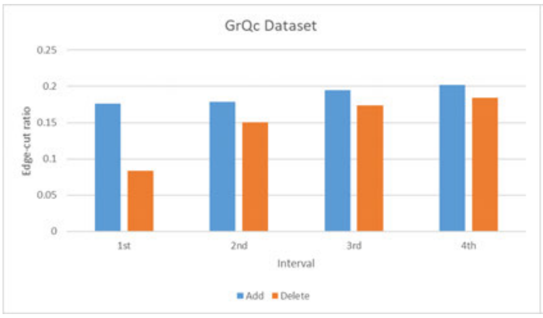
(B) 4elt Dataset



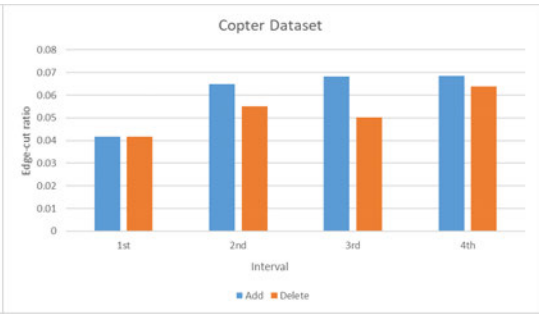
(C) AstroPh Dataset



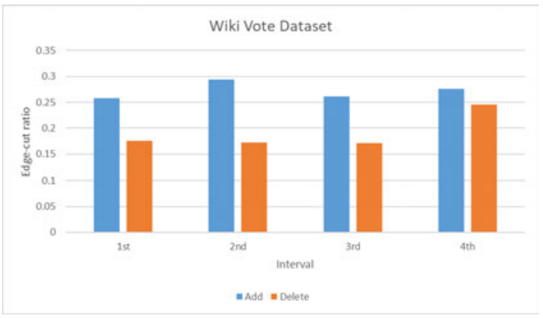
(D) Email-Enron Dataset



(E) GrQc Dataset



(F) Copter Dataset



(G) Wiki-Vote Dataset

FIGURE 6.5: Impact of addition/deletion with edge-cut

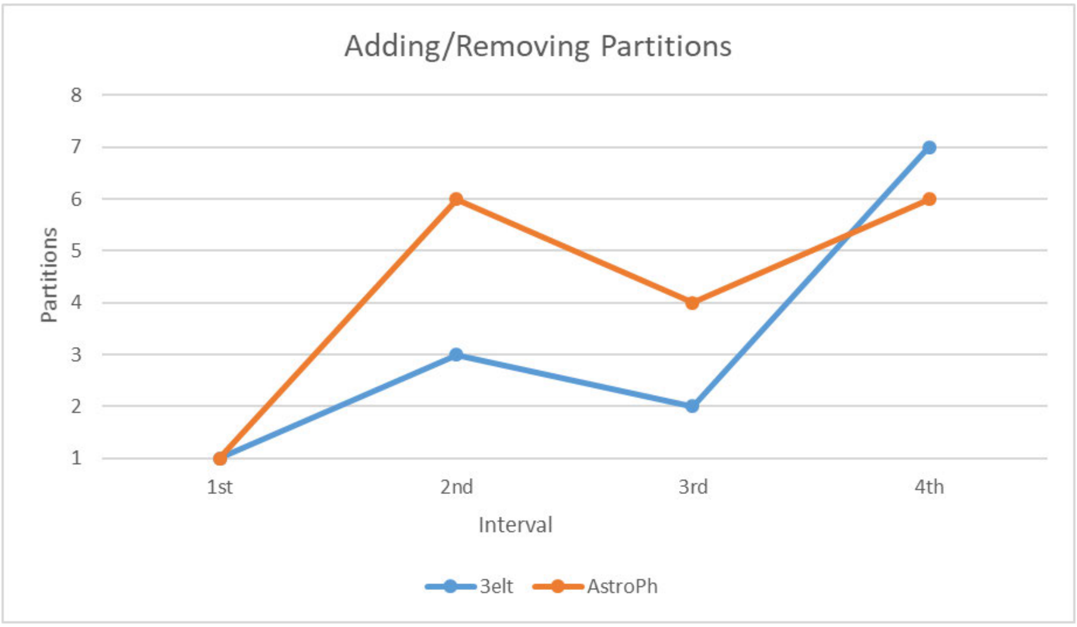


FIGURE 6.7: Adding/Removing partitions

6.9.5 Impact of Adding/Removing Partitions

As shown in Figure 6.7, a number of machines is being added and removed over time for the 3elt, AstroPh, and GrQc datasets. As per the demand of computational load, our partitioning method keeps adding and removing the machines based on the criteria explained in Section 6.7.3.

6.9.6 Time Comparison

In this section, we discuss the execution time for completing the partitioning task. We calculated the time from the beginning of the algorithm to the end of the execution of a dataset. Figure 6.8 shows the streaming execution time and, it includes the partitioning and input receiving time, because our algorithm does the partitioning task while receiving the input.

Figure 6.8 shows that our algorithm significantly reduces the execution time over the previous algorithms for most of the datasets except 3elt and GrQC.

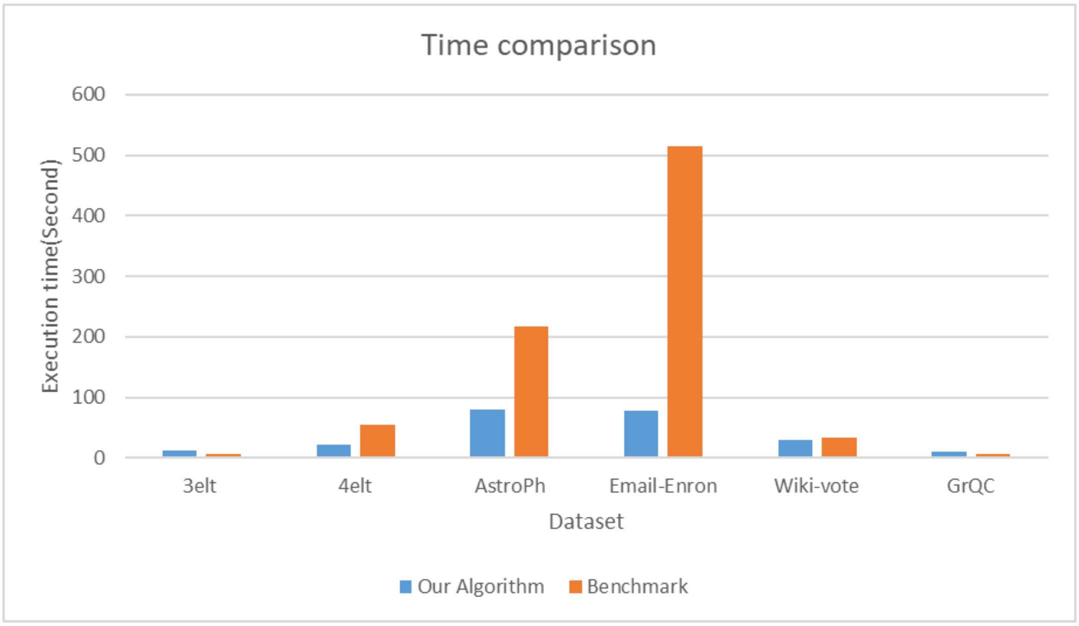


FIGURE 6.8: Execution time Comparison

6.10 Summary

In this chapter, we proposed a partitioning technique of a dynamic graph in a streaming manner. The study demonstrated a substantial improvement in reducing the edge-cut ratio for all the datasets. It also shows excellent performance in reducing the load imbalance in most of the datasets. A communication-aware balancing strategy to balance the computational load among the partitions was suggested. A dynamic auto-scaling method was employed in this study to provision and deprovision the cloud resources as per the demand of the computational load in a real-time manner. We evaluated the dynamic algorithm in a homogeneous cloud environment.

Chapter 7

Conclusion

This chapter discusses the objectives and overall contribution of this dissertation. Our three major findings and their significance are discussed in this chapter. We also discuss the future directions which we identified from this study.

7.1 Summary of This Dissertation

In this dissertation, we studied the problem of streaming graph partitioning with dynamic graph applications in the heterogeneous cloud environment. We answered the following research questions in this study:

- An auto-scaling method is proposed in order to optimise cloud resources and cost for streaming graph partitioning. The scaling algorithm scales the resources based on the upcoming streaming graph data and their computational load.

- A streaming graph partitioning algorithm is proposed for static datasets and applied to the PageRank application. The algorithm considered adding the data to the machines as the streaming graph data arrives, the deletion of data from the machine was not considered. A significant improvement is achieved in terms of minimising the communication and reducing the load imbalance as much as possible compared with the previous algorithm.

- A dynamic graph partitioning algorithm is proposed in a streaming manner with the aim of minimising inter-machine communication and balancing the computational load in distributed graph-oriented applications. An auto-scaling method is exploited in order to scale the number of cloud machines as per the demands of the computational load of a graph application. We used the homogenous cloud environment.

This dissertation contains three findings. We began with the resource utilisation and cost optimisation of cloud resources in a streaming graph partitioning algorithm. We determined that the number of machines and the type of machine depends on the upcoming streaming graph data in a dynamic graph-oriented application. We evaluated the graph partitioning technique with a proposed auto-scaling algorithm. We compared the algorithm with an existing algorithm and we observed that our algorithm outperformed in utilising the cloud resources and cost optimisation. In Chapter 4, we answered this research question by using an auto-scaling and cost optimisation algorithm.

A window-based streaming graph partitioning is proposed to minimise communication and to balance the load in the cloud environment. In Chapter 5, we proposed the WStream algorithm based on a stream window to look into the problem of streaming partitioning or one-pass partitioning. We used a stream window to accommodate more information of a graph for efficient partitioning in a one-pass manner. We also used a balancing strategy in order to balance the partitions while minimising the cut edges. This study outperforms to a certain degree to minimise the communication and to balance the computational load.

A scalable dynamic graph partitioner is proposed in the cloud environment. The proposed technique receives the input in a streaming manner for the dynamic dataset. The proposed partitioning algorithm allocates the vertices in such a way that minimises communication and balances the load as much as possible. We used a communication aware balancing strategy which optimises the load, based on the number of outgoing links of a vertex in a partition. In this scalable partitioner, we also proposed a scaling method which scales the number of machines as per the demands of computational load. The proposed algorithm outperformed the existing algorithms in terms of reducing the number of communications between partitions.

7.2 Future Direction

This research can extend in the following directions:

- is it possible to repartition the graph in real-time based on the changed behaviour and characteristics of vertices and edges? Rather than repartitioning the whole graph, it is possible that an updated part of the graph can be adapted among the partitions, with the aim to optimising the communication and balancing the load.
- A domain-specific dynamic graph partitioning framework which acts according to the applications and the characteristics of vertices and edges is required. Since the graph structure and their vertices' behaviour are application dependent, a graph partitioner would be able to act as per the demands of a particular application.
- How can cloud resources and cost be optimised based on the specific applications and their processing demands? The graph partitioner and resource scaling algorithm will work based on the particular application to scale the resources.

There are possibilities to extend this research with a few cross-domains. We will extend this dynamic partitioning problem with more real-world domains, for example, Blockchain, IoT application, and Sensor application.

7.2.1 Dynamic graph in Blockchain:

In a decentralised BlockChain application, the blocks and transactions are rapidly increasing in a Blockchain application. How

7.2.2 Community detection

In a social network, community detection is one of the most important applications to cluster or group similar data into a formal group. How can the updated graph be regrouped efficiently if any of the vertices' and edges' behaviour changes?

7.2.3 Streaming graph analysis in IoT

In the IoT environment, each device can be considered as a vertex and its connection with the other devices represents the edges. How can we reduce the communication between devices in an ever-changing number of devices in the IoT environment.

7.2.4 Dynamic Load Balancing in Sensor Network

Huge sensor data, which are being updated continuously, are available from the different sensor applications. How can we balance the ever-changing load between sensor devices in the IoT application to provide an efficient sensor application?

7.2.5 Streaming graph analysis in machine learning application

It would be interesting to look at the problem of machine learning applications with the stream of graph data. Networked data is always interdependent with each node and that makes it challenging to handle and maintain their relationship in an ever-growing stream of data. Most of the machine learning applications use the static data to make decisions. How can the machine learning application be adapted to a streaming graph and provide an efficient graph-oriented machine learning applications.

7.2.6 Exploring Memory Bottleneck and Optimisation

In graph partitioning, how much memory are being consuming in distributed environment and how much memory are being consumed of each machine during partitioning and also it would be interesting to look into the problem memory consumption dynamically.

Bibliography

[1] Finding the right number of servers in real-world queuing systems. *Interfaces*, 18(2):94–104, April 1988.

[2] Nectar cloud - nectar. https://nectar.org.au/research-cloud/, 2017. (Accessed on 15/12/2017).

[3] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. Streaming graph partitioning: An experimental study. *Proc. VLDB Endow.*, 11(11):1590–1603, July 2018.

[4] A. Abdolrashidi and L. Ramaswamy. Continual and cost-effective partitioning of dynamic graphs for optimizing big graph processing systems. In *2016 IEEE International Congress on Big Data (BigData Congress)*, pages 18–25, 2016.

[5] Amine Abou-Rjeili and George Karypis. Multilevel algorithms for partitioning power-law graphs. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 124–124, Washington, DC, USA, 2006. IEEE Computer Society.

[6] Lada A. Adamic, Rajan M. Lukose, Amit R. Puniyani, and Bernardo A. Huberman. Search in power-law networks. *Phys. Rev. E*, 64:046135, Sep 2001.

[7] Amr Ahmed, Nino Shervashidze, Shravan Narayanamurthy, Vanja Josifovski, and Alexander J. Smola. Distributed large-scale natural graph factorization. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 37–48, New York, NY, USA, 2013. ACM.

[8] Kook Jin Ahn and Sudipto Guha. Graph sparsification in the semi-streaming model. In *Proceedings of the 36th Internatilonal Collogquium on Automata, Languages and Programming: Part II*, ICALP '09, pages 328–338, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 120–124, New York, NY, USA, 2004. ACM.

[10] D. A. Bader and K. Madduri. Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12, 2008.

[11] Anshuman Biswas, Shikharesh Majumdar, Biswajit Nandy, and Ali El-Haraki. A hybrid auto-scaling technique for clouds processing applications with service level agreements. *J. Cloud Comput.*, 6(1):100:1–100:22, December 2017.

[12] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1):107 – 117, 1998. Proceedings of the Seventh International World Wide Web Conference.

[13] T. N. Bui, S. Chaudhuri, F. T. Leighton, and M. Sipser. Graph bisection algorithms with good average case behavior. *Combinatorica*, 7(2):171–191, Jun 1987.

[14] Thang Nguyen Bui and Byung Ro Moon. Genetic algorithm and graph partitioning. *IEEE Trans. Comput.*, 45(7):841–855, July 1996.

[15] Rajkumar Buyya, Christian Vecchiola, and S. Thamarai Selvi. *Mastering Cloud Computing: Foundations and Applications Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[16] Ümit V. Çatalyürek, Bora Uçar, and Cevdet Aykanat. *Hypergraph Partitioning*, pages 871–881. Springer US, Boston, MA, 2011.

[17] A. Cevahir, C. Aykanat, A. Turk, and B. B. Cambazoglu. Site-based partitioning and repartitioning techniques for parallel pagerank computation. *IEEE Transactions on Parallel and Distributed Systems*, 22(5):786–802, May 2011.

[18] S. Chaisiri, B. Lee, and D. Niyato. Optimization of resource provisioning cost in cloud computing. *IEEE Transactions on Services Computing*, 5(2):164–177, April 2012.

[19] Min Chen, Shiwen Mao, and Yunhao Liu. Big data: A survey. *Mobile Networks and Applications*, 19(2):171–209, Apr 2014.

[20] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 1:1–1:15, New York, NY, USA, 2015. ACM.

[21] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 85–98, New York, NY, USA, 2012. ACM.

[22] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 219–228, New York, NY, USA, 2009. ACM.

[23] Dong Dai, Wei Zhang, and Yong Chen. Iogp: An incremental online graph partitioning algorithm for distributed graph databases. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '17, pages 219–230, New York, NY, USA, 2017. ACM.

[24] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting*

Systems Design & Implementation - Volume 6, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[25] Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Algorithms and theory of computation handbook. chapter Dynamic Graph Algorithms, pages 9–9. Chapman & Hall/CRC, 2010.

[26] A. E. Dunlop and B. W. Kernighan. A procedure for placement of standard-cell vlsi circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 4(1):92–98, January 1985.

[27] Quang Duong, Sharad Goel, Jake Hofman, and Sergei Vassilvitskii. Sharding social networks. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 223–232, New York, NY, USA, 2013. ACM.

[28] G. Echbarthi and H. Kheddouci. Fractional greedy and partial restreaming partitioning: New methods for massive graph partitioning. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 25–32, Oct 2014.

[29] Ghizlane Echbarthi and Hamamache Kheddouci. Streaming metis partitioning. In *Proceedings of the 2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, ASONAM '16, pages 17–24, Piscataway, NJ, USA, 2016. IEEE Press.

[30] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *19th Design Automation Conference*, pages 175–181, June 1982.

[31] I. Filippidou and Y. Kotidis. Online and on-demand partitioning of streaming graphs. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 4–13, Oct 2015.

[32] Hugo Firth and Paolo Missier. Workload-aware streaming graph partitioning.

[33] Hugo Firth and Paolo Missier. Workload-aware streaming graph partitioning. In *EDBT/ICDT Workshops*, 2016.

[34] Hugo Firth and Paolo Missier. Taper: Query-aware, partition-enhancement for large, heterogenous graphs. *Distrib. Parallel Databases*, 35(2):85–115, June 2017.

[35] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, pages 47–63, New York, NY, USA, 1974. ACM.

[36] Casey Battaglino Georgia and Richard Vuduc Georgia. Grasp : Distributed streaming graph partitioning. 2015.

[37] J. R. Gilbert, G. L. Miller, and Shang-Hua Teng. Geometric mesh partitioning: implementation and experiments. In *Proceedings of 9th International Parallel Processing Symposium*, pages 418–427, April 1995.

[38] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[39] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 599–613, Berkeley, CA, USA, 2014. USENIX Association.

[40] H. Goudarzi, M. Ghasemazar, and M. Pedram. Sla-based optimization of power and migration cost in cloud computing. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 172–179, May 2012.

[41] John Greiner and Guy E. Blelloch. High performance computing. chapter Connected Components Algorithms, pages 155–184. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[42] Alessio Guerrieri and Alberto Montresor. Distributed edge partitioning for graph processing. *CoRR*, abs/1403.6270, 2014.

[43] Yong Guo, Sungpack Hong, Hassan Chafi, Alexandru Iosup, and Dick Epema. Modeling, analysis, and experimental comparison of streaming graph-partitioning policies. *Journal of Parallel and Distributed Computing*, 108:106 – 121, 2017. Special Issue on Scalable Computing Systems for Big Data Applications.

[44] R. Han, L. Guo, M. M. Ghanem, and Y. Guo. Lightweight resource scaling for cloud applications. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 644–651, May 2012.

[45] Vitali Henne, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, and Christian Schulz. n-level hypergraph partitioning. *ArXiv*, abs/1505.00693, 2015.

[46] Desmond J. Higham, Gabriela Kalna, and Milla Kibble. Spectral clustering and its use in bioinformatics. *Journal of Computational and Applied Mathematics*, 204(1):25 – 37, 2007. Special issue dedicated to Professor Shinnosuke Oharu on the occasion of his 65th birthday.

[47] M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a scalable high quality graph partitioner. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, April 2010.

[48] Jiewen Huang and Daniel J. Abadi. Leopard: Lightweight edge-oriented partitioning and replication for dynamic graphs. *Proc. VLDB Endow.*, 9(7):540–551, March 2016.

[49] Y. Ishizuka, W. Chen, and I. Paik. Workflow transformation for real-time big data processing. In *2016 IEEE International Congress on Big Data (BigData Congress)*, pages 315–318, June 2016.

[50] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving graph processing at scale. In *Proceedings of the Fourth International*

Workshop on Graph Data Management Experiences and Systems, GRADES '16, pages 5:1–5:6, New York, NY, USA, 2016. ACM.

[51] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[52] Nilesh Jain, Guangdeng Liao, and Theodore L. Willke. Graphbuilder: Scalable graph etl framework. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 4:1–4:6, New York, NY, USA, 2013. ACM.

[53] Otis B. Jennings, Avishai Mandelbaum, William A. Massey, and Ward Whitt. Server staffing to meet time-varying demand. *Manage. Sci.*, 42(10):1383–1394, October 1996.

[54] J. Jiang, J. Lu, G. Zhang, and G. Long. Optimal cloud resource auto-scaling for web applications. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 58–65, May 2013.

[55] W. Jiang, J. Qi, J. X. Yu, J. Huang, and R. Zhang. Hyperx: A scalable hypergraph framework. *IEEE Transactions on Knowledge and Data Engineering*, 31(5):909–922, May 2019.

[56] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 28–28, Nov 1998.

[57] George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society.

[58] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[59] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96 – 129, 1998.

[60] George Karypis and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71 – 95, 1998.

[61] Matthias Keller and Holger Karl. Response time-optimized distributed cloud resource allocation. In *Proceedings of the 2014 ACM SIGCOMM Workshop on Distributed Cloud Computing*, DCC '14, pages 47–52, New York, NY, USA, 2014. ACM.

[62] Brian W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49:291–307, 1970 1970.

[63] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 169–182, New York, NY, USA, 2013. ACM.

[64] Scott Kirkpatrick, Charles Daniel Gelatt, and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220 4598:671–80, 1983.

[65] Shad Kirmani and Padma Raghavan. Scalable parallel graph partitioning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 51:1–51:10, New York, NY, USA, 2013. ACM.

[66] S. Koranne. A distributed algorithm for k-way graph partitioning. In *Proceedings 25th EUROMICRO Conference. Informatics: Theory and Practice for the New Millennium*, volume 2, pages 446–448 vol.2, Sep. 1999.

[67] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[68] Alan G. Labouseur, Paul W. Olsen, and Jeong hyon Hwang. Scalable and robust management of dynamic graph data, 2013.

[69] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, pages 177–187, New York, NY, USA, 2005. ACM.

[70] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, 2014.

[71] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.

[72] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[73] Daniel Margo and Margo Seltzer. A scalable distributed graph partitioner. *Proc. VLDB Endow.*, 8(12):1478–1489, August 2015.

[74] C. Martella, D. Logothetis, A. Loukas, and G. Siganos. Spinner: Scalable graph partitioning in the cloud. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1083–1094, April 2017.

[75] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2):25:1–25:39, 2015.

[76] H. Meyerhenke, P. Sanders, and C. Schulz. Parallel graph partitioning for complex networks. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 1055–1064, May 2015.

[77] Jayanta Mondal and Amol Deshpande. Managing large dynamic graphs efficiently. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 145–156, New York, NY, USA, 2012. ACM.

[78] M. E. J. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, 2006.

[79] M. E. J. Newman and Juyong Park. Why social networks are different from other types of networks. *Physical review. E, Statistical, nonlinear, and soft matter physics*, 68 3 Pt 2:036122, 2003.

[80] M.E.J. Newman. The structure and function of networks. *Computer Physics Communications*, 147(1):40 – 45, 2002. Proceedings of the Europhysics Conference on Computational Physics Computational Modeling and Simulation of Complex Systems.

[81] Joel Nishimura and Johan Ugander. Restreaming graph partitioning: Simple versatile algorithms for advanced balancing. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1106–1114, New York, NY, USA, 2013. ACM.

[82] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. In *Proceedings of the 7th International World Wide Web Conference*, pages 161–172, Brisbane, Australia, 1998.

[83] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. Hdrf: Stream-based partitioning for power-law graphs. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, pages 243–252, New York, NY, USA, 2015. ACM.

[84] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine(s) that could: Scaling online social networks. *IEEE/ACM Transactions on Networking*, 20(4):1162–1175, Aug 2012.

[85] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical review. E, Statistical, nonlinear, and soft matter physics*, 76 3 Pt 2:036106, 2007.

[86] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi. Ja-be-ja: A distributed algorithm for balanced graph partitioning. In *2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*, pages 51–60, Sept 2013.

[87] Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, Mark Jelasity, and Seif Haridi. A distributed algorithm for large-scale graph partitioning. *ACM Trans. Auton. Adapt. Syst.*, 10(2):12:1–12:24, June 2015.

[88] Jason Riedy and David A. Bader. Massive streaming data analytics: A graph-based approach. *XRDS*, 19(3):37–43, 2013.

[89] Maria Alejandra Rodriguez and Rajkumar Buyya. Containers orchestration with cost-efficient autoscaling in cloud computing environments. *ArXiv*, abs/1812.00300, 2018.

[90] N. Roy, A. Dubey, and A. Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 500–507, July 2011.

[91] H. P. Sajjad, A. H. Payberah, F. Rahimian, V. Vlassov, and S. Haridi. Boosting vertex-cut partitioning for streaming graphs. In *2016 IEEE International Congress on Big Data (BigData Congress)*, pages 1–8, June 2016.

[92] Sherif Sakr, Faisal Moeen Orakzai, Ibrahim Abdelaziz, and Zuhair Khayyat. *Large-Scale Graph Processing Using Apache Giraph*. Springer Publishing Company, Incorporated, 1st edition, 2017.

[93] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 22:1–22:12, New York, NY, USA, 2013. ACM.

[94] Peter Sanders and Christian Schulz. Distributed evolutionary graph partitioning. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*,

ALENEX '12, pages 16–29, Philadelphia, PA, USA, 2012. Society for Industrial and Applied Mathematics.

[95] John E. Savage and Markus G. Wloka. Parallelism in graph-partitioning. *Journal of Parallel and Distributed Computing*, 13(3):257 – 272, 1991.

[96] Kirk Schloegel, George Karypis, and Vipin Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *J. Parallel Distrib. Comput.*, 47(2):109–124, December 1997.

[97] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning. In Arndt Bode, Thomas Ludwig, Wolfgang Karl, and Roland Wismüller, editors, *Euro-Par 2000 Parallel Processing*, pages 296–310, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[98] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning (distinguished paper). In *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, Euro-Par '00, pages 296–310, Berlin, Heidelberg, 2000. Springer-Verlag.

[99] Kirk Schloegel, George Karypis, and Vipin Kumar. Graph partitioning for dynamic, adaptive and multi-phase scientific simulations. *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 271–273, 2001.

[100] A. Sen, H. Deng, and S. Guha. On a graph partitioning problem with applications to vlsi layout. In *1991., IEEE International Sympoisum on Circuits and Systems*, pages 2846–2849 vol.5, June 1991.

[101] S. Seo, E. J. Yoon, J. Kim, S. Jin, J. Kim, and S. Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 721–726, Nov 2010.

[102] Z. Shang and J. X. Yu. Catch the wind: Graph workload balancing on cloud. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 553–564, April 2013.

[103] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, NY, USA, 2013. ACM.

[104] J. Shi, F. Dong, J. Zhang, J. Jin, and J. Luo. Resource provisioning optimization for service hosting on cloud platform. In *2016 IEEE 20th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 340–345, May 2016.

[105] Zhan Shi, Junhao Li, Pengfei Guo, Shuangshuang Li, Dan Feng, and Yi Su. Partitioning dynamic graph asynchronously with distributed fennel. *Future Gener. Comput. Syst.*, 71(C):32–42, June 2017.

[106] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, New York, NY, USA, 2013. ACM.

[107] Kamran Siddique, Zahid Akhtar, Yangwoo Kim, Young-Sik Jeong, and Edward J. Yoon. Investigating apache hama: A bulk synchronous parallel computing framework. *J. Supercomput.*, 73(9):4190–4205, September 2017.

[108] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri. Partitioning trillion-edge graphs in minutes. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 646–655, May 2017.

[109] Isabelle Stanton. Streaming balanced graph partitioning algorithms for random graphs. In *Proceedings of the Twenty-fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, pages 1287–1301, Philadelphia, PA, USA, 2014. Society for Industrial and Applied Mathematics.

[110] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 1222–1230, New York, NY, USA, 2012. ACM.

[111] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From "think like a vertex" to "think like a graph". *Proc. VLDB Endow.*, 7(3):193–204, November 2013.

[112] Charalampos Tsourakakis. Streaming graph partitioning in the planted partition model. In *Proceedings of the 2015 ACM on Conference on Online Social Networks*, COSN '15, pages 27–35, New York, NY, USA, 2015. ACM.

[113] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM '14, pages 333–342, New York, NY, USA, 2014. ACM.

[114] Johan Ugander and Lars Backstrom. Balanced label propagation for partitioning massive graphs. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 507–516, New York, NY, USA, 2013. ACM.

[115] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The anatomy of the facebook social graph. *CoRR*, abs/1111.4503, 2011.

[116] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[117] Luis Vaquero, Félix Cuadrado, Dionysios Logothetis, and Claudio Martella. Adaptive partitioning for large-scale dynamic graphs. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 35:1–35:2, New York, NY, USA, 2013. ACM.

[118] Luis M. Vaquero, Félix Cuadrado, Dionysios Logothetis, and Claudio Martella. xdgp: A dynamic graph processing system with adaptive partitioning. *CoRR*, abs/1309.1049, 2013.

[119] C. Walshaw, M. Cross, and M.G. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 47(2):102 – 108, 1997.

[120] C. Walshaw, M. Cross, and M.G. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *J. Parallel Distrib. Comput.*, 47(2):102–108, December 1997.

[121] Christopher Walshaw. The graph partitioning archive. http://http://chriswalshaw.co.uk/partition/, 2016.

[122] L. Wang, Y. Xiao, B. Shao, and H. Wang. How to partition a billion-node graph. In *2014 IEEE 30th International Conference on Data Engineering*, pages 568–579, March 2014.

[123] R. Wang and K. Chiu. A stream partitioning approach to processing large scale distributed graph datasets. In *2013 IEEE International Conference on Big Data*, pages 537–542, 2013.

[124] Ward Whitt. Partitioning customers into service groups. *Management Science*, 45(11):1579–1592, 1999.

[125] Roy D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Pract. Exper.*, 3(5):457–481, October 1991.

[126] Cong Xie, Wu-Jun Li, and Zhihua Zhang. S-powergraph: Streaming graph partitioning for natural graphs by vertex-cut. *CoRR*, abs/1511.02586, 2015.

[127] Cong Xie, Ling Yan, Wu jun Li, and Zhihua Zhang. Distributed power-law graph computing: Theoretical and empirical analysis.

[128] Ning Xu, Lei Chen, and Bin Cui. Loggp: A log-based dynamic graph partitioning method. *Proc. VLDB Endow.*, 7(14):1917–1928, October 2014.

[129] Shengqi Yang, Xifeng Yan, Bo Zong, and Arijit Khan. Towards effective partition management for large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 517–528, New York, NY, USA, 2012. ACM.

[130] Chunxing Yin, Jason Riedy, and David A. Bader. A new algorithmic model for graph analysis of streaming data. In *Proceedings of the 14th International Workshop on Mining and Learning with Graphs (MLG)*, May 2018.

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

[132] W. Zhang, Y. Chen, and D. Dai. Akin: A streaming graph partitioning algorithm for distributed graph storage systems. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 183–192, May 2018.

[133] B. Zheng, H. Su, W. Hua, K. Zheng, X. Zhou, and G. Li. Efficient clue-based route search on road networks. *IEEE Transactions on Knowledge and Data Engineering*, 29(9):1846–1859, Sept 2017.

[134] J. Y. Zien, M. D. F. Schlag, and P. K. Chan. Multilevel spectral hypergraph partitioning with arbitrary vertex sizes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(9):1389–1399, Sep. 1999.