DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING, SECOND CYCLE, 30 CREDITS

STOCKHOLM, SWEDEN 2016

Streaming Graph Partitioning

DEGREE PROJECT IN DISTRIBUTED COMPUTING AT KTH INFORMATION AND COMMUNICATION TECHNOLOGY

ZAINAB ABBAS

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION

TRITA TRITA-ICT-EX-2016:121

KTH Information and
Communication Technology

KTH Royal Institute of Technology

Dept. of Software and Computer Systems

Degree Project in Distributed Computing

Streaming Graph Partitioning

Author: Zainab Abbas
Supervisors: Vasiliki Kalavri
Paris Carbone

Examiner: Prof. Vladimir Vlassov, KTH, Sweden

Abstract

Graph partitioning is considered to be a standard solution to process huge graphs efficiently when processing them on a single machine becomes inefficient due to its limited computation power and storage space. In graph partitioning, the whole graph is divided among different computing nodes that process the graph in parallel. During the early stages of research done on graph partitioning, different offline partitioning methods were introduced; these methods create high computation cost as they process the whole graph prior to partitioning. Therefore, an online graph partitioning method called as *streaming graph partitioning* was introduced later to reduce the computation cost by assigning the edges or vertices on-the-fly to the computing nodes without processing the graph before partitioning.

In our thesis, we presented an experimental study of different streaming graph partitioning methods that use two partitioning techniques: vertex partitioning and edge partitioning. Edge partitioning has proved good for partitioning highly skewed graphs. After implementing different partitioning methods, we have proposed a partitioning algorithm that uses degree information of the vertices. Furthermore, we measured the effect of different partitioning methods on the graph stream processing algorithms.

Our results show that for vertex partitioning Fennel has performed better than Linear Greedy as it shows lower edge-cuts and better load balancing. Moreover, for edge partitioning, the Degree based partitioner has performed better than Least Cost Incremental and Least Cost Incremental Advanced in reducing the replication factor, but the Degree based partitioner does not do well in load balancing. In the end, we show that the custom partitioning methods, compared to default hash partitioning, save the memory space by reducing the size of aggregate states during execution of different graph processing algorithms on the resulting partitions. The Degree based partitioner performed well by reducing the size of aggregate states on average up to 50%. Other algorithms include: Fennel, Linear Greedy, Least Cost Incremental and Least Cost Incremental Advanced, they reduced the size of aggregate states on average up to 21%, 10%, 27% and 48%.

Referat

Grafpartitionering anses vara en standardlösning för att effektivt bearbeta stora grafer, när behandling av dem på en enda maskin blir ineffektiv på grund av dess begränsade beräkningskraft och lagringsutrymme. I grafpartitionering är hela grafen delad mellan olika beräkningsnoder som bearbetar grafen parallellt. Under de tidiga stadierna av forskning gjord på grafpartitionering har olika offline partitioneringsmetoder introducerats; dessa metoder skapar höga beräkningskostnadner eftersom de behandlar hela grafen före uppdelning. Därför introducerades senare en online graffördelningsmetod som kallas streaming graph partitioning för att minska beräkningskostnaden genom att tilldela kanterna eller hörnen under processen till beräkningsnoder utan att bearbeta grafen före partitionering.

I vår uppsats presenterade vi en experimentell studie av olika strömmande grafpartitioneringsmetoder som använder två uppdelningstekniker: hörnpartitionering och kantpartitionering. Kantpartitionering har visat sig vara bra för uppdelning av mycket skeva grafer. Efter genomförandet av olika partitioneringsmetoder, har vi föreslagit en partitioneringsalgoritm som använder gradinformationen från hörnen. Dessutom mätte vi effekten av olika partitioneringsmetoder i graph stream processing-algoritmerna.

Våra resultat visar att Fennel presterade bättre än Linear Greedy för hörnpartitionering eftersom den visar lägre kantavskärning och bättre lastbalansering. Dessutom för kantpartitionering, den stegbaserade partitioneringen presterade bättre än Least Cost Incremental och Least Cost Incremental Advanced att minska replikationsfaktorn, men stegbaserade partitioneringen hanterar inte lastbalansering så bra. I slutändan, visar vi att de anpassade partitioneringsmetoder, jämfört med standard hash partitionering, sparar minnesutrymme genom att minska storleken av aggregerade tillstånd under utförande av olika grafalgoritmer på de resulterande partitionerna. Stegbaserade partitioneringen presterade väl genom att minska storleken av aggregerade tillstånd i genomsnitt upp till 50%. Andra algoritmer inkluderar: Fennel, Linear Greedy, Least Cost Incremental och Least Cost Incremental Advanced. De minskade storleken på aggregerade tillståndet med i genomsnitt upp till 21%, 10%, 27% och 48%.

Acknowledgment

I am very thankful to all the great people who have been helpful to me during my thesis.

First of all I thank my supervisors Vasiliki Kalavri and Paris Carbone for helping, guiding and motivating me throughout the project. They helped a lot in solving my issues whenever I was stuck. It has indeed been a great experience to work with them.

Secondly, to my great EMDC colleagues, Ashansa Perera, Shelan Perera and another friend Riccardo for being there every day at work to instantly help and advice in case of need. They have kept the working atmosphere friendly and entertaining.

Lastly, to my family for supporting me and trusting me with whatever I wanted to do with my life.

Stockholm, 24 July 2016

Zainab Abbas

Contents

List of Figures

List of Tables

Chapter 1

1 Introduction

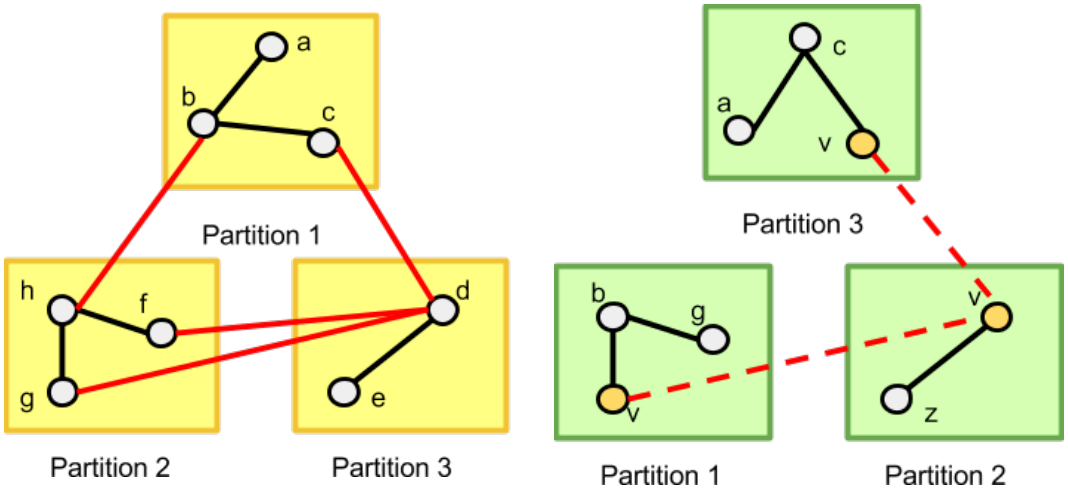
1.1 Problem Statement

Almost every data can be represented in the form of entities and relationships between them. Graphs can be used to represent such entities and relations in the form of vertices and edges. Nowadays, the graphs are increasing in size. For example, the Web graph having 4.75 billion indexed pages [25] and Facebook having 1.65 billion monthly active users [26]. Therefore, it is inefficient to process huge graphs that contain billions of edges and vertices, or even more, on a single machine because of the limited memory and computation power. A way to solve this is to partition the graph across multiple machines and use distributed graph processing algorithms.

Google's Pregel [1] based on Bulk Synchronous Parallel (BSP) model [2] and vertex centric approach was introduced for large-scale graph processing; it supports iterative computations which are required by many graph processing algorithms. Other frameworks for distributed graph computation like Apache Giraph [3] and PEGASUS [5] also emerged. These systems use a hash partitioner, that computes the hash of the vertex ID (a unique number to identify the vertex) and uses it for splitting the graph among different partitions, which usually ends up in randomly divided vertices. This partitioning method does not take into account the graph structure, so it has large chances of placing the neighboring vertices in different partitions. Therefore, in case where the neighboring vertices need to communicate with each other, this placement can cause an increase in the communication cost if the vertices are placed in different partitions. Hence, this gives the motivation for creating better partitioning methods.

Graph partitioning can be done using two techniques. One of this technique is vertex partitioning. It refers to dividing the vertices across different partitions, which might result in placing two neighboring vertices, having an edge between them, into different partitions. This edge between two partitions is called an *edge-cut*, as shown in Figure 1.1(a). The Hash partitioner discussed in the previous paragraph, might result in a large number of edge-cuts due to the fact that it does not take into account

the graph structure, which results in increase of the communication cost across the partitions. Hence, a better approach is needed. Another relatively new technique is known as edge partitioning, which instead of the vertices, divides the edges in to different partitions. As a result, if a vertex appears in more than one partition, then this forms a *vertex-cut*, as shown in Figure 1.1(b).



(a) Vertex Partitioning and Edge-Cut

(b) Edge Partitioning and Vertex-Cut

Figure 1.1: Different Partitioning Techniques

Good graph partitioners have different objectives like balancing the load across the partitions and reducing the edge or vertex-cut. However, handling dynamic graphs is a challenge. Dynamic graphs are important as most of the social media graphs like Facebook and Twitter are dynamic, which means they are continuously updated. For example, when a user makes new friends and removes some friends on Facebook, the user actions trigger different events. A new approach known as streaming graph partitioning [6] can work with dynamic graphs, which includes reading a vertex or an edge of the whole graph one by one and assigning it to the partitions on-the-fly without knowing the whole state of the graph. Different streaming graph partitioning heuristics have recently been developed. Some of the popular ones are: Fennel [7], HDRF [8] and PowerGraph Greedy Algorithm [9].

Our work aims to, firstly, perform a detailed survey of streaming graph partitioners, secondly, implement some of the streaming graph partitioners and measure their partitioning quality. Lastly, based on the qualities of these partitioners, identify new partitioning functions that can have a better partitioning quality and performance than the former. Furthermore, study the partitioning functions for the effect of their partitioning on different graph stream algorithms. We implemented our work using

the Graph stream processing framework [10], which is build on top of Apache Flink [11].

1.2 Objective

The objectives of this thesis are as follows:

- To conduct a study of different streaming graph partitioning algorithms.
- To implement and compare different streaming graph partitioning algorithms using the Apache Flink graph streaming API [10].
- To improve the current partitioning techniques.
- To perform experimental analysis for different partitioning techniques and measure their effect on graph stream approximations.

1.3 Contribution

The main contributions of this thesis are as follows:

- We performed a detailed literature study of different streaming graph partitioning algorithms. The summary of these algorithms, along with their comparison tables, is given in section 2.4 of this thesis.
- To the best of our knowledge, some efficient streaming graph partitioning algorithms include: Linear Greedy [6], Fennel [7], Least Cost Incremental [12] and a variation of Least Cost Incremental [12]. We implemented them and tested them for verification purpose.
- We propose new partitioning function based on the degrees of vertices. The degree of a vertex is a global parameter, which is not known beforehand. Therefore, we use the evolving degree, which keeps updating as we process the vertices one-by-one, for partitioning.
- We evaluate the partitioning heuristics based on different metrics like the execution time, load balancing and the vertex-cut or the edge-cut. In addition to that, we analyse how the partitioning step improves the performance of graph stream processing algorithms.
- Our work is an open source contribution to the Flink Graph Streaming repository [10].

1.4 Methodology

This section summarises the scientific research method involved in our thesis work. We used the resources available to us during the whole process. We have briefly explained the observation, analysis, hypothesis, design, development and testing phases. Our research is based on empirical and mathematical methods to avoid subjectivity in the whole process.

1.4.1 Observation and Requirements Gathering

Our work is based on observation and experiment. During the literature review phase, we studied different algorithms for graph partitioning. This gave us an idea of what has been done so far for graph partitioning. Streaming graph partitioning is a quite new technique, therefore all the work done for it is recent. To the extent of our knowledge, we chose the most recent and efficient partitioning algorithms for implementation. Certain algorithms are based on mathematical models that require a deductive logic for the proof. Moreover, we also observed and tested the current partitioning approach for graph streams used by different Graph processing APIs in order to find out how we can improve it. This observation and testing helped us finding the problem. Our approach is based on both reason and research.

1.4.2 Design and Development

The literature study and testing, lead us to the design phase of the project. For the proof of concept, we implemented the existing streaming graph partitioning algorithms and compared them. The major challenge we faced was that there was no open source code for these partitioning algorithms. Therefore, we had to design methods and propose different data structures for implementing them. Firstly, we implemented them in Java for single-threaded implementation, and secondly, we ported them to the Apache Flink Graph streaming API for multi-threaded implementation. As a result, we came up with our own custom partitioner, having certain properties of the existing ones.

1.4.3 Testing and Evaluation

We performed our experiments with all the available resources, which include: the online available resources mentioned in the bibliography section, an open source Apache Flink API [11], and the cluster machines from our department. Furthermore, to achieve efficient results during the experiments, an isolated environment is maintained. Latest versions of the processing engine i.e Apache Flink is used for creating and running our tests to keep everything up-to-date. All the data set information is

included in the thesis for reproducibility. The input data sets used are generated from a very recent release of the Apache Flink Gelly API [11]. version: 1.1.

1.5 Structure of Thesis

After the Introduction in section 1, section 2 is about Graph partitioning, which explains different partitioning approaches and PowerLaw graphs [9]. In this section we discuss in detail different partitioning algorithms implemented in the thesis. This section includes the theoretical explanation and mathematical models of these partitioning algorithms along with their comparison.

Section 3 contains the literature review and background work. This section gives a good overview of streaming models and graph processing models along with references to the related work. Moreover, it also contains a detailed topic about Apache Flink explaining Gelly and the Flink Streaming API. Our Implementation details for porting these algorithms to Flink are discussed in section 4.

The experimental setup, tests, input data and output results are presented in section 5. Lastly, the conclusion of the thesis is presented in section 6, this also includes the future work.

Chapter 2
2 Graph Partitioning

Processing huge graphs on a single machine is inefficient due to the limited memory size and processing power. Therefore, one solution is that these graphs have to be partitioned across different computing nodes and processed in parallel. Exchanging large amount of data between these computing nodes is expensive, so it is important to reduce the communication between them. Good quality partitioning is achieved by focusing on two objectives: balancing the computation load among different computing nodes and reducing the communication cost between them. This problem of dividing the graph among different computing nodes keeping the communication cost minimum and balancing the load is called *balanced graph partitioning*. It is an NP-hard problem [12].

Assumptions: We have explained the communication cost with respect to the *message-passing model* [1] in which the vertices of the graph communicate by sending messages. This communication is not necessary for all graph processing algorithms as there are algorithms that do not require the vertices to communicate, but in order to explain the communication cost we consider the message-passing model.

2.1 Partitioning Techniques

There are two main approaches for graph partitioning, namely: *Edge partitioning* and *Vertex partitioning*. Different graph partitioning algorithms have been developed based on these approaches.

2.1.1 Vertex Partitioning

In vertex partitioning, the vertices of a graph are divided into k equal sized partitions such that the edges between the partitions are kept minimum. This is also referred as *edge-cut* partitioning.

Edge-Cut Definition: For a graph $G = (V, E)$, having E edges and V vertices; $E\backslash' E$ is the set of edges such that the graph $G' = (V, E\backslash E')$ is disconnected. Here, E' is the *edge-cut*.

For understanding the edge-cuts consider an input graph being partitioned, during partitioning if a vertex is assigned to one partition and its neighbor to another, then the edge between them forms an edge-cut. As can be seen in Figure 1.1(a) the vertices are placed in different partitions, and the lines between the partitions are the edge-cuts.

The aim of good vertex partitioning algorithms is to reduce the *edge-cuts* and balance the computation load. In a message-passing model the neighboring vertices of a graph communicate by sending messages. When the neighbors of a vertex belong to different partitions, then the cost of sending messages between the partitions is called the cut-cost. These messages can be sent with or without aggregation. Aggregating the messages means that different messages that are supposed to be sent to the neighbors present in a different partition are combined, and an aggregate of these messages is created. This aggregate message is later sent to the partition. As shown in Figure 2.1 the messages of vertex b and c are aggregated and sent to the vertex h in the other partition. Whereas, for no aggregation all the messages are sent separately to the neighbors belonging to the other partition; as in Figure 2.1 the vertices b and c send messages separately to the vertex h.

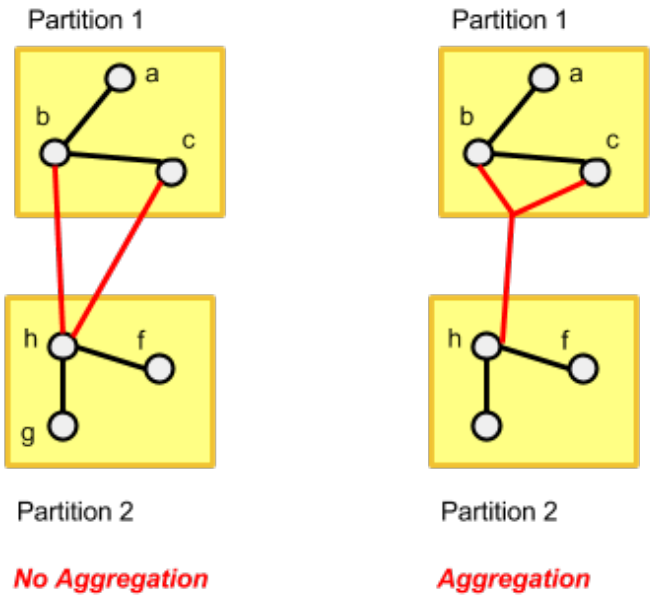


Figure 2.1: No Aggregation and Aggregation of Messages in Vertex Partitioning

In case of no aggregation, the cut-cost for a vertex to its neighbors is equal to the number of its neighbors placed in different partitions. In Figure 2.1 this cost is two for the vertex h, as the vertex h gets two messages from its neighbors b and c. However, in-case of aggregation, the cut-cost of for a vertex is equal to the number of partitions in which its neighbors are placed. In Figure 2.1 this cost is one as the neighbors of the vertex h are present in one partition, other than the partition where h is present.

If a graph is partitioned in a way that there are large number of edge-cuts, then this creates a lot of communication cost between the partitions due to the fact that large number of messages are exchanged between the partitions. Therefore, the aim of a good partitioner is to keep the edge-cuts minimum.

2.1.2 Edge Partitioning

A relatively new technique for graph partitioning was proposed in [9]; it is called *edge partitioning* [19]. In *edge partitioning* the edges of a graph are divided among k equal sized partitions such that the vertices that are cut between the partitions are kept minimum. This is also referred as *vertex-cut* partitioning.

Vertex-Cut Definition: For a graph $G = (V, E)$, having E edges and V vertices; $V \backslash' V$ is the set of vertices with E' set of edges incident to them, such that the graph $G' = (V \backslash V', E \backslash E')$ is disconnected. Here, V' is the *vertex-cut*.

Each edge contains two vertices, called the *end vertices*. The *end vertices* indicate the source and the destination vertex for the edge. To understand the vertex-cut consider that during partitioning of a graph if an edge is assigned to one partition, and another edge having same end vertex is assigned to another partition, then a *vertex-cut* is formed between the partitions. For the vertex-cuts, the vertex copies or replicas are created in different partitions depending upon the distribution of their edges among the partitions. As in Figure 1.1(b) the edges are partitioned in different partitions, and the dotted lines between the partitions are the vertex-cuts. Vertex cut shows the link between two copies of the same vertex (v) maintained in different partitions.

The vertex sends messages for synchronization of its states to the partitions containing its copies. Therefore, synchronizing the state of the vertex with copies present in different partitions introduces a communication cost called as the cut-cost between the partitions. The messages can be aggregated or sent without aggregation. We assume that one copy of the vertex act as the master vertex, which collects the messages from its neighbors in other partitions. In Figure 2.2 the vertex v in the partition 1 act as the master vertex; it collects messages from the vertex b and g present in the partition 2. Aggregation of messages means that the messages from the vertex b and g are combined and then sent to the vertex v, whereas for no

aggregation these messages are sent separately.

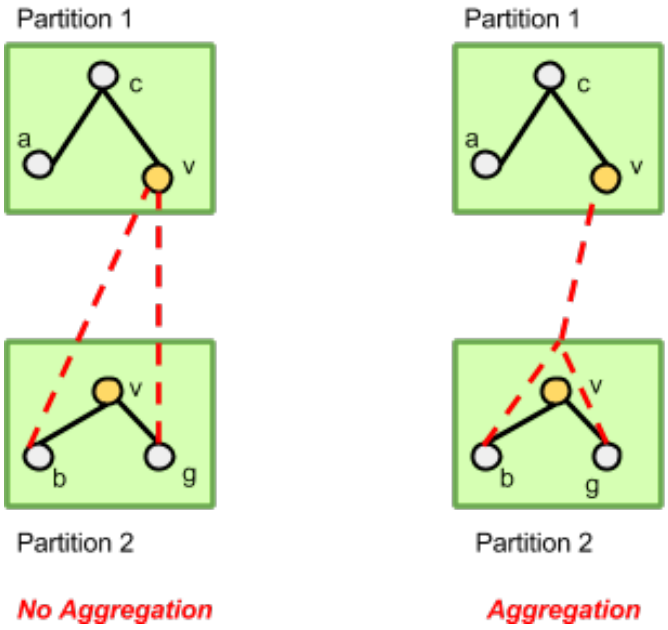


Figure 2.2: No Aggregation and Aggregation of Messages in Edge Partitioning

In case of no aggregation, the communication cost between the copies of a vertex is equal to the number of its neighbors in the partitions other than the one containing the master vertex. In Figure 2.2 this cost is two as the vertex v has two neighbors in the partition 2. However, with aggregation this cost is equal to the number of partitions containing the vertex copies, which in the above case is one as there is only one partition containing the copy of the vertex v other than the one containing the master vertex.

A large number of vertex-cuts increase the communication cost for the vertex having its replicas in different partitions. The aim of good vertex partitioning algorithms is to reduce the vertex-cuts and balance the computation load among the computing nodes.

2.2 Power-Law Graphs

Before explaining the partitioning algorithms in the next section, it is important to understand the power-law [9] of graphs, since it impacts the partitioning problem. According to the graph theory research [9] on natural graphs, most of the real world graphs like the World Wide Web, social network graphs, communication

graphs, biological system graphs and many others have a degree distribution that follows the power-law. Therefore, we evaluated our partitioning algorithms on power-law graphs with an aim to partition natural graphs efficiently by minimizing the communication and the computation cost. Power graphs are difficult to partition due to their highly skewed nature [20,22]. The challenges faced in partitioning power-law graphs are mentioned in detail in [9]. As far as we know, the Power Graph Greedy Vertex-Cut [9] algorithm is one of the most efficient algorithm to partition the power-law graphs, our custom Degree based graph partitioning algorithm is based on this algorithm for partitioning power-law graphs efficiently using the vertex-cut approach.

According to the power-law, for a given vertex V, the probability of this vertex having the degree d is given by

$$d^{-\alpha} \tag{2.1}$$

where,
α = *positive constant.*

The constant α controls the skewness of the degree of the graph. To give an intuitive idea of power-law graphs, think of a social network where celebrities have more followers or friends than other people, but the number of common people exceeds the number of celebrities. This means there are more nodes with a low degree than the ones with a high degree.

2.2.1 Partitioning Power-Law Graphs

In this section we compare the vertex partitioning technique with the edge partitioning technique on power-law graphs. Our thesis implements both these approaches for understanding how these techniques work on natural graphs.

The traditional vertex partitioning approach is not suitable for power-law graphs because tools like [28,29], that create balance edge-cut partitions, perform inefficiently [20,21,22] on power-law graphs. The reason for this is that in vertex partitioning, the edge-cuts create a network and a storage overhead because a copy of the adjacency information (the information of an edge between the partitions along with the source and the destination vertex for that edge) is maintained in both partitions. Some approaches like [23] maintain a ghost vertex, which is a local copy of the vertex, and the edge data for each edge-cut. As shown in Figure 2.3 two ghost vertices and the edge data is maintained in the partitions. In case of a change in the vertex or the edge data, the change must be communicated to all the partitions containing the vertex and the edge data.

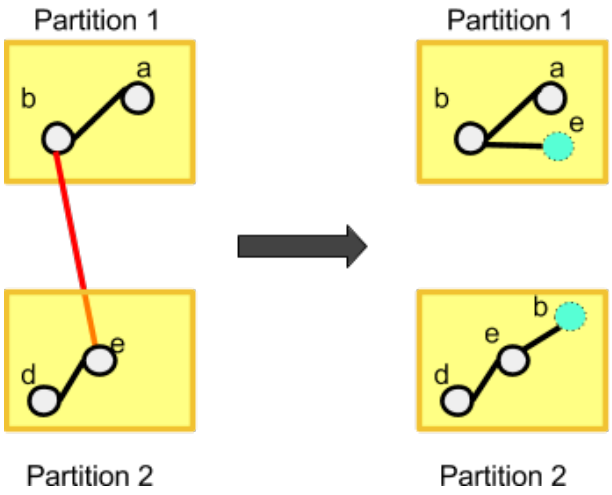


Figure 2.3: Edge Partitioning and Ghost Vertex

The PowerGraph [9] abstraction proposed an edge partitioning approach for natural graphs. In this proposed approach the edges are stored only once in the partitions, so a change in the edge data does not need to be communicated across the partitions. However, the vertex copies are maintained in different partitions; therefore a change in the vertex must be copied to all partitions containing the vertex copies as shown in Figure 2.4.

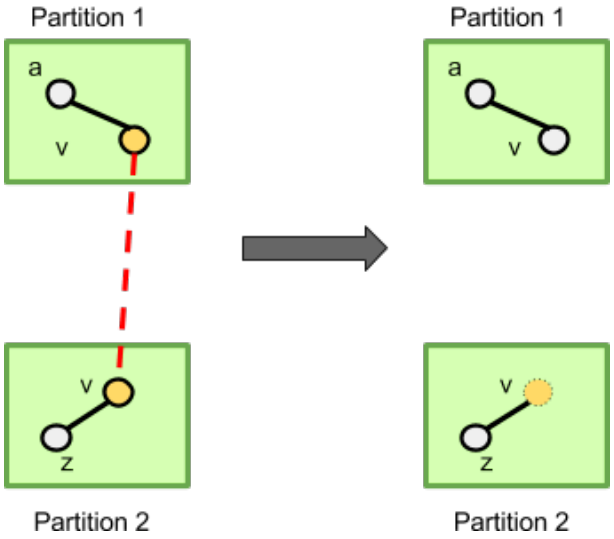


Figure 2.4: Vertex Partitioning and Vertex Copies

According to the vertex-cut approach proposed in the PowerGraph [9] abstraction, it would be better to partition the high-degree vertices, as they are less in number to reduce the replication. However, partitioning the low-degree vertices will increase the replication of vertices due to their large quantity. Our degree based partitioning method uses the degree information of the vertices to partition the high-degree vertices.

2.3 Partitioning Algorithms

Large graphs, like social media graphs, can be processed efficiently in a distributed set-up because it is hard to process them on a single commodity machine due to its limited processing power and storage capacity. Therefore, for distributed processing, these graphs need to be partitioned across several computing nodes. They can be partitioned using vertex partitioning or edge partitioning. Traditional partitioning methods were offline, but our work is based on the implementation of partitioning methods that are online. The motivation behind using online partitioning methods is that offline partitioning methods like *METIS* [24] need to observe the whole graph before partitioning. Thus, creating a high computation cost, whereas the online partitioning methods work on-the-fly reducing the computation cost. These online partitioning methods are based on the *stream partitioning* [6] approach. It partitions the data as it arrives only by knowing the current state of the data instead of knowing about the data that will arrive in the future. This technique makes computations faster. In our case as we partition graph streams, so the input is in the form of a vertex stream or an edge stream. This method of partitioning is called *streaming graph partitioning* [6]. We have implemented partitioning algorithms for the graph streams, the partitioning is done on-the-fly and in one-pass to reduce the computation cost.

To the best of our knowledge, the algorithms we chose to implement are some of the efficient vertex and edge stream partitioning algorithms in terms of reducing the communication and the computation cost across the computing nodes. Vertex partitioning algorithms include: *Linear Greedy* [6] and *Fennel* [7]. Edge partitioning algorithms include: *Least Cost Incremental* [12], *Least Cost Incremental Advanced* [12] and our own variation of *Power Graph Greedy Vertex-Cuts* [9] called as *Degree based partitioner*. We are interested to know how these different partitioning techniques perform on the graph streams by evaluating their partitioning quality metrics like the cut-costs and load balancing.

Assumptions: We consider a streaming graph which is represented by $G = (V, E)$, having the total number vertices n and the total number of edges m. The vertices and the edges of a graph arrive in the form of stream, which is partitioned using the partitioning algorithms. All of these algorithms are one-pass algorithms; they take decision on-the-fly, providing low-latency. Furthermore, once a vertex or an edge is assigned

to a partition, it cannot be reassigned to another, making the assignment irrevocable. Reassigning the vertex or edge increases the communication cost as the data needs to be transferred to other partitions in case of re-assignment.

2.3.1 Algorithms for Vertex Stream

In this section we give the details of the algorithms for partitioning vertex streams. The input is in form of a vertex stream, where each vertex has a unique vertex ID, and its neighboring vertices' IDs.

2.3.1.1 Linear Greedy

This algorithm is regarded as the most efficient one in terms of having less edge-cuts, from the algorithms that were first introduced for streaming graph partitioning [6]. It follows a greedy approach for partitioning; the vertices, as they arrive, are sent to the partition which has most of its neighbors. There is also a penalty factor involved based on the load of the partition for load balancing.

Formula:

For a graph $G = (V, E)$, having the total number vertices n and the number of edges m. It assigns a vertex v to a partition out of k total partitions. P^t represents the set of partitions at time t and $P^t(i)$ is the individual partition referred by the index i. $\bigcup_{t=1}^{k} P^t(i)$ is equal to the vertices assigned to the partitions so far. $w(i,t)$ is the penalty factor for the partition with an index i at time t. The partition with a maximum value of the function $g(P)$ is assigned the vertex, this value is calculated based on the following formula:

$$\begin{aligned} g(P) =& argmax_{i \epsilon [k]} \{ |P^t(i) \cap \Gamma(v)| w(t,i) \} \\ & w(t,i) = 1 - \frac{P^t(i)}{C} \end{aligned} \tag{2.2}$$

where,
v = the incoming vertex
P^t = set of partitions at any time t
$P^t(i)$ = an individual partition
$\Gamma(v)$ = set of neighbors of the vertex v
C = the capacity constraint for each partition, in this case v/k

Each vertex in the input vertex stream contains information about its neighbors. In Linear Greedy, this information is used to check if the neighbors are present in the

partitions or not. The input vertex is assigned to the partition containing most of its neighbors, until the load of that partiton is large enough that the value of $g(P)$ for that partiton becomes lower than the value of $g(P)$ for the other partitions. The vertex is always assigned to the partition with the highest value of $g(P)$.

The formula for Linear Greedy gives a high priority to the number of neighbors of the input vertex in different partitions than the load across different partitions to preserve the locality of the vertices. This approach would result in lower edge-cuts. Furthermore, this algorithm also tries to do load balancing as well by penalizing the partitions based on their load.

2.3.1.2 Fennel

This algorithm improves the idea of Linear Greedy by adding an additional cost factor to the formula. It considers two properties of the input vertex for partitioning: the highest number of its neighbors in the partition and the lowest number of its non-neighbors in the partition.

The cost function consists of the inner and the outer-cost. Cost function is based on the following formula:

$$f(P) = C_{out} + C_{in}$$

where,
C_{in} = Inter-partition cost which depends on the number of edge cuts between the partitions.
C_{out} = Intra-partition cost which depends on the loads in the partitions

Fennel keeps in account both these costs, with an objective to keep the cost a minimum as possible.

Formula:

For an incoming vertex v, the total number of partitions are k. The set of partitions are represented by P and an individual partition is referred by an index i, as P_i. It assigns the vertex v to the partition i with the maximum value of $\delta g(v, P_i)$, such that $\delta g(v, P_i) \geq \delta g(v, P_j)$, $\forall j \epsilon \{1, ..., k\}$.

$\delta g(v, P_i)$ is calculated using the following formula:

$$\begin{aligned} \delta g(v, P_i) =& |N(v) \cap P_i| - \alpha\gamma|P_i|^{\gamma-1} \\ & loadlimit = v\frac{n}{k} \end{aligned} \tag{2.3}$$

where,
$\gamma = 1.5$
$\alpha = \sqrt{k}\frac{m}{n^{3/2}}$
$v = 1.1$
$m =$ the total number of edges
$n =$ the total number of vertices
$N(v) =$ neighbors of the vertex v
$|P_i| =$ the number of vertices in a partition P

The partitions cannot have load more than the *loadlimit*. The parameters α, γ and v are tunable, we chose the values that suited our test-case after experimenting and based on the research [7] done for Fennel.

For partitioning a vertex stream, Fennel uses the neighbors' information of a vertex like Linear Greedy. It considers the partition containing the maximum number of neighbors of the input vertex. In addition, Fennel considers the number of non-neighbors as well, it tries to minimize this number; hence we can say that Fennel interpolates between the neighbors and the non-neighbors to provide better results.

The parameters α, γ and v control the amount of weightage given for maximizing the number of neighbors and minimizing the number of non-neighbors for the input vertex during partitioning. Maximizing the number of neighbors means that the vertex is placed in the partition containing the maximum number of its neighbors, which results in reduced number of edge-cuts. On the other hand, minimizing the number of non-neighbors means that the vertex is placed in the partition having the least number of its non-neighbors, which results in reducing the edge-cuts and balancing the load across the partitions.

2.3.2 Algorithms for Edge Stream

An edge stream consists of edges with values of its end vertices. Each edge has a source vertex ID and a destination vertex ID. These IDs are the unique numbers used for identification of the vertices. The edge can also have an edge value to represent its weight.

2.3.2.1 Least Cost Incremental

This is a simple algorithm. The algorithm assigns a cost value from 0 to 2 to the partitions when an edge is processed. The goal is to keep the cost as low as possible.

Each partition has a cost 0, 1 or 2 based on the following rules:

- 0 : If both end vertices of the edge e are already present in the given partition.
- 1 : If one end vertex of the edge e is already present in the given partition.
- 2 : If none of end vertices of the edge e are present in the given partition.

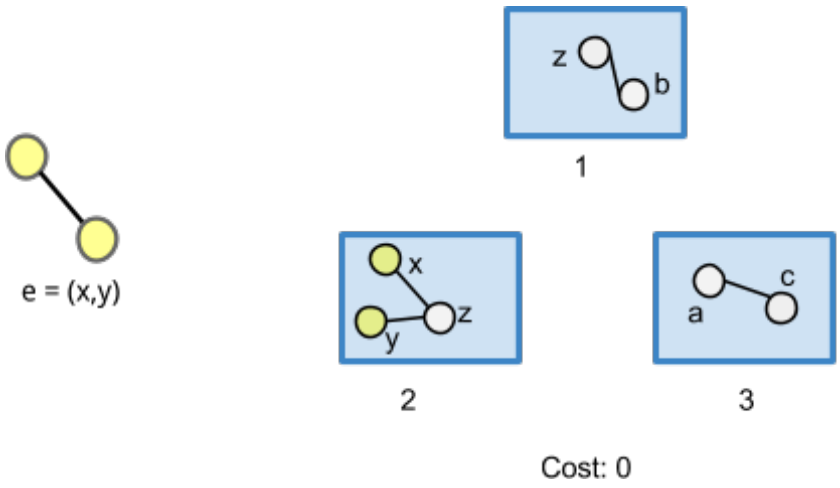


Figure 2.5: Cost 0 case, Adapted from Presentation on Paper Balanced Graph Edge Partition [12],2014. By F. Bourse, M. Lelarge, and M. Vojnovi. Retrieved from [4]

In Figure 2.5, the edge $e = (x, y)$ is the input edge, x and y are its end vertices. Only the partition 2 contains these end vertices, so the cost to place the edge in partition 2 will be zero; thus, the partition 2 in the above case is the best choice.

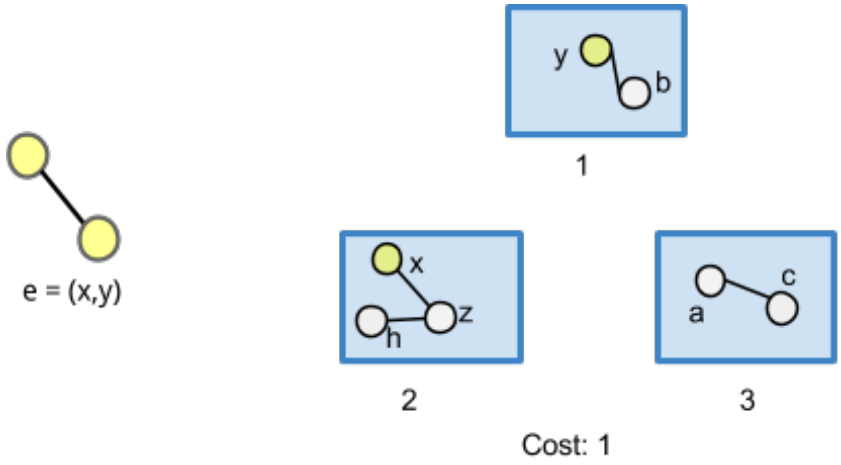


Figure 2.6: Cost 1 case, Adapted from Presentation on Paper Balanced Graph Edge Partition [12],2014. By F. Bourse, M. Lelarge, and M. Vojnovi. Retrieved from [4]

In Figure 2.6, one end vertex x of the incoming edge $e = (x, y)$ belongs to the partition 2 and the other end vertex y to the partition 1. The cost for both partitions in this case is one, so the edge can go either to the partition 1 or the partition 2.

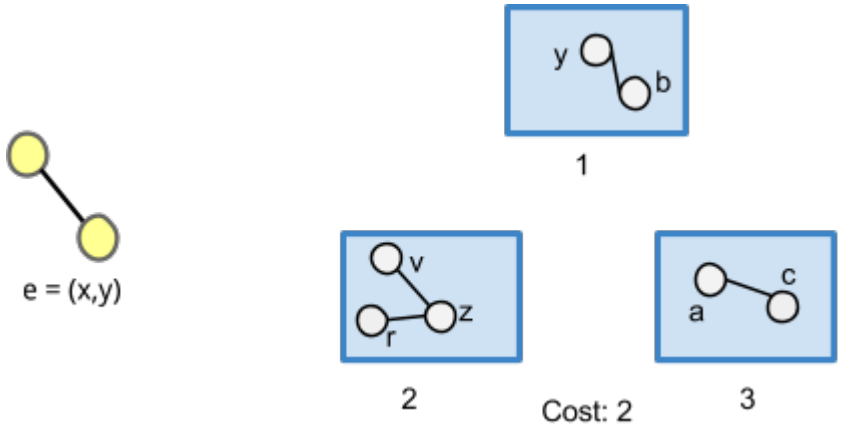


Figure 2.7: Cost 2 case, Adapted from Presentation on Paper Balanced Graph Edge Partition [12],2014. By F. Bourse, M. Lelarge, and M. Vojnovi. Retrieved from [4]

For the case shown in Figure 2.7, there is no partition that contains the end vertices of the input edge $e = (x, y)$. The cost for all the partitions is 2. In this condition the edge will be assigned to any random partition out of the three.

The algorithm is simple to understand. It completely ignores the load balancing criteria by just considering the cost in a greedy manner. It will place the input edge to the partition containing the maximum number of its end vertices.

2.3.2.2 Least Cost Incremental Advanced

This algorithm is an advanced version of the *Least Cost Incremental* algorithm as it also considers the load balancing criteria along with the cost mentioned in the *Least Cost Incremental* algorithm. To implement load balancing, an increasing cost function $c(x)$ is added for putting a penalty based on the load of the partition.

For an edge $e = (x, y)$, it belongs to the partition P_j from the set of partitions P, with the maximum value of I. The total number of partitions is k. For load balancing an increasing convex function $c(x)$ is used, such that $c(0) = 0$.

The value of I can be computed from the following formula:

$$I = argmax_{j\epsilon[k]}\{|V(P_j) \cap (x, y)| - [c(|P_j \cup (x, y)|) - c(|P_j|)]\} \tag{2.4}$$

where,
k= the total number of partitions
P_j = individual partition from a set of partitions P
$V(P_j)$ = set of vertices in the partition P_j

This algorithm uses the end vertex information of the input edge like Least Cost Incremental. It counts the number of end vertices present in each partition and the load on each partition. This information is used for partitioning by finding the partition with the highest value of I; hence using a better approach in terms of load balancing than Least Cost Incremental.

The input edge is most likely to be placed in the partition containing its end vertices. However, the convex function $c(x)$ penalizes the partitions based on their load, which means that if the load increases, the penalty factor also increases. Thus, decreasing the chance of placing the edge in the partition with a high load.

2.3.2.3 Degree Based Partitioner

This algorithm uses the basics of the *Power Graph Greedy Vertex-Cut* [9] heuristic, which is suitable for partitioning highly skewed graphs. We have already discussed the importance of power-law graphs in section 2.2. First we will briefly explain the *Power Graph Greedy Vertex-Cuts* heuristic for a better understanding of the algorithm.

Power Graph Heuristic:

Suppose that the input edge is represented by $e = (x, y)$, where x and y are its end vertices. $S(x)$ is the set having partition numbers that contain the vertex x. Similarly $S(y)$ is the set having partition numbers containing the vertex y. It uses the degree

information of all vertices for keeping in account the number of unassigned edges of the vertices.

For any incoming edge $e = (x, y)$ the steps followed are :

- Step 1: Find $S(x)$ and $S(y)$, if $S(x) \cap S(y)$ is not empty then assign the edge to the partition number from the intersection.
- Step 2: If $S(x) \cap S(y)$ is empty then assign the edge to the partition in $S(x) \cup S(y)$ that contains either x or y with the most number of unassigned edges of the end vertices x and y.
- Step 3: In case only one out of x and y has been assigned previously, then choose the partition with the assigned vertex.
- Step 4: If both $S(x)$ and $S(y)$ are empty then assign the edge to the least loaded partition.

It is clear that the algorithm follows a greedy approach by placing the edges to the partitions that have already seen one or both of the end vertices for the input edge. In worst case it can happen that all the edges end up in the same partition if the input is traversed in a breadth-first search order. There must be a limit on the load of the partitions to avoid this. Another drawback is the degree information, for most of the graphs it cannot be known prior to processing.

Degree Based Partitioner:

In the *Degree based* partitioner we used basic rules of the Power Law Greedy Vertex-Cut heuristic, which is suitable for processing the power law graphs keeping in account the two major drawbacks, which include: the load on the partitions and the degree information of the end vertices.

For an incoming edge $e = (x, y)$, we keep updating the degree of the end vertices as we process the stream; hence, eliminating the need of knowing the degree before hand. We call this as the evolving degree. This approach of using an evolving degree is also used in *HDRF* [15] graph partitioning algorithm.

Sets $S(x)$ and $S(y)$ are maintained containing the partition numbers having the end vertices x and y. The following steps are followed for partitioning:

- Step 1: Find $S(x)$ and $S(y)$, if $S(x) \cap S(y)$ is not empty then assign the edge to the partition number from the intersection set (same as the Power Graph Greedy Vertex-Cuts heuristic).

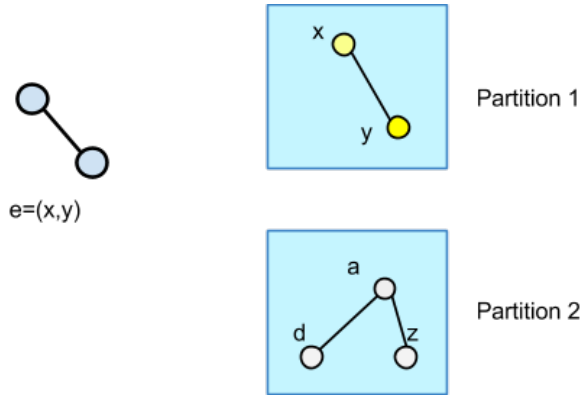


Figure 2.8: Case 1: Degree Based Partition

In Figure 2.8, the end vertices x and y of the edge $e = (x, y)$ belong to the partition 1. Therefore, according to the algorithm the edge $e = (x, y)$ will be placed in the partition 1.

- Step 2: If $S(x) \cap S(y)$ is empty then find $S(x) \cup S(y)$. For every partition referred by an index i in the set $S(x) \cup S(y)$, calculate the value of $I(v, i)$, where v represents the end vertex i.e either x or y.

Formula:

$$\begin{aligned} & I(v,i) = d(v) + Z \\ Subject\,to \quad & |P_i| \leq load\,limit \end{aligned} \tag{2.5}$$

where,
$Z = \alpha\gamma|P_i|^{(1-\gamma)}$
P_i = edges in a partition P
$\gamma = 1.5$
$\alpha = \sqrt{k}\frac{m}{n^{3/2}}$
$v = 1.1$
m = the total number of edges
n = the total number of vertices
$d(v)$ = the degree of vertex v recorded so far
$|P_i|$ = the number of elements in partition P_i, here P is the set of partitions
k = the total number of partitions.

Assign the edge $e = (x, y)$ to the partition i such that $I(v, i) \leq I(v, j)$, where $j\epsilon[S(x) \cup S(y)]$.

This step places the edge to the partition based on the degree of its end vertices. There is more probability that the edge is placed to the partition containing its end vertex with a lower degree if the penalty value Z is not too much depending upon the load in that partition.

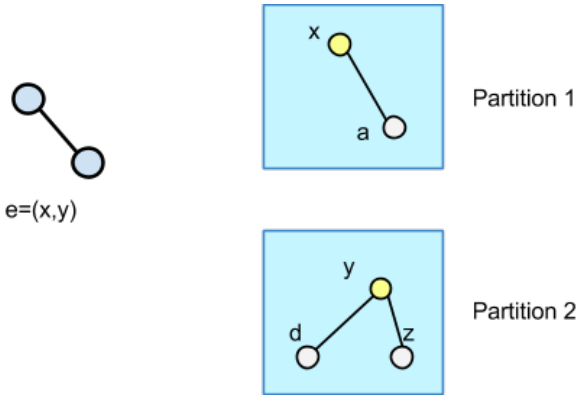


Figure 2.9: Case 2: Degree Based Partition

In Figure 2.9, the incoming edge $e = (x, y)$ has its end vertices assigned in both the partitions 1 and 2. The degree of the vertex x in the partition 1 is one where as the degree of the vertex y is two in the partition 2. There are more chances that the edge will move to the partition 1 as the degree of x is lower than y, but the penalty factor Z is also added to the degree for load balancing purpose. The idea is to divide the high degree vertices as they are less in number compared to the low degree vertices in power-law graphs for reducing the vertex-cuts.

- Step 3: Same as step 3 of the Power Graph Greedy Vertex-Cuts heuristic. Only if the partitions containing one of the end vertices are more than one, then assign the edge to the partition where the degree of the assigned vertex is higher. In addition to this, the load penalty factor Z is subtracted from the degree value for load balancing.

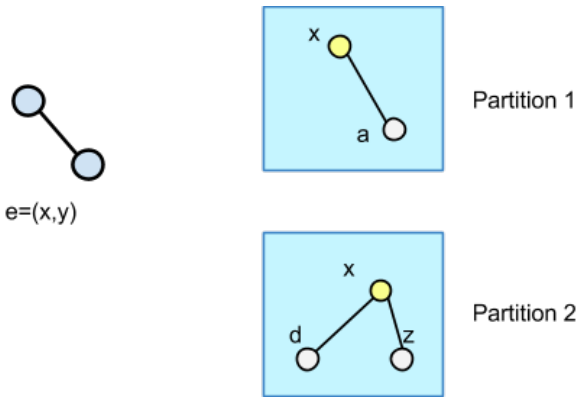


Figure 2.10: Case 3: Degree Based Partition

In Figure 2.10, the incoming edge $e = (x, y)$ has its end vertex x assigned in

both the partitions 1 and 2. The degree of the vertex x in the partition 1 is one where as the degree is two in the partition 2. There are more chances that the edge will move to the partition 2, as the degree of x is greater in the partition 2, but the penalty factor Z is also subtracted form the degree value for load balancing purpose.

- Step 4: If both $S(x)$ and $S(y)$ are empty then assign the edge to the least loaded partition but keeping in mind the *loadlimit*. The load of any partition cannot exceed the *loadlimit*.

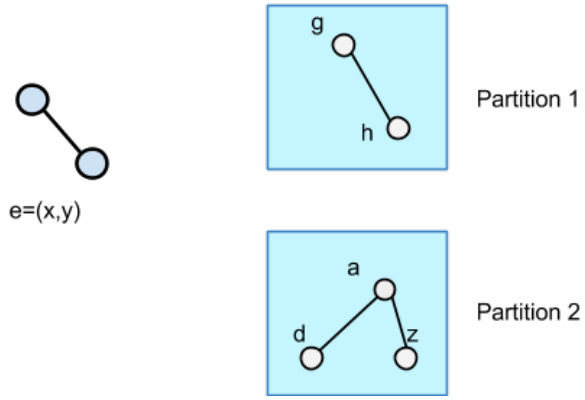


Figure 2.11: Case 4: Degree Based Partition

According to this step, in Figure 2.11, the end vertices x and y of $e = (x, y)$ do not belong to any partitions. Therefore, the $e = (x, y)$ will be placed in the partition 1 as it is less loaded than the partition 2.

For partitioning an edge stream, the Degree based algorithm uses the end vertices' information of the input edge and checks the number of end vertices present in the partitions. If both of the end vertices are present in a partition then the edge is assigned to that partition; if only one end vertex is present in the partition then the edge is assigned to that partition; if one end vertex is present in more than one partitions or both end vertices are present in different partitions, then use the degree based formula present in the equation 2.5 and place the edge in the partition containing the minimum value of $I(v, i)$. In case no end vertex is present in the partitions, then load balancing is performed. This algorithms gives priority to create vertex-cuts for the vertices with a high-degree as they are less in number than the low-degree vertices; therefore, it is good for partitioning highly skewed graphs.

2.4 Feature Comparison

A comparison table comparing different algorithms discussed in sections 2.3.1 and 2.3.2 is given below.

Algorithm	Data model	Programming model	Partitioning factors	Requirements	Cuts
Linear Greedy	Vertex Stream	Dynamic model: works on the fly	neighbors	Requires total number of vertices	Edge-cuts
Fennel	Vertex Stream	Dynamic model: works on the fly	neighbors and non-neighbors	Requires total number of edges and vertices	Edge-cuts
Least Incremental Cost	Edge Stream	Dynamic model: works on the fly	Cost based on vertices present in the partition	Does not require any information	Vertex-cuts
Least Cost Incremental Advanced	Edge Stream	Dynamic model: works on the fly	Cost based on vertices present in the partition and load	Requires total number of edges	Vertex-cuts
Degree Based	Edge Stream	Dynamic model: works on the fly	Based on Degree	Requires total number of edges and vertices	Vertex-cuts

Table 2.1: Comparison Table for Partitioning Algorithms

For algorithms working on a vertex stream *Fennel* seems to be more efficient than the *Linear Greedy* algorithm, because it does not only considers the neighbors but also the non-neighbors. *Linear Greedy* being simple to implement, is a good competitor of *Fennel* for comparison.

Algorithms for partitioning an edge stream are as simple as *Least Cost Incremental* , which does not require any prior information about the graph. However, the complex ones like *Least Cost Incremental Advanced* and *Degree based* require the total number of vertices and edges to be known before partitioning.

Chapter 3
3 Background

3.1 Data Stream Processing

Stream processing is a programming paradigm that is useful for performing low-latency, incremental computations on data. The input data for this paradigm is in the form of streams, like different values of network traffic data, bank transactions or hourly weather report data etc., that are continuously being generated from different data sources. These data streams are not completely stored in the memory because stream processing works with a limited amount of memory following the memory constraints. In addition, it allows to get intermediate results before processing the complete data.

Processing the data while it is incoming is important for systems that rely on timely updates, like the weather systems. The algorithms designed for stream processing work with a limited memory and provide a low-latency.

3.1.1 Data Stream Processing Models

Streaming data is processed record-by-record. The processing can be either single-pass, where each element is processed once, or multi-pass, where each element is processed more than once.

Main stream processing models [13] include: *The Cash register model*, *The Turnstile model* and *The Sliding window model*.

The *Cash register model* works by maintaining a vector or a one dimensional function $A = |0...N-1|$ of values, while it processes the elements $a_1, a_2, a_3...$ in the data stream. During each step a value in the vector is updated. The update can be either positive or negative. The model with a negative update is called a *Turnstile model*.

In the *Sliding window model*, first the elements of a data stream are placed in a window, then the window is evicted and the elements are processed. After eviction,

the window moves to next elements. The window can further have two types: *Tumbling* or *Sliding*. All the elements of the window are evicted by a *Tumbling window*, with no elements overlapping. Whereas, if the window does not evict all of the elements, and the elements overlap when the window slides over them, it is called a *Sliding window*.

Furthermore, the windows can have a fixed size (n) for holding n elements, and they can also be based on time intervals, for example: data elements with a timestamp in the range of 1-5 sec will go to one window, and from 5-10 sec will go to another. The timestamp value is based on the time at which the data element was generated from the source. This helps in accurate approximation of the results for an out of order stream.

3.1.2 Data Stream Approximation Strategies

Data streams can be processed using techniques like *sampling* and *sketching*. In *sampling*, the samples of the input data stream are created using probability for determining whether to keep the data element in a particular sample or not. On the other hand, *sketching* involves creating synopsis of the data elements processed. The synopses are approximate data structures stored in the memory during processing, targeting specific computations or measures. These synopsis are updated each time a new element in the stream is processed. For example: while calculating the average value of a stream, the sum value is updated every time an element in the stream is processed. This technique gives us an approximate value for the stream processed so far.

3.2 Graph Stream Processing

In this section, we present an overview of graph streaming. Almost every data can be represented in the form of entities and relationships between them; graphs are the best choice to represent such entities and relations in the form of vertices and edges. Large-scale graph processing becomes challenging due to the fact that a single machine often has an insufficient capacity for such computation in terms of the memory and the computation power. The data streaming model is well suitable for such dynamic, unbounded graphs which can be processed with a limited amount of memory, and in parallel.

3.2.1 Graph Stream Models

Graph stream can arrive in any order. The two most common models based on orderings are known as: the *Adjacency model* and the *Incidence model*. In the *adjacency model*, the edge stream arrives in a random order and there is no limit on the degree of a vertex. On the contrary, the *incidence model* is based on an edge stream where all the edges belonging to the same vertex arrive together, and there is a limit on the degree of the vertex.

The streaming models discussed so far have a memory constraint linear limit. Hence, it is not a practical approach for performing graph processing algorithms that require to store the vertices of a graph exceeding the memory limits of the system. A relatively new model called the *semi-streaming model* [14] solves this problem.

Real-life graphs have $n(the\ number\ of\ edges) >> m(the\ number\ of\ vertices)$. The *Semi-streaming model* is useful due to that fact that it stores the number of vertices, because storing the number of edges will cost a large amount of memory. This model allows to use $O(n * polylog(n))$ of memory, where n is the number of vertices, and allows a constant or logarithmic number of passes considering the number of vertices(n).

3.2.2 Graph Stream Representations

In this section, we discuss different ways in which a graph stream is formed. Each approach has its own advantages and disadvantages.

A graph represented by $G = (V, E)$, consists of vertices $V = (V_1, V_2...V_n)$ and edges $E = (E_1, E_2...E_n)$. There are different ways in which this graphical information can be streamed. The simplest one, and the commonly used one is referred as the *Edge-only stream*, formed with an edge stream that contains the end vertex values. Another approach is the *Combined Stream of Vertices and Edges*; it has separate streams for the edges and the vertices. Finally, the last approach is called the *Stream of Triplets*, formed with triplets that include: the source vertex, the target vertex and the edge value. It is good in terms of having the vertex values, but we might face empty values in the triplet if some of the vertex values are not known while processing.

3.3 Apache Flink

This section is aimed to give an idea of Apache Flink, explaining how Flink does data processing, which involves batch and stream processing. Moreover, the graph stream processing API [10] developed on top of Flink is also presented briefly because our partitioning algorithms are developed using it.

3.3.1 Flink as Data Processing Engine

Flink is an Apache project for processing big data in a distributed environment; it aims at a low-latency stateful processing with consistency guarantees. Flink's distinctive feature is low-latency stream processing and memory management. Most of the time the program runs inside the memory, but when the memory is not enough the intermediate data and state can be transparently spilled to the disk.

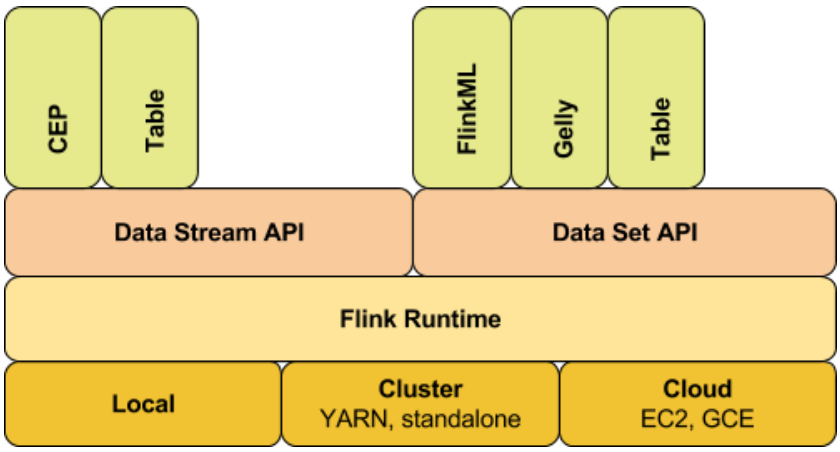


Figure 3.1: Apache Flink Stack

The Flink stack is shown in Figure 3.1. Programs can run locally on Flink as well as in cluster mode. For running Flink programs on a cluster, it is possible to use the YARN[14] cluster or the standalone Flink cluster.

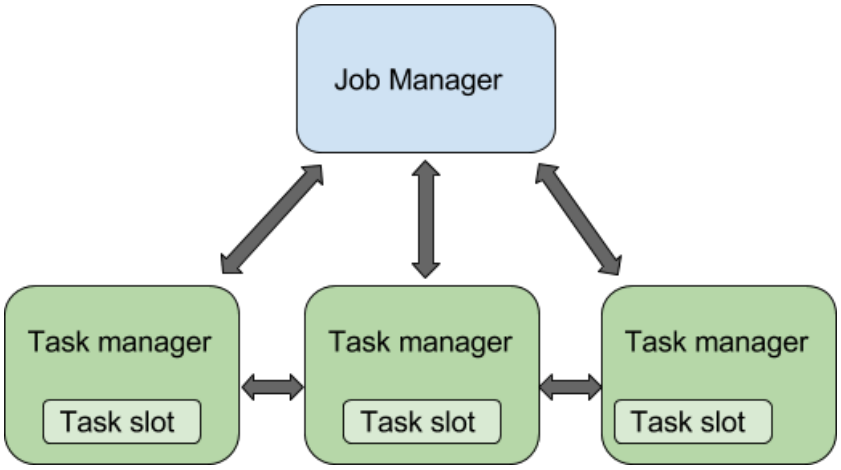


Figure 3.2: Task Management in Flink

Flink tasks are executed by the Job manager and Task managers. The scheduling in Flink is done by the Job manager, it keeps a check on the tasks assigned to the task managers, and monitors the resources. On the other hand, the task managers execute the tasks independently and exchange data with each other. They have several slots for handling the tasks in order to achieve parallelism.

3.3.2 Flink Streaming API

The Flink Streaming API processes streams by using a pipelined approach i.e it pipelines the data as the data keeps on arriving from the source. Flink supports different data sources like file systems, message queues (Twitter API, Kafka etc.), TCP sockets and arbitrary sources defined by the user. The API provides support for stream connectors that act as an interface for accessing data from third party sources. The connectors currently supported include the Twitter streaming API [30], Apache Kafka [31], Apache Flume [32] and RabbitMQ [33]. Furthermore, different data transformations can be applied to the data stream like map, reduce, filter and aggregations to create new transformed streams. Windowing semantics are also supported by Flink for streaming.

3.3.3 Flink Graph Processing API

Gelly[16] is Flink's Graph processing API; it contains different methods to simplify graph processing. Different transformations and methods provided in the Flink batch processing API can be used in Gelly, since Gelly is developed on top of Flink batch processing API. Furthermore, Gelly includes certain graph algorithms that can be used as library functions.

Graph Representation: Graph in Gelly can be represented using the `DataSet` type. `DataSet` of edges and `DataSet` of vertices can be used in this regard. In `DataSet` of vertices, each vertex has a unique ID, similarly in `DataSet` of edges, each end vertex of the given edge has a unique ID representing the source and the destination of the edge.

Transformations and Common Utilities: There are several methods available to get basic graph parameters like the number of vertices, the number of edges and the degree of a vertex. The basic transformations like map, filter and join etc. can be applied on graph objects, which makes possible to perform several operations on these graph datasets.

Neighborhood Operations: Neighborhood methods allows to perform operations on the first-hop neighbor of any vertex. Functions like `reduceOnEdges` allows to access the end vertex ID and the edge value of the neighbor edges.

Iterations: Iterations are useful for implementing graph processing related algorithms and machine learning techniques. Gelly aims to support multiple iterative methods [18]. Currently, it supports programs written using the gather-sum-apply [17], scatter-gather [18] and vertex-centric [1] model. These methods cannot be used for the stream processing API, since they are all based on iterations.

3.3.4 The Graph Streaming API for Flink

The Graph stream processing framework[10] works on top of the Flink's streaming API and it provides certain functions similar to Gelly. It works well in a distributed environment setup for graph processing.

Certain basic transformations like map and filter on edges and vertices are provided by the framework, including functions to calculate parameters like the graph vertices, edges and degree. All the algorithms implemented are one-pass and they work for the edge stream. The algorithms are executed in parallel, where the input edge stream is divided into different partitions. The results from different windows are reduced to get the final result. Special *aggregation* functions are used to perform this merging of results.

3.3.4.1 Implemented Algorithms

This section gives a brief description of some of the graph processing algorithms implemented by the Graph stream processing framework. Following are few of the main algorithms implemented:

Bipartition:

The bipartiteness algorithm is used to check if the graph is bipartite or not. If the vertices of the graph can be divided into two groups such that there are no edges within those groups of vertices, then the graph is bipartite. In the API, a merge tree is used to implement this algorithm. First, the `fold` method is applied to the data stream that assigns a true or a false value depending on whether the subgraph is bipartite or not, then `reduce` method is applied to combine results from different windows to get the final result for the graph.

Connected Components:

The connected component algorithm finds the number of connected components in the graph. If there is an edge between two vertices then they are considered to be part of the same connected component. First, the `fold` method is applied to the data stream that assigns a component ID to the vertices if they are connected by edges. Later, the `reduce` method is applied to combine results from different windows to get the final result for the graph.

Triangle Count:

This algorithm is used for counting the number of triangles in the graph stream. If two neighbors of a vertex are also neighbors then they form a triangle. The API uses a broadcast method over the edge stream to make sure all sub-tasks receive the input edge. Later, a sampling mapper is applied to direct the edges to a class called `TriangleEstimate`, which keeps in account the edges processed. In the end a mapper combines all the values from different mappers to get the triangle estimate.

Chapter 4
4 Implementation

The partitioning algorithms discussed in section 2.3 have been implemented for the Graph stream processing framework [10]; it is built on top of the Flink streaming engine. The framework contains different graph stream processing algorithms. The algorithms accept an edge stream as input and are one-pass algorithms as each edge is processed only once.

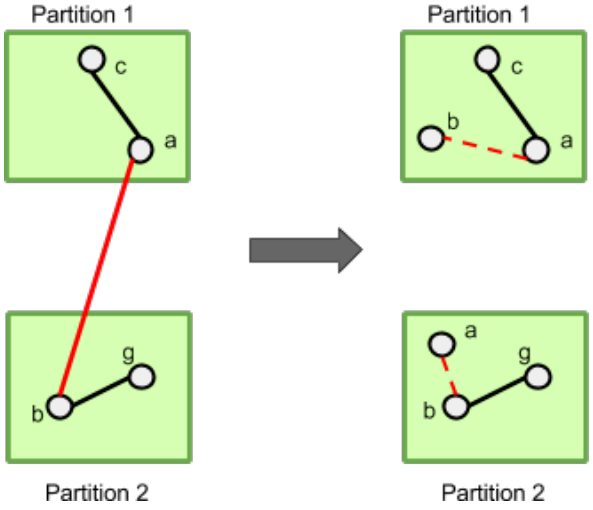


Figure 4.1: Conversion of a Vertex Stream to an Edge Stream

We measured the effect of different partitioning algorithms on these graph stream processing algorithms. The first two partitioning algorithms namely: Linear Greedy and Fennel are for vertex stream partitioning. Therefore, we converted the vertex stream into an edge stream after partitioning for running the graph stream processing algorithms on them. To convert the vertex stream into an edge stream after partitioning, we replicated the edges in the same partitions where its end vertices are placed. For example: as in Figure 4.1 the vertex a belongs to the partition 1 and

the vertex b belongs to the partition 2, the edge $e = (a, b)$ creates an edge-cut. This edge is replicated in the partition 1 and 2 as the end vertices of this edge belong to these partitions.

The edge stream partitioning algorithms we implemented include Least Cost Incremental, Least Cost Incremental Advanced and Degree Based. The graph stream processing algorithms for an edge stream can easily work on the partitioned edge stream after executing these edge stream partitioning algorithms.

4.1 Stream Order

Stream ordering is important as it affects the partitioning quality in terms of the communication and computation costs across the computing nodes. For example, some partitioning algorithms like Least Cost Incremental end up placing all the edges in one partition if the stream follows a breadth first search traversal; thus, increasing the computation cost on one partition. In breadth first search traversal, a vertex of the graph is selected at random, then the neighbors of that vertex are processed first. After that the next level neighbors (the neighbors of the neighbors) are processed. This ordering will keep moving the neighbors in one partition, as they arrive in an order, if the partitioning is done using the Least Cost Incremental algorithm

We consider using the *random* order for the stream as it is the standard order for theoretically analyzing the streaming algorithms [6]. This order assumes that the vertices or the edges arrive at random from the streaming source. Random ordering can help preventing bad orderings that can worsen the partitioning quality of an algorithm like the one mentioned in the previous paragraph for Least Cost Incremental, where the breadth first search ordering is a bad ordering as all the vertices might end up in the same partition following this ordering. However, the random ordering can ignore the locality of edges or vertices in the stream, which is preserved in the depth first search and the breadth first search ordering as the neighbors of the vertices arrive in an order.

4.2 Vertex Stream

The Flink's Stream Processing API provides different functionalities for processing data streams. It provides support for data stream sources like files, message queues (Twitter API, Kafka etc.) and TCP sockets along with user's custom defined data stream sources.

Data streams are commonly created using `Tuples`. Flink supports different tuple data types, and contains its own custom tuple implementation.

For creating a vertex stream we used the `DataStream` type with a `Tuple2` of two elements as shown in Listing 4.1.

Listing 4.1: Vertex Stream

```
DataStream<Tuple2<Long, List<Long>>> vertices = getGraphStream(env);
```

The first element in the `DataStream` of `Tuple2` is the ID of a vertex which is to be placed in a certain partition, the second element is the List of type Long representing the neighboring vertices' IDs. The neighborhood IDs are useful for partitioning because they help in placing the vertex to the partition where its neighbors belong, if they have already been processed by the partitioner.

This vertex stream can be partitioned by using different partitioning algorithms. We implemented two of them, Linear Greedy and Fennel.

4.3 Edge Stream

Gelly provides support for representing an edge in a graph using the `Edge` type. An `Edge` type contains three fields: the source, the target and the value. Methods like `getSource, getTarget` and `getValue` are used for getting the source, the target and the value of the edge. To represent an `Edge` stream, the `DataStream` is used with an `Edge` type as shown in Listing 4.2.

Listing 4.2: Edge Stream

```
DataStream<Edge<Long, NullValue>> edges = getGraphStream(env);
```

An edge is a link between two vertices, these vertices are the source vertex and the destination vertex of the edge. The first element in the `DataStream` of `Edge` is the source vertex ID, and the second is the destination vertex ID. The third element is the edge value. In our case we assigned this value as $NULL$. The source and the destination vertex information of the edge is helpful for placing the edge in the partition containing its end vertices.

This edge stream can be partitioned by using different partitioning algorithms, which in our case are: Least Cost Incremental, Least Cost Incremental Advance and Degree Based partitioner.

4.4 Partitioners

Flink has its own default partitioners; by default, it uses a hash based partitioning method, which results in dividing the vertices or the edges in the stream randomly across the computing nodes without considering the graph structure. Hence, resulting in a large number of edge-cuts or vertex-cuts.

The Streaming API provides support for the creation of custom partitioning methods. To create custom partitioners, we implemented the `Partitioner` interface. The code for the `Partitioner` interface is shown below in Listing 4.3. This interface contains a `partition` method that takes the key and the number of partitions as input parameters, and returns the partition ID.

Listing 4.3: Partitioner interface

```
// Partitioner.class
import java.io.Serializable;
import org.apache.flink.annotation.Public;

@Public
public interface Partitioner<K> extends Serializable {
    int partition(K var1, int var2);
}
```

The key required by the `partition` method is the value based on which the partitioning logic is created. This key is extracted from the elements in the input stream using the `KeySelector` class. For generating keys from the input stream, we created our custom key selector class called the `CustomKeySelector` that implements the `KeySelector` class. Two custom key selectors are used, one generating keys from a vertex stream and the other from an edge stream.

The code for the custom key selector of the vertex stream is shown below in Listing 4.4. The vertex stream consists of the vertex ID and a List of vertex IDs of its neighbors. This key selector uses a HashMap for mapping the vertex IDs as keys to the List of neighbors' IDs as values. The `getkey` method is used to return the key1 which is the vertex ID and `getvalue` method is used to return the list of neighboring vertex IDs. Both the key1 and the list of neighbors' IDs information are required by the vertex stream partitioning algorithms we have implemented.

Listing 4.4: Customised Key Selector for Vertex Stream

```
private static class CustomKeySelector<K, EV> implements
    KeySelector<Tuple2<K, List<EV>>, K> {
      private final K key1;
      private EV key2;
      private static final HashMap<Object, Object> keyMap = new HashMap<>();
```

```
    public CustomKeySelector(K k) {
       this.key1 = k;
    }

    public K getKey(Tuple2<K, List<EV>> vertices) throws Exception {
       keyMap.put((vertices.getField((Integer) key1)),
           vertices.getField((Integer) key1 + 1));
       return vertices.getField((Integer) key1);
    }

    public EV getValue(Object k) throws Exception {
       key2 = (EV) keyMap.get(k);
       keyMap.clear();
       return key2;
    }
  }
```

The code shown in Listing 4.5 is for the key selector that extracts keys from an edge stream. This `KeySelector` is used for generating two keys: the key1, which is the source vertex ID of the edge, and the key2, which is the destination vertex ID of the edge. This key selector uses a HashMap for mapping the source vertex IDs as keys to the destination vertex IDs as values. `getkey` method is used to return the key1, which is the source vertex ID and `getvalue` method is used to return the key2, which is the destination vertex ID. We require both of these values for the edge stream partitioning algorithms.

Listing 4.5: Customised Key Selector for Edge Stream

```
private static class CustomKeySelector<K, EV> implements
    KeySelector<Edge<K, EV>, K> {
     private final int key1;
     private EV key2;
     private static final HashMap<Object, Object> keyMap = new HashMap<>();

     public CustomKeySelector(int k) {
        this.key1 = k;
     }

     public K getKey(Edge<K, EV> edge) throws Exception {
        keyMap.put(edge.getField(key1),edge.getField(key1+1));
        return edge.getField(key1);
     }

     public EV getValue (Object k) throws Exception {
        key2= (EV) keyMap.get(k);
        keyMap.clear();
        return key2;
```

```
    }
}
```

The `CustomKeySelector` is used in our custom partitioner class that implements the `Partitioner` interface. The `partition` method of this interface, as in Listing 4.3, contains the partitioning logic of every algorithm based on the formulas discussed in section 2.3.

4.4.1 Vertex Stream Partitioning Algorithms

The vertex stream partitioning algorithms we implemented are Linear Greedy and Fennel. The code given below in Listing 4.6. is for the vertex stream partitioning algorithm Linear Greedy. The following paragraph explains the methods and the instance variables of the class that implements this algorithm.

- `HashMap <Long, List<Long> > vertices`: This HashMap contains the partition IDs as keys, and the vertex IDs of the vertices placed in them as the values of the HashMap.
- `List<Long> load`: This list contains the load of every partition. The load of each partition is updated in this list every time a vertex is assigned to a partition.
- `CustomKeySelector<T,?> keySelector`: `CustomKeySelector` is for getting the vertex ID and its neighbors' IDs from the vertex stream.
- `Long k`: The number of partitions.
- `Double c`: The capacity constraint for the algorithm.
- `partition`: This method uses the vertex ID and its neighbors' IDs to return the partition ID, after calculating it based on the Greedy algorithm.
- `getValue`: This method gives the number of neighbors of a vertex present in the given partition.

Listing 4.6: Linear Greedy Partitioner Class Implementation

```
//LinearGreedyCustom.java
private static class Greedy<K, EV, T> implements Partitioner<T> {
   private static final long serialVersionUID = 1L;
   private final HashMap<Long, List<Long>> vertices = new
       HashMap<>();//partitionid, list of vertices placed
   private final List<Long> load = new ArrayList<>(); //for load of each
       partiton
   CustomKeySelector<T, ?> keySelector;
   private Long k; //no. of partitions
```

```
    private Double c;   // no. of vertices/total no. of partitions

    public Greedy(CustomKeySelector<T, ?> keySelector, int m) {...}

    @Override
    public int partition(Object key, int numPartitions) {...}

    public int getValue(int p, List<Long> n) {...}

}
```

To partition the vertex stream, first the `partition` method is called. This method gets the vertex ID as input, and uses this ID to get the neighbors' IDs from the stream using the `CustomKeySelector`. After this, `getValue` method is called for the list of neighbors' IDs. This method returns the number of neighbors present in the partitions to the `partition` method. Later, this number is used in the formula of Linear Greedy in the `partition` method to compute the partition ID. The `partition` method then returns this partition ID for placing the input vertex in it. The pseudocodes for the `partition` method and the `getValue` method of Linear Greedy are shown in procedures 1 and 2.

Procedure 1 *partition(key, k)* for Linear Greedy

Input: key: Input vertex ID, k: total no. of partitions
Output: partition ID

1: **procedure** *partition(key, k)*
2: **list** $\Gamma(v)$ = keySelector.getValue(key) ▷ // getValue(key) will return the list of neighbors of the input vertex
3: **for** *all partitons* $i = 1$ *to* k **do**
4: $P^t(i) \cap \Gamma(v) = getValue(i, \Gamma(v))$; ▷ // getValue(i,neighbors) returns the number of neighbors of input vertex present in each partition (1,...,k).
5: $g(i) = |P^t(i) \cap \Gamma(v)| w(t,i)$
6: $w(t,i) = 1 - \frac{P^t(i)}{C}$
end for
7: **for** all partitions j = 1 to k **do**
8: find highest value of $g(j)$
end for
9: Return j ▷ // j is the ID of partition with highest value of $g(j)$

Similarly, for the other vertex partitioning algorithm Fennel, the implementation of the `getValue` method is same except the `partition` method. It has a different logic according to the algorithm and it contains more instance variables used in the formula. The pseudocode for the `partition` method of Fennel is shown in procedure 3.

Procedure 2 *getValue(p, neighbors)* for Linear Greedy

Input: p: partition number, neighbors: list of neighbors of input vertex
Output: number of neighbors of input vertex present in the partition

```
procedure getValue(p, neighbors)
    list vertices = get(p)                                          ▷
// get(p) will return the list of vertices placed in partition p
    for all neighbors i = 1 to size of neighbors list do
        if vertices contain neighbors[i] then
            count++
        end if
    end for
    Return count ▷ // count is the number of neighbors of input vertex present in
the partition p
```

Procedure 3 *partition(key, k)* for Fennel

Input: key: Input vertex ID, k: total no. of partitions
Output: partition ID

```
1:  procedure partition(key, k)
2:      list N(v) = keySelector.getValue(key)                       ▷
    // getvalue(key) will return the list of neighbors of the input vertex
3:      for all partitons i = 1 to k do
4:          N(v) ∩ P_i = getValue(i, N(v));   ▷ // getValue(i,neighbors) returns the
    number of neighbors of input vertex present in each partition (1,...,k).
5:          δg(i) = |N(v) ∩ P_i| − αγ|P_i|^(γ−1)
        end for
6:      var max = δg(0)
7:      for all partitions j = 1 to k do
8:          find highest value of δg(j)
9:          if load on partiton j <= load limit and δg(j) >= max then
10:             max = δg(j)
            end if
        end for
11:     Return j for δg(j)   ▷ // j is the ID of partition with highest value of δg(j)
```

4.4.2 Edge Stream Partitioning Algorithms:

We implemented three edge stream partitioning algorithms: Least Cost Incremental, Least Cost incremental advanced and Degree based algorithm. The code for Least Cost Incremental is given below in Listing 4.7. The methods and the instance variables of the class that implements this algorithm are as follows:

- `CustomKeySelector<T,?> keySelector: CustomKeySelector` is for getting the source vertex ID and the target vertex ID of the edge.
- `HashMap <Long, List<Long> > vertices:` This HashMap contains the partition IDs as keys and the vertex IDs of the vertices placed in them as the values of the HashMap.
- `List<Long> load:` This list contains the load of every partition. The load of each partition is updated in this list every time an edge is assigned to a partition.
- `List<Long> cost:` This list contains the cost value for each partition while processing the edges in the stream. These costs are assigned according to the Least cost incremental algorithm discussed in section 2.3.2.1. Based on these cost values the partitioning is done.
- `Long k:` The number of partitions.
- `partition:` This method uses the source vertex ID and the destination vertex ID to calculate the partition ID based on the partitioning algorithm.
- `compareCost:` This method is used to compare the costs that are assigned to different partitions and return the partition ID with lowest cost to the `partition` method.
- `getValue:` This method computes the values of `List<Long> cost`.

Listing 4.7: Least Cost Incremental Partitioner Class Implementation

```
//LeastCost.java
private static class LeastCostPartitioner<K, EV, T> implements
    Partitioner<T> {
   private static final long serialVersionUID = 1L;
   CustomKeySelector<T, ?> keySelector;
   private final HashMap<Long,List<Long>> vertices = new HashMap<>();
       //for <partition.no, vertexId>
   private final List<Long> load = new ArrayList<>(); //for load of each
       partiton
   private final List<Long> cost = new ArrayList<>();
   private Long k; //no. of partitions

   public LeastCostPartitioner(CustomKeySelector keySelector) {...}
```

```
    @Override
    public int partition(Object key, int numPartitions) {...}

    public int compareCost() {...}

    public int getValue(Long source,Long target, int p) {...}

}
```

For partitioning the edge stream, first the `partition` method is called. This method gets the source vertex ID and the destination vertex ID from the stream using the `CustomKeySelector`. After this, the `getValue` method is used to get the cost for each partition. This method returns the cost list to the `partition` method. This cost list is then compared using the `compareCost` method. In the `compareCost` method the lowest cost is computed, and the partition ID for the partition having the lowest cost is sent to the `partition` method. The `partition` method then returns this partition ID for placing the input edge in it. The pseudocodes for the `partition` method and the `getValue` method of Least Cost Incremental are shown in procedures 4 and 5.

Procedure 4 *partition(key, k)* for Least Cost Incremental

Input: key: end vertex ID of the input edge, k: total no. of partitions
Output: partition ID

1: **procedure** *partition(key, k)*
2: **var** key2 = keySelector.getValue(key) ▷ // getvalue(key) will return the other end vertex of the input edge
3: x = key, y = key2
4: **for** *all partitons i = 1 to k* **do**
5: cost[i] = getValue(x, y, i); ▷ // getValue(x, y, i) returns the cost for each partition based on the number of end vertices present in the partitions (1,...,k). **end for**
6: p = compareCost() ▷ // compareCost() returns the ID of partition with smallest value of $cost[i]$
7: Return p ▷ // p is the ID of partition with smallest value of $cost[j]$

Procedure 5 *getValue(x, y, partition)* for Least Cost Incremental

Input: x and y: end vertices of the input edge, partition: partition ID
Output: cost for the partition

```
1: procedure getValue(x, y, partition)
2:     var cost = 0
3:     if partition contains x and y then
4:         cost = 0
       end if
5:     else if partition contains x and not y then
6:         cost = 1
7:     end else if
8:     else if partition contains y and not x then
9:         cost = 1
10:    end else if
11:    else if partition does not contain x and y then
12:        cost = 2
13:    end else if
14:    Return cost ▷ // cost is the cost value for the given partition and end vertices
```

Least cost incremental advanced algorithm has the same implementation, except the `partition` method has a different algorithm logic, and it contains more instance variables than the ones used in the formula of the Least cost incremental algorithm. The code for the `getValue` method has same logic, except it returns different values when the end vertices are present in the partitions. The pseudocodes for the `partition` method and the `getValue` method are given in procedures 6 and 7.

Procedure 6 *partition(key, k)* for Least Cost Incremental Advanced

Input: key: end vertex ID of the input edge, k: total no. of partitions
Output: partition ID

1: **procedure** *partition(key, k)*
2: **var** key2 = keySelector.getValue(key) ▷ // to get other end vertex
3: x = key, y = key2
4: **for** *all partitons j = 1 to k* **do**
5: $V(P_j) \cap (x, y) = getValue(x, y, j)$; ▷ // getValue(x, y, j) returns the number of end vertices present in partition j.
6: $I[j] = |V(P_j) \cap (x, y)| - [c(|P_j \cup (x, y)|) - c(|P_j|)]$
end for
7: p = compareCost() ▷ // compareCost() returns the ID of partition with highest value of $I[j]$
8: Return p ▷ // p is the ID of partition with highest value of $I[j]$

Procedure 7 *getValue(x, y, partition)* for Least Cost Incremental Advanced

Input: x and y: end vertices of the input edge, partition: partition ID
Output: number of end vertices present in the partition

```
1:  procedure getValue(x, y, partition)
2:      var num = 0
3:      if partition contains x and y then
4:          num = 2
        end if
5:      else if partition contains x and not y then
6:          num = 1
7:      end else if
8:      else if partition contains y and not x then
9:          num = 1
10:     end else if
11:     else if partition does not contain x and y then
12:         num = 0
13:     end else if
14:     Return num ▷ // num is the number of end vertices present in the partition
```

The code for the Degree based partitioner for an edge stream is given in the Listing 4.8. The following paragraph explains the methods and the instance variables of the class that implement this algorithm.

- `CustomKeySelector<T,?> keySelector: CustomKeySelector` is for getting the source vertex ID and the target vertex ID of the edge.
- `Table <Long, Long, Long> degree:` This hash based table contains the partition ID, the vertex ID and the degree of the vertex. It contains the evolving degree of vertices as they are placed in the partitions. This implementation helps in getting the degree of a vertex placed in a partition. The degree information is used in the `partition` method.
- `List<Long> load:` This list contains the load of every partition. The load of each partition is updated in this List every time an edge is assigned to a partition.
- `List<Long> partitionsCount:` This list contains numbers indicating the presence of end vertices of the edge in each partition. If one of the end vertex is present in the partition then this number is 1; if both end vertices are present then 2, otherwise 0.
- `Long k:` The number of partitions.
- `Double loadlimit:` This indicates the load limit for the degree based formula discussed in section 2.3.2.3.

- `m,n,alpha` and `gamma`: Values of the variables required in the formula for the degree based algorithm discussed in section 2.3.2.3.
- `partition`: This method uses the source vertex ID and the destination vertex ID for computing the partition ID according to the Degree based algorithms discussed in section 2.3.2.3.
- `compute`: This method is used to compare the values of the list `partitionsCount` for each partition, and calculate the partition ID based on the Degree based algorithm. This partition ID is sent to the `partition` method, which updates different instance variables and returns the partition ID for assigning the input edge to that partition.
- `getValue`: This method computes the values of the list `partitionsCount`.

Listing 4.8: Degree Based Partitioner Class Implementation

```
//DegreeBased.java
private static class DegreeBasedPartitioner<K, EV, T> implements
    Partitioner<T> {
   private static final long serialVersionUID = 1L;
   CustomKeySelector<T, ?> keySelector;
   private final Table<Long, Long, Long> degree =
       HashBasedTable.create(); //for <partition.no, vertexId, Degree>
   private final List<Double> load = new ArrayList<>(); //for load of
       each partiton
   private final List<Long> partitionsCount = new ArrayList<>();
   private Long k; //no. of partitions
   private Double loadlimit = 0.0;
   private int m = 0; // no. of edges
   private int n = 0;
   private double alpha = 0; //parameters for formula
   private double gamma = 0;

   public DegreeBasedPartitioner(CustomKeySelector<T, ?> keySelector,
       int n, int m) {...}

   @Override
   public int partition(Object key, int numPartitions) {...}

   public int compute(Long source, Long target) {...}

   public int getValue(Long source, Long target, int p) {...}

}
```

While partitioning an edge stream using the Degree based partitioner, first the `partition` method is called. This method gets the source vertex ID and the destination vertex ID from the stream using the `CustomKeySelector`. After this, the

`getValue` method is used to get the list of numbers (`List<Long> partitionsCount`) indicating the number of end vertices of the edge present in each partition. This list is returned to the `partition` method. Later, this list is used for the Degree based algorithm formula in the `compute` method, which returns the calculated partition ID to the `partition` method. The `partition` method then returns this partition ID for placing the input edge in it.

The pseudocode for the `getValue` method is same as that for the Least Cost Incremental Advanced algorithm. The logic for the `partition` method and the `compute` method are different, the pseudocodes for these methods are presented in procedures 8 and 9.

Procedure 8 *partition(key, k)* for Degree Based Partitoner

Input: key: end vertex ID of the input edge, k: total no. of partitions
Output: partition ID

1: **procedure** *partition(key, k)*
2: **var** key2 = keySelector.getValue(key) ▷ // getValue(key) will return the other end vertex of the input edge
3: x = key, y = key2
4: **for** *all partitons i = 1 to k* **do**
5: partitionCount[i] = getValue(x, y, i); ▷ // getValue(x, y, i) returns the number of end vertices present in partition i
end for
6: p = compute() ▷ // compute() returns the ID of partition according to degree partitioning logic
7: Return p ▷ // p is the ID of partition based on degree partitioner

Procedure 9 *compute(x, y)* for Degree Based Partitoner

Input: x and y: end vertices of the input edge.
Output: partition ID

```
1: procedure compute(x, y)
2:     var max = partitioncCount[0]
3:     var p = 0  ▷ max is for finding highest value in partitioncCount[0] and p is
   the partition ID for that value
4:     x = key, y = key2
5:     for all partitons i = 1 to k do
6:         if max < partitioncCount[i] and load[i] < loadlimit then
7:             max = partitioncCount[i]
8:             p = i
           end if
9:         else if max = partitioncCount[i] and max=1 then
10:            if degree table contains x and y then
11:                I(v) = d(v) + Z Subject to      |P_i| ≤ load limit
12:                Z = αγ|P_i|^(1−γ)                  ▷ // here v is end vertex x or y
13:                max = smaller value of I(v) for vertex x or y
14:                p = i
               end if
15:            else if degree table contains either x or y in only one partition i then
16:                max = partitioncCount[i]
17:                p = i
18:            end else if
19:            else if degree table contains either x or y in more than one partitions then
   ▷ for these partitions compute I(v) for the vertex x or y present in them
20:                I(v) = d(v) + Z Subject to      |P_i| ≤ load limit
21:                Z = αγ|P_i|^(1−γ)
22:                max = smaller value of I(v) for partiton i or p
23:                p = i
24:            end else if
25:        end else if
26:        else if max = partitioncCount[i] and max=0 then
27:        find the partition p with lowest load
28:        end else if
29:        Return p
30:    end for              ▷ // p is the ID of partition based on degree partitioner
```

4.5 Post-Partitioning

After partitioning the stream, the graph approximation algorithms are applied on this graph stream to check whether the custom partitioning has improved the performance of these approximation algorithms or not.

Shown in Figure 4.2 is the whole process, where first the input stream is partitioned using the partitioners. This partitioned stream is then subjected to other graph stream processing algorithms like connected component that run parallel in each partition using the `fold` method. During the `fold` operation, the intermediate results for each partition are computed. Then the `reduce` method is used to combine the result of different partitions to get the overall result for the algorithm. In our evaluation section we will show how these `fold` and `reduce` operations are affected by our custom partitioners.

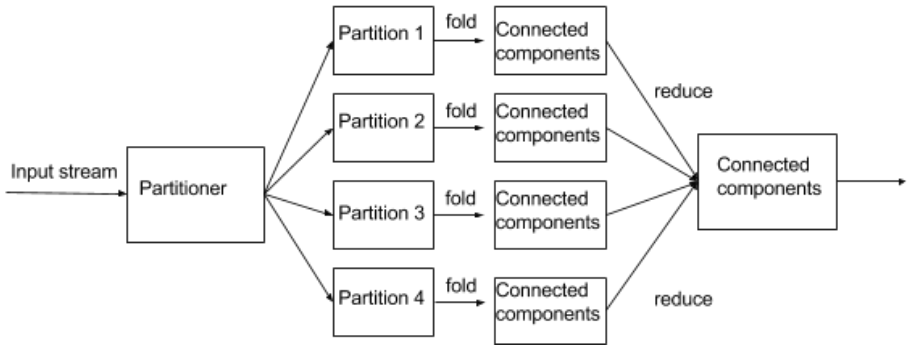


Figure 4.2: Work Flow

Chapter 5
5 Evaluation

The partitioning algorithms we implemented consist of both vertex stream partitioning and edge stream partitioning algorithms. The aim of our experiments was firstly, to measure the partitioning quality of different partitioning algorithms and then to compare their effect on other graph processing algorithms. We cannot compare certain partitioning quality metrics of edge stream partitioning algorithms with vertex stream partitioning algorithms. This is because edge stream partitioning has different quality metrics to be measured than vertex stream partitioning, so we measured them separately and compared the edge stream partitioning algorithms with each other and the vertex stream partitioning algorithms with each other.

The metrics that we measured for the vertex stream partitioning algorithms are the execution time, the edge-cuts and the load ratio. The aim of this was to check which algorithm works better for reducing the edge-cuts and balancing the load across the partitions, while partitioning a vertex stream. Similarly, for edge stream partitioning algorithms we measured the execution time, the replication factor and the load ratio. The aim of this was to check which algorithm reduces the replication of vertices and balances the load across the partitions, while partitioning an edge stream.

After partitioning, the graph processing algorithms we ran on the partitioned streams include connected components and bipartiteness check. We measured how the custom partitioning methods have improved the performance of these graph processing algorithms as compared to the default hash partitioning. We did not expect to improve the partitioning time as the hash partitioner is faster, but there are other metrics like the amount of memory space used and convergence to the final result during the execution of other graph stream processing algorithms, which we expected to get improved with the custom partitioning methods.

All our experiments were performed in a distributed set-up, using large input data sets. The details of the input streams and the environmental setup used in our experiments are discussed in the next sections.

5.1 Input Data Streams

We have used synthetic graphs generated from the Apache Flink Gelly API [16]. version: 1.1.

The type of graphs we used are complete graphs and power-law graphs. In a complete graph there is an edge between all possible pair of nodes, as shown in Figure 5.1. We used this graph for measuring the execution time of the partitioning algorithms.

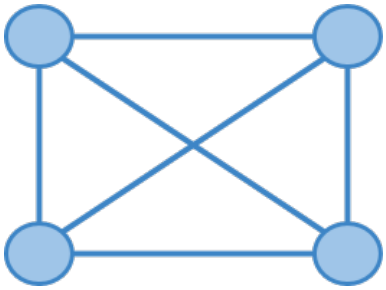


Figure 5.1: Complete Graph

The power-law graphs we used are generated using the *R-Mat (Recursive Matrix) model* [27]. These graphs are used in the experiments for measuring the partitioning quality metrics which include the edge-cut, replication factor and the load ratio.

Undirected graphs are used for our experiments but the algorithms can also work for directed graphs. In an undirected graph for every edge, that is from the source vertex to the destination vertex, there is a matching edge from the destination vertex to the source vertex. Moreover, the duplicate edges were removed in the settings provided by the API. The stream order is kept *Random* for all the experiments, the reason for this order is discussed in section 4.1.

5.2 Experimental Setup

We ran all the experiments in a distributed set-up using the Flink cluster. The Flink cluster had a job manager and two task managers. Each task manager had 4 slots for parallelism, making the total parallelism equal to 8. Moreover, the task managers used a heap space of 32 gigabytes with 8192 network buffers. The network buffers control the network throughput when the task manager and the job manager communicate with each other over the network. The streaming source we used was

a file. This file was placed on a shared HDFS [14] cluster. All the experiments are reproducible using these input files.

We used total two machines for our experiment, both with an Intel(R) Xeon(R) CPU @ 2.80GHz and a 44 GB memory. Each machine contains 2 nodes, where each node has 6 cores; hence making a total of 12 cores on each machine. Machine 1 had task a manager and a job manager, and machine 2 only had a task manager.

5.3 Partitioning Algorithms

In this section we discuss the different quality metrics measured for the partitioning algorithms. The number of partitions in all the experiments is fixed to 4. This number can be varied for measuring the effect of the number of partitions on the algorithms. All the experiment were performed three times, the average of these three readings is presented in the graphs.

5.3.1 Execution Time

The execution time for the partitioning algorithms is measured in a distributed environment using a complete graph as input. The size of the input graph ranges from the smallest graph containing 1×10^7 edges to the largest graph containing 7×10^7 edges. The input stream generated was an edge stream. This stream was converted to a vertex stream for the vertex stream partitioning algorithms. We measured the execution time in seconds for both the vertex and the edge stream partitioning algorithms as shown in Figure 5.2.

The execution time measured is subject to change depending upon the cluster usage. Therefore, these readings can vary. However, we can observe that the execution time increases linearly for Fennel, Linear Greedy and the Degree based partitioner. For Least Cost and Least Cost Advanced it shows a behaviour close to exponential.

The Least Cost Incremental algorithm performs worse as its execution time tends to reach almost 8000 seconds for partitioning 7×10^7 edges. Least Cost Advanced is also bad in terms of the execution time, because it takes almost 6000 seconds to partition a graph of 7×10^7 edges. Fennel and Linear Greedy are almost close to each other, their execution time is below 1000 seconds for the largest input graph. The Degree based algorithm is fast compared to other algorithms; however, there is a sudden increase in the execution time for processing the graph of 7×10^7 edges.

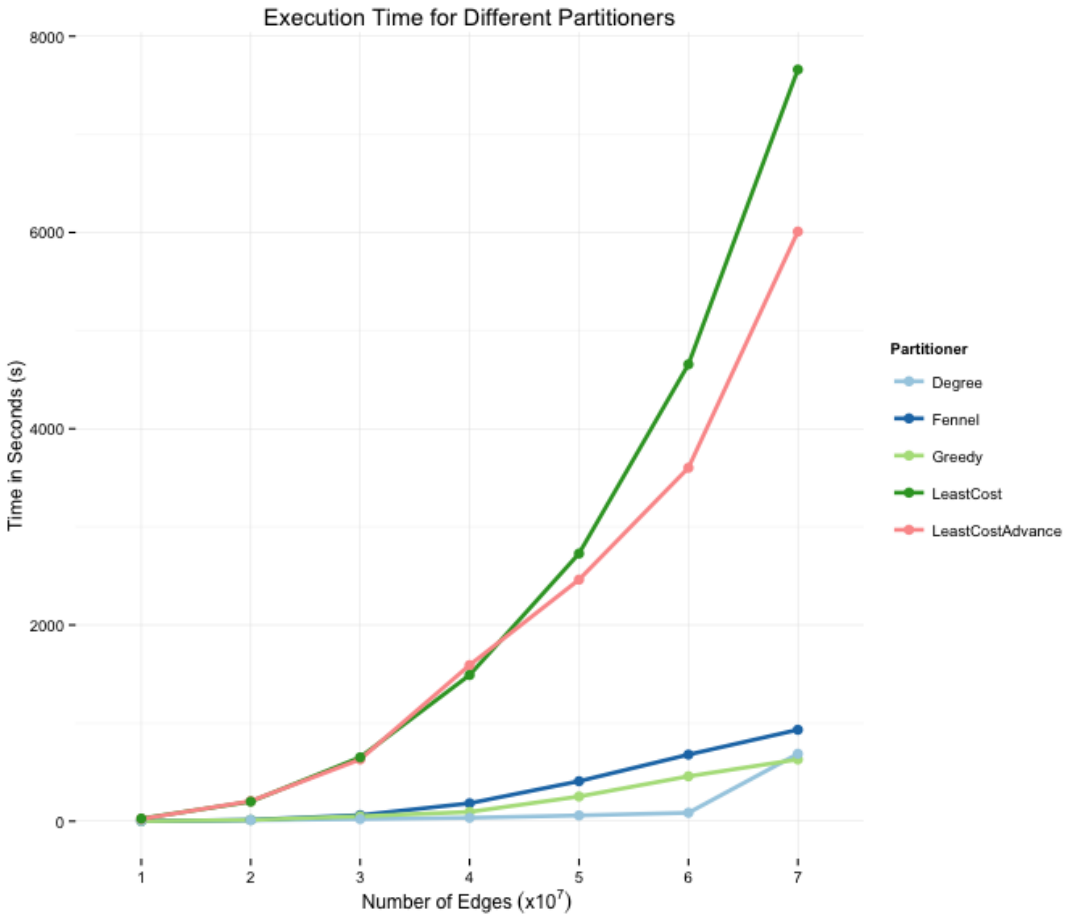


Figure 5.2: Execution Time

The reason for Linear Greedy and Fennel to have less execution time than Least Cost and Least Cost Advanced could be the type of input stream used for processing. The vertex stream consists of the vertex arriving with all its neighbors information; hence making the size of stream smaller compared to the edge stream where each vertex arrives only with one neighbor at a time. In short, as the number of edges >> the number of vertices, this makes the edge stream partitioning time consuming as each edge is processed one by one, whereas in the vertex stream partitioning vertices are partitioned one by one taking less time than the former. On the contrary to what we discussed about the edge stream and the vertex stream, the Degree based partitioner is fast, even though it works with the edge stream. This could be because of the implementation logic it uses. For example, Linear Greedy and Fennel have to go through all the partitions to check how many neighbors are present in them, whereas the degree based partitioner just checks the degree of the end vertices without going through all the partitions. It uses a `Table <partition ID, vertex ID, degree> degree` , which takes the partition ID and the vertex ID as input and returns the degree for that vertex as output.

5.3.2 Edge-Cut

The edge-cut is evaluated for the vertex stream partitioning algorithms that include Linear Greedy and Fennel. This metric is used to check the partitioning quality by measuring the fraction of edges cut from the resulting partitions. The formula for measuring fraction of edges cut is:

$$Fraction\ of\ edges\ cut = \frac{No.\ of\ edges\ cut\ by\ the\ partitions}{Total\ no.\ of\ edges} \tag{5.1}$$

The input graphs used for this experiment are the power-law graphs. The smallest graph contains 1×10^5 vertices and the largest graph contains 5×10^5 vertices. The R-Mat (Recursive Matrix) model generated these graphs with the power-law degree distribution.

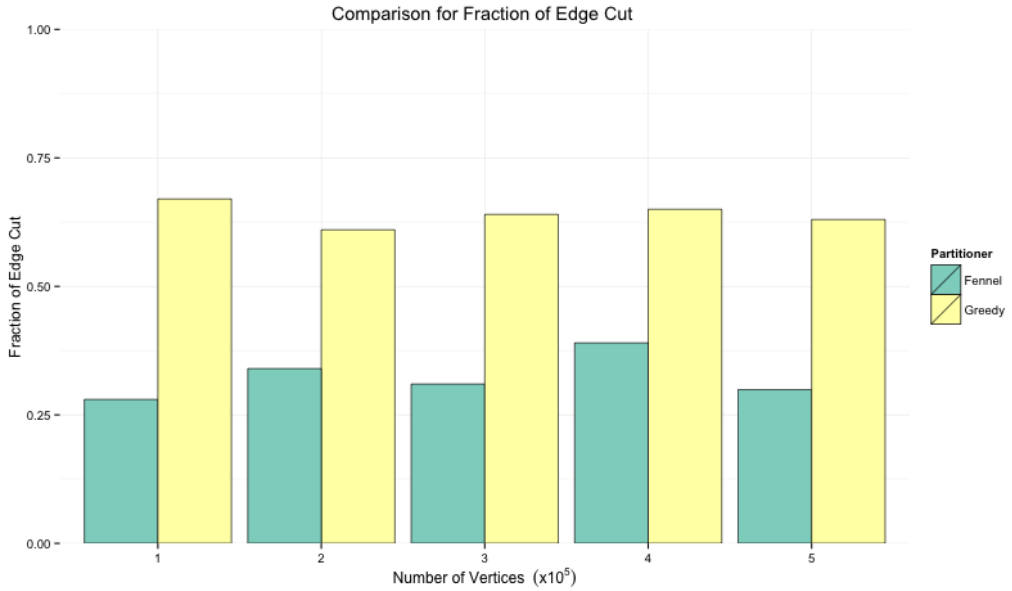


Figure 5.3: Edge-Cut

From Figure 5.3, Fennel shows less fraction of edges cut than Linear Greedy. The fraction of edges cut for Fennel is approximately between 0.27 - 0.38, whereas for Linear Greedy it is approximately between 0.6 - 0.65. We can say that Fennel has less edge-cuts than Linear Greedy according to our results.

Fennel shows better results than Linear Greedy because in the partitioning logic of Fennel, the non-neighbors of the input vertex are also considered along with its neighbors, whereas for Linear Greedy only neighbors are considered and load balancing is done based on the penalty factor that does not do as good as Fennel.

5.3.3 Replication Factor

The replication factor is evaluated for the edge stream partitioning algorithms. These edge stream partitioning algorithms are Least Cost Incremental, Least Cost Incremental Advanced and the Degree based partitioner. During edge stream partitioning, some vertices are replicated in the partitions based on the distribution of edges, creating vertex-cuts. The partitioning quality depends on this replication of vertices as the communication cost increases with the increase in replication. We have discussed in detail the communication cost for vertex-cuts in section 2.1.2.

The replication factor is calculated using the following formula:

$$Replication\ factor = \frac{Total\ copies\ of\ vertices}{Total\ no.\ of\ vertices} \tag{5.2}$$

The input graphs used for this experiment are the power-law graphs. The smallest graph contains 1×10^5 vertices and the largest graph contains 5×10^5 vertices.

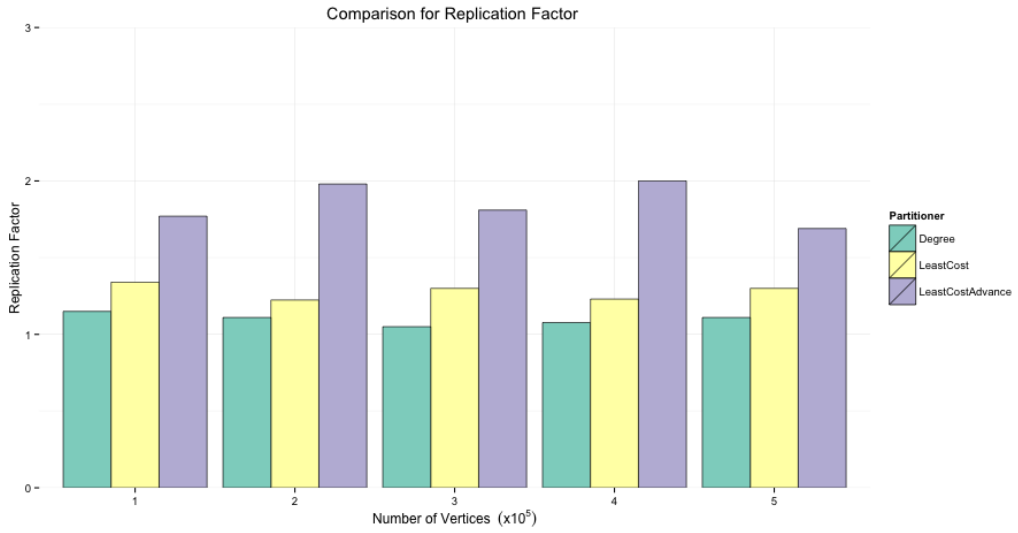


Figure 5.4: Replication Factor

In Figure 5.4, the replication factor for Least Cost Incremental Advanced is the highest compared to others, with values from 1.8 to 2. A replication factor of value 2 means that half of the vertices are replicated. After this, Least Cost Incremental has a replication factor of value between 1.3 to 1.4. The Degree based partitioner has a replication factor lower as compared to the other two algorithms, with values ranging between 1.1 to 1.2. A replication factor of value close to 1 means one fourth of the vertices are replicated.

According to the power-law of degree distribution there are more low-degree vertices than the high-degree vertices. Therefore, the Degree based partitioner shows less replication compared to the others, because it creates vertex-cuts for high degree vertices. On the other hand, Least Cost and Least Cost Advanced do not consider the degree information during partitioning; hence, they can create vertex-cuts for the low-degree vertices, which are more in number than the high-degree vertices, so they result in creating more replicas of the vertices. Furthermore, the Least Cost algorithm does not consider the load balancing while partitioning the edges; it tries to assign the input edge to the partitions containing both or at least one of its end

vertices. Hence, it shows a lower replication factor compared to Least Cost Advanced, which considers the load balancing along with the partitions containing end vertices of the input edge during partitioning.

5.3.4 Load Balancing

To measure how well the load is balanced between the partitions we calculated the normalized load for the highest loaded partition. The normalized load is calculated using the following formula:

$$Normalized\ load = \frac{Load\ on\ highest\ loaded\ partition}{\frac{n}{k}} \tag{5.3}$$

where,
n = the size of the input, which is the number of edges for an edge stream partitioner and the number of vertices for a vertex stream partitioner.
k = the total number of partitions.

The input graph used for this experiment is a power-law graph containing 5×10^5 vertices. The edges are 15 times the number of vertices. Table below shows the normalized load value for the algorithms.

Algorithm	Normalized load
Linear Greedy	1.05
Fennel	1.02
Least Cost Incremental	1.08
Least Cost Incremental Advanced	1.03
Degree Based	1.12

Table 5.1: Normalized Load Value for Partitioning Algorithms

The normalized load for all the partitioning algorithms is shown in Table 5.1. The normalized load value 1 means that the input is equally distributed between the partitions. The closer the value is to 1, the better. Normalized load for Fennel has the lowest value that is 1.02, which is close to 1. This means Fennel is good in terms of balancing the load across the partitions. After Fennel, Least Cost Incremental Advanced and Linear Greedy are good with values 1.03 and 1.05. Least Cost Incremental has value 1.08 for the normalized load, whereas, Degree Based

is the worst for balancing load with value 1.12, which is the highest amongst the all.

The Degree based partitioner and Least Cost Incremental give very low priority to load balancing during partitioning; therefore, they have higher value of normalized load compared to other partitioning algorithms. Fennel shows very good results because its partitioning logic contains parameters, discussed in section 2.3.1.2, to achieve a good load balancing along with less edge-cuts.

5.4 Post-Partitioning

After partitioning, the graphs stream processing algorithms which are executed on the input stream are for finding connected components and doing bipartiteness check. These algorithms are one-pass algorithms. During the execution of these algorithms, a `fold` operation is performed on the partitions, this operation creates aggregate states in all the partitions. These aggregates states are then merged together in the `reduce` operation, as discussed in section 4.5. To measure the effect of partitioning on these algorithms we measured the size of aggregate states and the convergence percentage for each `reduce` operation during the execution of these algorithms.

5.4.1 Size of Aggregate States

The sizes of aggregate states are recorded and the average value of these sizes is computed. The average size value is then compared with the average size of aggregate states that was recorded after using the default hash partitioner. We calculated the percentage of reduction in the size of aggregate states for this experiment using the following formula:

$$Reduction\ in\ Size\ \% = \frac{Av_{Sh} - Av_{Sc}}{Av_{Sh}} \times 100 \tag{5.4}$$

where,
Av_{Sh} = The average size of aggregate states after hash partitioning
Av_{Sc} = The average size of aggregate states after custom partitioning

The input graph used for this experiment is the power-law graph with size ranging from the smallest graph containing 1×10^5 vertices to the largest graph containing 5×10^5 vertices.

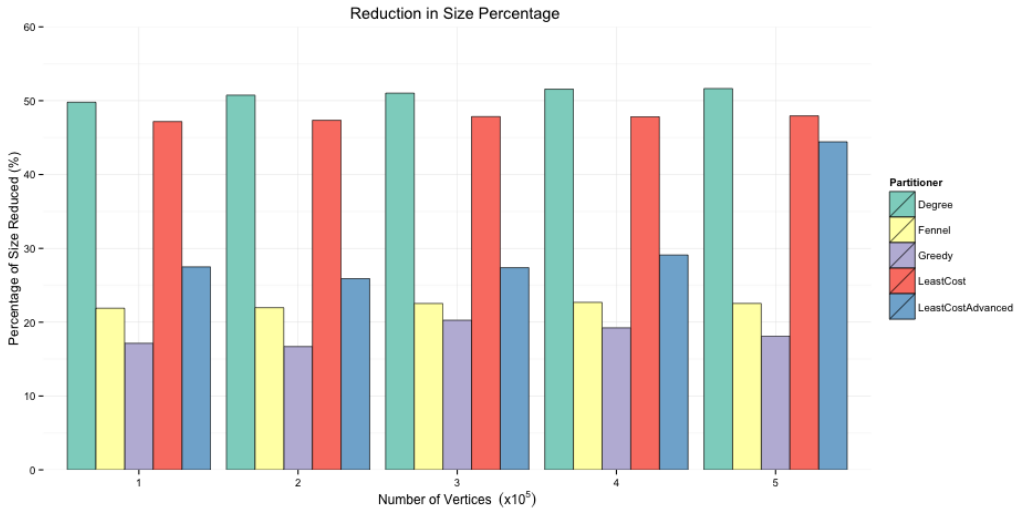


Figure 5.5: Percentage of Reduction in Size of Aggregate States

We have recorded the percentage of reduction in the size of aggregate states for different sizes of the input graph. From Figure 5.5, we can say that the increase in the size of the input graph has not affected the reduction in size of the aggregate states much. The vertex partitioning algorithms, Fennel and Linear Greedy, did not decrease the size of aggregate states much because of the edges that were replicated for converting the partitioned vertex stream to the edge stream, as discussed in chapter 4. The Degree based partitioner and Least Cost Incremental performed better than others by reducing on average up to 50% and the later up to 48% of the aggregate states' size approximately. On the other hand, Least Cost Advanced did not do as good as them by reducing on average up to 27% of the size approximately. This reduction in the size of aggregate states helps saving the memory space during the execution of the graph processing algorithms.

The default partitioning method creates more vertex-cuts than the custom partitioning methods because it assign the edges at random. These vertex-cuts increase the replicas of the vertex; therefore, the size of aggregate states, discussed above, increases. The Degree based partitioner and Least Cost Incremental showed good results compared to others, because they have a very less replication factor. Fennel and Linear Greedy did not show good results, the reason could be the conversion of the vertex stream to the edge stream after partitioning. During this conversion, for each edge creating an edge-cut, a vertex copy was created in the partitions containing the end vertices of the edge, as discussed in section 4.5. This increased the replication of vertices; hence resulting in a lower percentage of reduction in size of the aggregate states.

5.4.2 Convergence

The convergence towards the final state is computed during each `reduce` operation while performing the graph processing algorithms after partitioning. The `reduce` operation is explained in section 4.5. To measure the percentage of data converged, we used the following formula:

$$Percentage\ of\ data\ converged = \frac{E_f - E_i}{E_f} \times 100 \tag{5.5}$$

where,
E_f = The no. of elements in the final state
E_i = The no. of elements in the intermediate state

Total four `reduce` operations are performed while executing the graph processing algorithm; after 4^{th} `reduce` operation, we get the final state or the result of the algorithm being executed on the graph. During each `reduce` operation, more elements are added to the state called the intermediate state. We compare the number of elements in the intermediate state with ones in the final state. The input graph used for this experiment is the power-law graph. The result for the experiment of the graph containing 1×10^5 vertices is shown in Figure 5.6.

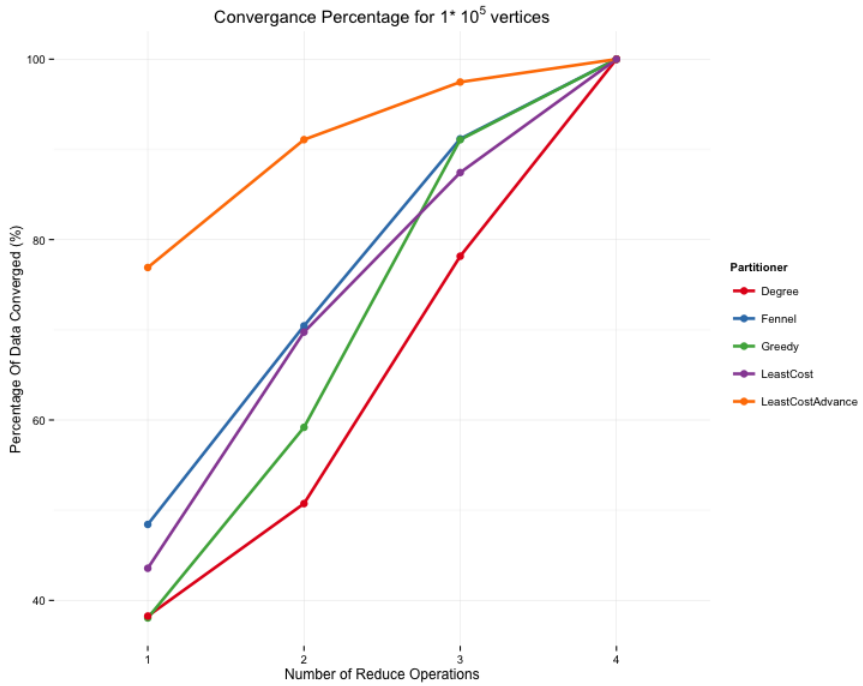


Figure 5.6: Percentage of Data Converged for 1×10^5 *vertices*

We have seen that the Degree based algorithm has the lowest convergence percentage, whereas, Least Cost Incremental Advanced has the convergence percentage highest compared to the others. Least Cost and Fennel are very close to each other. Moreover, Greedy also shows a very little variation from them. The reason for the Degree partitioner having a lower convergence percentage compared to others could be because of the way we measured the convergence rate, i.e, by counting the number of elements in the states. We have seen in section 5.4.1 that Degree based partitioner reduces the size of aggregate states more as compared to other algorithms; therefore, resulting in a low convergence rate. This experiment is also performed for other input graphs sizes, as shown below in figures 5.7, 5.8, 5.9 and 5.10.

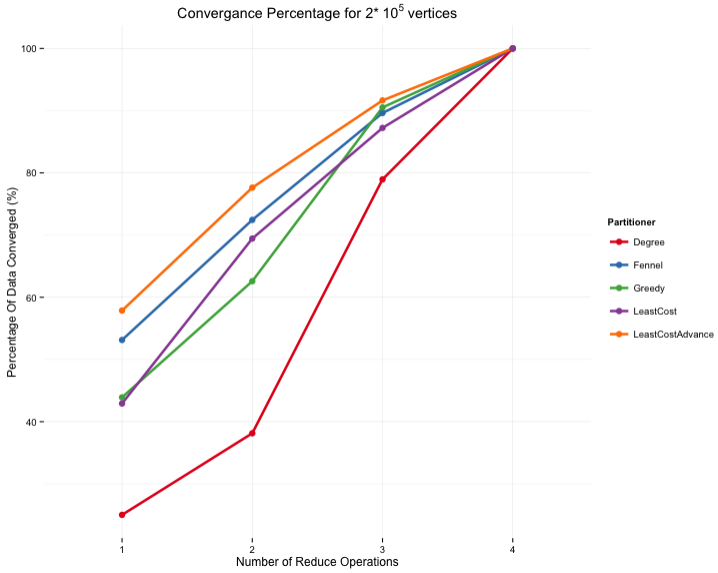
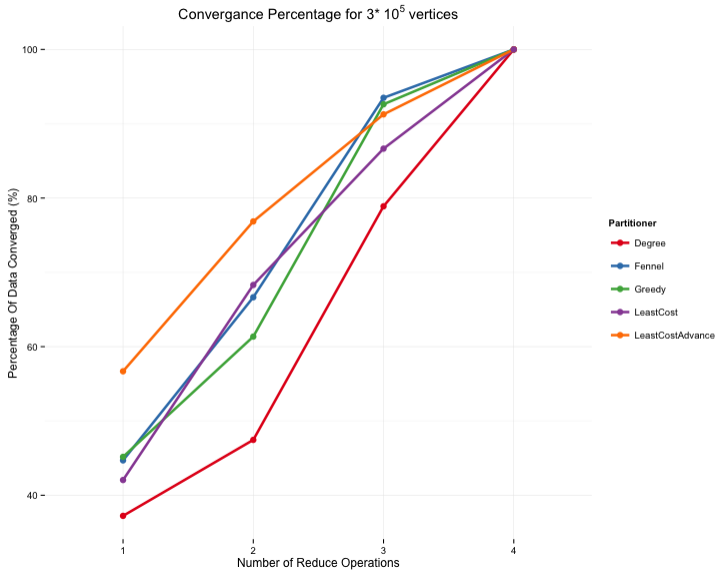


Figure 5.7: Percentage of Data Converged for 2×10^5 *vertices*



[H]

Figure 5.8: Percentage of Data Converged for 3×10^5 *vertices*

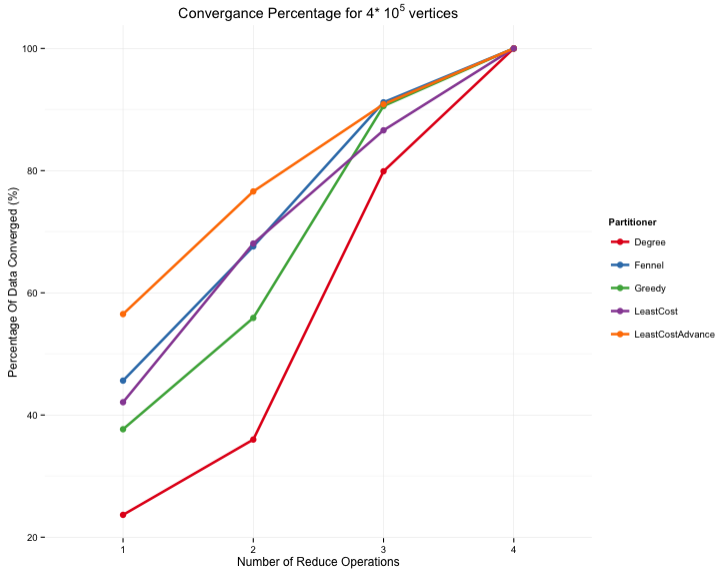


Figure 5.9: Percentage of Data Converged for 4×10^5 *vertices*

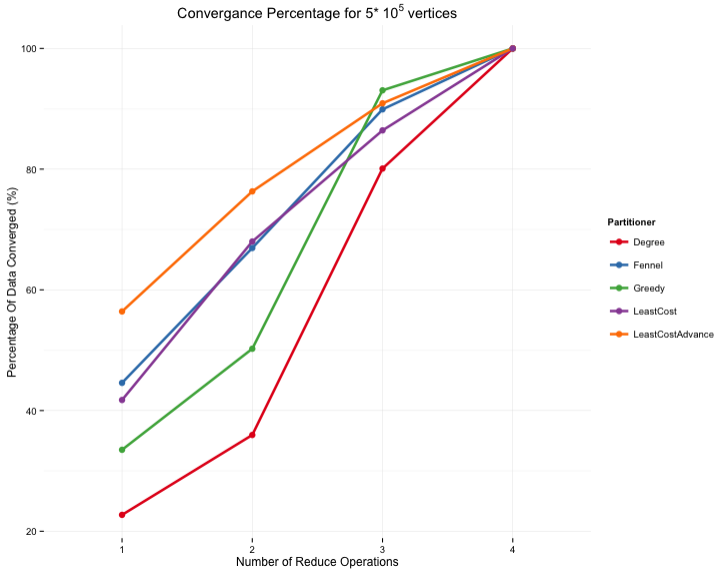


Figure 5.10: Percentage of Data Converged for 5×10^5 *vertices*

The convergence percentage does not change much with the increase in the size of the input graphs, as can be seen in figures 5.7, 5.8, 5.9 and 5.10.

5.5 Evaluation Summary

Our first set of experiments is discussed in section 5.3. In that section, firstly, we have compared different vertex stream partitioning algorithms in terms of their partitioning quality metrics which include the edge-cuts and the load balancing. Secondly, we have compared the partitioning quality of different edge stream partitioning algorithms by measuring the replication factor and the load balancing.

According to the results of our experiments for vertex stream partitioning, Fennel showed lower edge-cuts and a better load balancing compared to Linear Greedy. This is because of the partitioning logic of Fennel; it keeps a good balance between maximizing the number of neighbors of the input vertex and minimizing the number of non-neighbors. This interpolation between neighbors and non-neighbors helps to achieve a good balance between decreasing the edge-cuts and balancing the load across the partitions. However, Linear Greedy works by placing the input vertex in the partition containing the most number of its neighbors; meanwhile, penalizing the partitions if the load is high. This approach helps reducing the edge-cuts and achieving an average load balancing among the partitions.

On the other hand, for edge stream partitioning, the Degree based partitioner outperformed others by having a lower value of the replication factor for vertices. This

means that using the degree based approach fewer replicas of the vertices are created across the partitions, i.e, less vertex-cuts. The reason for this reduced number of vertex-cuts is that the degree based approach is based on creating vertex-cuts for the high degree vertices compared to the low degree vertices. Since the high degree vertices are less in number than the low degree vertices in power-law graphs so this approach proved better than the others for reducing the vertex-cuts. However, it does not show good results for load balancing because it prioritizes more on reducing the vertex-cuts. Other algorithms include Least Cost Incremental and Least Cost Incremental Advanced. Least Cost Incremental tries to place the edge in the partition containing the maximum number of its end vertices. This approach is good in reducing the vertex-cuts but it does not takes into account the load balancing factor and shows a high normalized load value in Table 5.1. Therefore, to improve this we implemented the Least Cost Incremental Advanced algorithm that tries to balance the load along with the partitioning logic of Least Cost Incremental. This algorithms not only places the input edge in the partition containing the maximum number of its end vertices but also penalizes the partitions based on their load. Thus, it shows a good load balancing compared to the other two edge stream partitioning algorithms, but with more vertex-cuts.

Algorithm	Optimization
Linear Greedy	Edge-cuts
Fennel	Load balancing and Edge-cuts
Least Cost Incremental	Vertex-cuts
Least Cost Incremental Advanced	Load balancing
Degree Based	Vertex-cuts

Table 5.2: Evaluation Table

The evaluation summary of these partitioning algorithms in shown in Table 5.2. According to this table, for vertex stream partitioning, Linear Greedy is good at minimizing the edge-cuts, whereas Fennel is good at load balancing and minimizing the edge-cuts. Similarly, for edge stream partitioning, Least Cost Incremental and Degree based are good at minimizing the vertex-cuts, whereas, Least Cost Incremental Advanced is good at load balancing. As a result of our discussion based on the experimental results we can conclude that Fennel is a good choice for partitioning a vertex stream, whereas, for an edge stream it depends on the requirements. If lower vertex-cuts are required then the degree based approach is good and if a good load balancing is required then the Least Cost Incremental Advanced algorithm is good.

After partitioning, the effects of these partitioning algorithms were measured on other graph processing algorithms in section 5.4. We measured the percentage of

reduction in the size of aggregate states and the convergence of the intermediate states towards the final state. We compared the percentage of reduction in the size of aggregate states and concluded that the Degree based partitioner outperforms all by reducing the size up to 50%; hence, saving a lot of memory space. This is because the Degree based partitioner showed a lower value for the replication factor (vertex-cuts) after partitioning compared to others, so less replicas of the vertices are created resulting in few elements being stored in the aggregate states. However, the percentage of convergence is lowest for the Degree based partitioner and highest for the Least Cost Incremental Advanced because the number of elements in the aggregate states are lowest for the Degree based partitioner. The convergence to final results could be measured in other useful ways. Therefore, we have suggested some other useful metrics that can be used to measure the effect of different customised partitioning methods on the graph stream processing algorithms in our future work section 6.1.

Chapter 6
6 Conclusion

Our thesis work provides a detailed study of different streaming graph partitioning algorithms and their implementation. We have measured and evaluated the partitioning quality metrics for these algorithms by partitioning power-law graphs that are highly skewed graphs. Moreover, we introduced a Degree based algorithm for partitioning the power-law graphs with an aim to improve the partitioning quality by reducing the replication of vertices in the partitions. Furthermore, our work consists of measuring the effect of these partitioning algorithms on different graph stream processing algorithms.

We have concluded that for edge stream partitioning, the Degree based partitioner works better than the other partitioning methods we implemented in terms of reducing the replication factor, but does not work well for the load balancing. Additionally, this algorithm outperforms the others by reducing the size of aggregate states up to 50% while executing the graph stream processing algorithms.

For the vertex stream partitioning, Fennel is better than Linear Greedy in every aspect; It has fewer edge-cuts and a good normalized load value. However, both of these algorithms did not do as good as the edge stream partitioning algorithms while executing the graph stream processing algorithms, because of the edge replication done for converting the vertex stream to the edge stream after partitioning. This replication increased the replicas of vertices in the partitions. Therefore, they show a lower percentage of reduction in the size of the aggregate states compared to the edge stream partitionig algorithms.

Finally, we can conclude that compared to random (hash) partitioner, custom partitioning methods not only improve the partitioning quality, but also help in saving the memory space used for other graph stream processing algorithms. The reduction in size of aggregate states indicates that some memory is saved during the execution of the graph stream processing algorithms after custom partitioning.

6.1 Future Work

We have shown only few cases in our experiments as an initial step for measuring the partitioning quality metrics like the edge-cuts, the replication factor and the load ratio for different streaming graph partitioning algorithms. Moreover, in our work we used synthetic graphs with an input stream of random order. This work can be extended by using real world graphs and different order of the input streams. These orderings include: the breadth-first search and the depth-first search orderings. Their effect on partitioning methods can be measured. Furthermore, the number of partitions can also be varied, which in our case is fixed to four, and their effect can also be interesting to observe.

We have only implemented few algorithms that we found were good in terms of reducing the edge-cuts or the vertex-cuts and balancing the load across the partitions. However, there are other stream partitioning algorithms, like HDRF [8] that can be implemented for this study. Additionally, the effect of the partitioning algorithms on other graph stream processing algorithms is interesting to measure. We measured the reduction in the size of aggregate states and the convergence, there can be other useful metrics to look for. For example, the maximum size of the aggregate states instead of the average size and the number of steps performed during the `reduce` operation.

Bibliography

[1] G. Malewicz, M. H Austern, A. JC Bik, J. C Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data,ACM, 2010, pages 135-146.

[2] L. G. Valiant. A Bridging Model for Parallel Computation. Published in magazine Communications of the ACM, Volume 33 Issue 8, 1990 Aug, pages 103-111.

[3] Apache Giraph. Link: `http://giraph.apache.org`. Accessed: 24-07-2016.

[4] `https://www.di.ens.fr/~fbourse/publications/Balanced%20Graph%20Edge%20Partition.pptx`. Accessed: 24-07-2016.

[5] U. Kang, C. E. T., and C. Faloutsos. Pegasus: A peta-scale graph mining system. In ICDM, 2009, pages 229-238.

[6] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In ACM KDD, 2012, pages 1222-1230.

[7] C. E. Tsourakakis, C. Gkantsidis, B. Radunovi, M. Vojnovi. FENNEL: Streaming Graph Partitioning for Massive Scale Graphs. In Proceedings of the 7th ACM International conference on Web search and data mining. ACM, 2014, pages 333-342.

[8] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, Hdrf: Stream-based partitioning for power-law graphs. In Proceedings of the 24th ACM International on Conference on Information Management. ACM, 2015, pages 243-252.

[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, Powergraph: Distributed graph-parallel computation on natural graphs. Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation, 2012, pages 17-30.

[10] J. D. Bali, V. Kalavri, P. Carbone. Streaming Graph Analytics Framework Design, 2015. `https://github.com/vasia/gelly-streaming`. Accessed: 24-07-2016.

[11] Apache Flink. `https://github.com/apache/flink`. Accessed: 24-07-2016.

[12] F. Bourse, M. Lelarge, and M. Vojnovi. Balanced Graph Edge Partition. In Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining, 2014 Feb, pages 1456-1465. Link: `https://www.microsoft.com/en-us/research/publication/balanced-graph-edge-partition/`. Accessed: 24-07-2016.

[13] S. Muthukrishnan. Data Streams: Algorithms and Applications. Foundations and Trends in Theoretical Computer Science, 2005, pages 117-236.

[14] Model K. Ahn, S. Guha: Graph Sparsification in the Semi-Streaming, May 2009. Link: `http://repository.upenn.edu/cgi/viewcontent.cgi?article=1427&context=cis_papers`. Accessed: 24-07-2016.

[15] Apache Hadoop. Link: `http://hadoop.apache.org/`. Accessed: 24-07-2016

[16] Flink Gelly. Link: `https://github.com/apache/flink/tree/master/flink-libraries/flink-gelly`. Accessed: 24-07-2016.

[17] `https://flink.apache.org/news/2015/08/24/introducing-flink-gelly.html`. Accessed: 24-07-2016.

[18] Iterative graph processing. Link: `https://ci.apache.org/projects/flink/flink-docs-master/apis/batch/libs/gelly.html#iterative-graph-processing`. Accessed: 24-07-2016.

[19] M. Kim, K. S. Candan. SBV-Cut: Vertex-Cut based Graph Partitioning using Structural Balance Vertices. Data and Knowledge Engineering, Volume 72, February 2012, pages 285-303, ISSN 0169-023X. Link: `http://dx.doi.org/10.1016/j.datak.2011.11.004`. Accessed: 24-07-2016.

[20] A. Abou-Rjeili and G. 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.

[21] K. Lang. Finding good nearly balanced cuts in power law graphs. Technical Report , Yahoo! Research Labs, 2004.

[22] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. Internet Mathematics, 2008, pages 29-123.

[23] D. Gregor and A. Lumsdaine. The parallel bgl: A generic library for distributed graph computations. In Proceedings of POOSC, 2005.

[24] G. Karypis and V. Kumar. Multilevel graph partitioning schemes. In ICPP, pages 113-122, 1995.

[25] World Wide Web. Link: `http://www.worldwidewebsize.com/`. Accessed: 24-07-2016.

[26] `https://techcrunch.com/2016/04/27/facebook-q1-2016-earnings/`. Accessed: 24-07-2016.

[27] D. Chakrabarti, Y. Zhan, C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. SDM, 2009, pages 442-446.

[28] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. Journal of Parallel and Distributed Computing, Volume 48 Issue 1, 1998 Jan, pages 96-129.

[29] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In High-Performance Computing and Networking, volume 1067 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1996, pages 493-498.

[30] Twitter Platform. Link: `https://dev.twitter.com/`. Accessed: 24-07-2016.

[31] Apache Kafka. Link: `http://kafka.apache.org/`. Accessed: 24-07-2016.

[32] Apache Flume. Link: `https://flume.apache.org/`. Accessed: 24-07-2016.

[33] RabbitMQ. Link: `https://www.rabbitmq.com/`. Accessed: 24-07-2016.

Declaration

I hereby certify that I have written this thesis independently and have only used the specified sources and resources indicated in the bibliography.

Stockholm, 24. July 2016

.....

Zainab Abbas