A streaming graph library for Apache Flink

Ma, X.

Award date:
2018

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
A streaming graph library for Apache Flink

Master Thesis

Xin Ma

Supervisors:

Professor dr. Mykola Pechenizkiy
dr. George Fletcher
Erik de Nooij, IT Chapter Lead Fraude & IAM, ING
Fred Teunissen, IT Specialist, ING

Final 1.1

Eindhoven, August 2018
Abstract

As the study on streaming graph is still in its infancy and most existing tools or libraries are designed and optimized for specific graphs and algorithms, we develop a generic streaming graph library on top of Apache Flink. The goal of this library is to process massive streaming graph events with a high throughput and execute user-defined queries with a low latency.

To efficiently store a streaming graph and get a distribution that is conducive to subsequent queries, our library supports three typical graph partition strategies and one redundancy mechanism. A graph partition strategy is to help determine the partitioning of each streaming graph events, and we implement three strategies including Simple Hash, Incremental Hash and Repartition Hash. Because there is always a trade-off between load balancing and minimum communication cost for a graph partition strategy. Users of the library can choose their preferred strategy to increase the speed of their customized queries. And the redundancy mechanism is to replicate part of the graph events and thus accelerate the corresponding queries. These multiple options make the library more generalized to support different kinds of graph related queries.

In order to facilitate users to implement their own queries, our work introduces an abstract computation model called iterative scatter-gather model. The model includes three phases: Scatter phase can scatter a query into several sub-queries and send them to corresponding graph partitions; Iterate phase executes these sub-queries on each graph partition in parallel; Gather phase aggregates the results from each scattered sub-query. The query will go to the next iteration if it is not complete yet. This computation model is abstracted as two layers: The first layer gives more flexibility where users can implement the three phases by themselves and the second layer further encapsulates the first layer and provides an experimental Cypher-like API. Cypher is a declarative, SQL-inspired language for describing patterns in graphs. In this way, users can easily formulate a graph query even without a specialized background in programming.

Finally, we evaluate the performance of this library. We first test the general performance without targeting any specific query. Furthermore, we execute different graph-related queries against a large-scale graph. By setting different configurations, the library can load graph events with a maximum throughput of 200,000 events per second and execute queries within seconds. The performance is in line with our expectations. At the end, We discuss the pros and cons of our library and give further advice to users of our library.
I would like to thank Prof. Mykola Pechenizkiy and George Fletcher for proposing me the project. I am grateful for their academic support on this project. Additionally, since this project is done during my thesis internship in Internationale Nederlanden Groep (ING), a Dutch multinational banking and financial services corporation, I would also like to express my gratitude to my mentors in ING, Erik de Nooij and Fred Teunissen, for their knowledge in cutting-edge technologies and their applications. I would also like to thank my colleagues in ING, who encouraged me and offered me much practical advice when I was stuck.

Specifically, I want to express my gratitude to my parents, Guangling Zou and Huaiyi Ma, who have been supporting me throughout my education life. Finally, I want to thank all my classmates that helped me during my project, may it be for practical advice, inspirational ideas or listening to my story. Thank you Luis Galdo Seara, Jiapeng Li, Nazly Santos Buitrago, JuanMan Gonzalez, Robin Esposito and Alessandro Terragni.
Contents

1 Introduction 1
  1.1 Motivation ......................................................... 1
  1.2 Goal ................................................................. 1
  1.3 Contributions ................................................... 2
  1.4 Thesis organization ................................................ 3

2 Related work 4
  2.1 Distributed graph processing ........................................ 4
  2.2 Streaming graph processing .......................................... 6
  2.3 Graph processing with Flink ......................................... 11
  2.4 Graph processing with Neo4j ......................................... 14
  2.5 Comparative study ................................................... 15

3 Library foundations 18
  3.1 Streaming property graph model ..................................... 18
  3.2 Abstraction of query model ........................................... 20
  3.3 Iterative scatter-gather model ....................................... 22
  3.4 Graph partition strategy ............................................. 23

4 Design 27
  4.1 Restrictions within the design ...................................... 27
  4.2 Key design decisions ................................................ 27
    4.2.1 Vertex-cut or Edge-cut ......................................... 28
    4.2.2 Implementation of partition strategy ............................. 29
    4.2.3 A redundancy mechanism to accelerate queries ................. 30
    4.2.4 Implementation of iterative scatter-gather model ............. 31
    4.2.5 An experimental Cypher-like Query API ........................ 32
  4.3 Library organization ............................................... 33

5 Evaluation 35
  5.1 Datasets .......................................................... 35
  5.2 Experiment design ................................................... 36
  5.3 Results ............................................................ 38
  5.4 Conclusions ......................................................... 44
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 Conclusions</td>
<td>46</td>
</tr>
<tr>
<td>6.1 Contributions</td>
<td>46</td>
</tr>
<tr>
<td>6.2 Future work</td>
<td>47</td>
</tr>
<tr>
<td>Bibliography</td>
<td>49</td>
</tr>
<tr>
<td>Appendix</td>
<td>53</td>
</tr>
<tr>
<td>A How to use the library?</td>
<td>53</td>
</tr>
<tr>
<td>B Queries used in benchmark</td>
<td>54</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Edge-cut vs Vertex-cut: An edge-cut (a) splits the graph along edges while a vertex-cut (b) splits the graph along vertices. In this illustration we partition the graph across three machines (corresponding to color), taken from [39] . . . . . . . 5
2.2 The power-law distribution of the vertex degree in the real-world dataset in log-log scale, taken from [27] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.3 Asynchronous vertex data processing model, taken from [32] . . . . . . . . . . . . . 8
2.4 Adaptive Partitioning with incremental hash functions: $\eta = 3$ (number of hash functions), taken from [41] . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.5 Deferred Vertex Migration to Ensure Message Delivery. Top: Failed message delivery due to incorrect synchronization. Bottom: Correct delivery. The dashed-red circle indicates when the vertex is in a "migrating" state waiting for one iteration (step) before actually migrating, taken from [38] . . . . . . . . . . . . . . . . . . 10
2.6 As a software stack, Flink is a layered system. The different layers of the stack build on top of each other and raise the abstraction level of the program representations they accept: The runtime layer receives a program in the form of a JobGraph, both the DataStream API and the DataSet API generate JobGraphs through separate compilation processes, taken from [38] . . . . . . . . . . . . . . . . . . 11
2.7 Vertex-centric model: Initially, each vertex has a value of infinite distance, except from the source vertex, which has a value of zero. During the first superstep, the source propagates distances to its neighbors. During the following supersteps, each vertex checks its received messages and chooses the minimum distance among them. If this distance is smaller than its current value, it updates its state and produces messages for its neighbors. If a vertex does not change its value during a superstep, then it does not produce any messages for its neighbors for the next superstep. The algorithm converges when there are no value updates or the maximum number of supersteps has been reached. The figure is taken from [8] . . 13
2.8 Scatter-gather model: In each superstep, each vertex sends a candidate distance message to all its neighbors. The message value is the sum of the current value of the vertex and the edge weight connecting this vertex with its neighbor. Upon receiving candidate distance messages, each vertex calculates the minimum distance and, if a shorter path has been discovered, it updates its value. If a vertex does not change its value during a superstep, then it does not produce messages for its neighbors for the next superstep. The algorithm converges when there are no value updates. The figure is taken from [8] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.9 Gather-sum-apply model: During the Gather phase, we calculate the new candidate distances, by adding each vertex value with the edge weight. In Sum, the candidate distances are grouped by vertex ID and the minimum distance is chosen. In Apply, the newly calculated distance is compared to the current vertex value and the minimum of the two is assigned as the new value of the vertex. Notice that, if a vertex does not change its value during a superstep, it will not calculate candidate distance during the next superstep. The algorithm converges when no vertex changes the value. The figure is taken from [8] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
LIST OF FIGURES

2.10 A typical block in Neo4j. The green node is an employee who works for the pink node, which is a company. The company is located in a city represented by the blue node, taken from [9] ............................................. 15

3.1 An example of vertices linked with edges: The green circle indicates a customer vertex, whereas the purple circle represents customer Bob’s accounts. The word on each edge specifies the edge label. ................................. 19

3.2 A simple example of bank network: The green circles represent customers, and the blue circles are the accounts owned by these customers, the graph is generated by Neo4j [9]. .................................................. 19

3.3 Iterative scatter-gather model: The computation model mainly contains three phases. Scatter phase is to broadcast the graph and query events to the partitions where they should go. Iterate phase is to store the graph events and execute the query according to these stored graph events. Gather phase is to aggregate the query results from different partitions in iterate phase and thus decide if the query enters next iteration. .................................................. 23

3.4 Three different hash partition strategies: Assume these vertices are bank accounts and the edges are transactions. The first strategy is good for calculating how much money a specific account transferred to others. The second strategy is efficient for computing how much money a specific account received from all other accounts. As for the last strategy, it is useful for counting all historical transactions between two specific accounts. .................................................. 24

3.5 Simple Hash Strategy: The blue circles represent the source vertices of each edge, which are used to decide the location of each edge. .................................................. 25

3.6 Comparison between Simple and Incremental Hash Strategy: The left figure is using simple hash strategy, all edges start from vertex 1 are distributed to the first partition. The right figure is using incremental hash function. The capacity for each partition is 4. Therefore, each partition can store at most 4 edges start from vertex 1. Now vertex 1 is using the third hash function. ................................. 25

3.7 An example of repartition hash strategy: The edge events colored red are already loaded. For the left figure, the next incoming edge event is (1,2). Since vertex 2 is already on partition 2, vertex 1 is more likely to migrate to second partition because its neighbor vertex 2 is on that partition. As for the right figure, the next incoming edge event is (2,4), which will be distributed to second partition. Since one of its neighbor vertex 4 is on first partition and another neighbor vertex 5 is on second partition, it will remain stored on the second partition. ................................. 26

4.1 Process logic of Cypher-like Query API: A user-defined query is parsed into a list of edge, each edge is a step within the query. The mappings included in each edge are the detected graph patterns. An edge with hidden structure can contain multiple layers of mappings, otherwise, it only has one layer of mapping. ............... 33

4.2 An overview of Flink visualized plan: The execution plan covered by green circle is data preprocessing. The plan covered by red circle is edge distribution. The brown circle covers the vertex distribution part. The blue circle covers the query result output. The execution plan is generated by Flink plan visualizer. ................................. 34

5.1 Power-law distribution of UK-2007 dataset: The x-axis is the number of out degree for each vertex, y-axis is the frequency. The figure shows low-degree vertices have much higher frequency than high-degree vertices. ................................. 35

5.2 Total throughput with different parallelism: The x-axis is time, y-axis is the events being loaded. The dotted lines show the throughput with different graph partition strategies. The solid lines show the throughput of simple hash partition strategy with different parallelism. ................................. 39

A streaming graph library for Apache Flink
5.3 **Edge distribution with a balanced graph stream**: The x-axis is time, y-axis is the number of edges on each partition. The test is based on a parallelism of 8. So each partition strategy has 8 solid lines in the figure. The orange one represents the simple hash strategy, the blue represents the incremental hash strategy and the green one represents the repartition hash strategy. 39

5.4 **Edge distribution with a skewed graph stream**: The x-axis is time, y-axis is the number of edges on each partition. The test is based on a parallelism of 8. The orange lines represent the simple hash strategy and the blue lines represent the incremental hash strategy. 40

5.5 **Total vertex-cuts under different parallelism with simple hash strategy**: The x-axis is time, y-axis is total number of vertex-cuts. Each line represents different settings of parallelism. 41

5.6 **Total number of vertex-cuts with different partition strategies**: The x-axis is the number of loaded edge events, y-axis is the total number of vertex-cuts. The green bar represents the simple hash strategy which has the minimum vertex-cuts. The blue bar represents the incremental hash strategy. The orange bar represents the repartition hash strategy which has the maximum number of vertex-cuts. 41

5.7 **Four histograms of query time usage with different configurations** 43
# List of Tables

2.1 Comparison between different graph cut approaches ........................................ 6  
2.2 Comparison between different iterative graph computation models, taken from [8] . 12  
2.3 Comparison between batch and stream graph processing, taken from [5] ............ 14  
2.4 Comparison between seven typical graph processing libraries/models .............. 17  

5.1 Performance Statistics under different graph sizes (Unit: ms) using parallelism 8, simple hash strategy and Cypher-like API ............................................. 42  
5.2 Performance Statistics under different query models (Unit: ms) using parallelism 8, simple hash strategy and 100 million edge events ........................................ 43  
5.3 Performance Statistics under different partition strategies (Unit: ms) using parallelism 8, 100 million edge events, parameter-base query model .................................. 44
Listings

2.1 A simple example of the Cypher query. The query will find out all accounts that used to transfer money to any account that has transferred money to Alice.

3.1 Two examples of parameter-based query. The first query is to calculate how much money account 1 has transferred to all other accounts between 2014 and 2019. The second query is to check if there is a transaction path from account 1 to account 4 and the length is no more than 4.

3.2 Examples of Cypher-like query. The first query is to check if there is a path between account 1 and account 10 that the length of this path is no more than 4? The second query is to calculate how much money Bob has received from all other bank accounts?

4.1 Basic Event Class
4.2 Vertex Class
4.3 Edge Class
4.4 Graph storage: Edge distribution
4.5 Graph storage: Vertex distribution

4.6 **KeySelector and HashPartitioner:** These two classes define our customized partition strategy.

4.7 **Edge replication:** A redundancy mechanism for some specific edge events.

4.8 **Query abstraction:** Users should extend the *Query* class and thus define their own query logic.
Chapter 1

Introduction

In the era of big data, massive data are continuously collected from various data sources. These unbounded data brings the needs for real-time analysis. Each piece of data in these data streams can be seen as an event. Among all kinds of data streams, some events are connected with each other, especially the events in the same stream. For example, users of social network can produce large amount of data every day. A typical event can be user Alice adds user Bob as her friend. When another event user Bob adds user Charlie as his friend is collected in the stream, these two events are connected. And we can infer that user Alice may also know user Charlie. Naturally, we find that graph is a good data structure to store such streaming connected events, and thereby we can call it streaming graph which is constantly changing as unbounded connected events are continuously gathered. We can mine valuable information from these real-world streaming events by constructing them as a streaming graph. We can collect statistics from a streaming graph (e.g. how many friends does user Alice have in a social network?). We can also detect if these streaming events are connected in a specific pattern (e.g. Do user Alice and Bob know each other directly or indirectly through no more than four in-between friends?).

1.1 Motivation

Graph theory has been proved to be an effective solution for mining the latent information among big data, representative examples of which include fraud detection in transaction network, social network mining and pagerank algorithm for search engine. And with the emergence of the era of big data comes new challenges: on the one hand, efficiency of processing a huge graph on a single machine reduces significantly as the size of graph grows; on the other hand, the ability to analyze a graph before it is completely loaded is increasingly valued. However, only a few graph libraries can process streaming graph in a distributed environment, and some of them are even restricted to handle specific types of graph (see Section 2.2). Consequently, a generic streaming graph library will have tremendous value for both academia and industry.

Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. Flink has been designed to run in all common cluster environments, perform computations at in-memory speed and at any scale [11]. Different from Apache Spark, Flink does not use micro-batch to simulate stream processing, which enables true real-time processing. Therefore, Flink is a good choice for building real-time processing system. As Flink only provides a Gelly API for batch graph processing (see Section 2.3), a streaming graph library will also be a good contribution to Flink’s community.

1.2 Goal

The goal of this project is to design and implement a tool to support customized queries against large-scale streaming graph. Currently, many existing tools are specifically designed for particu-
CHAPTER 1. INTRODUCTION

lar graph or algorithms (see Chapter 2). However, it is not practical in production environment because we can not easily aggregate different systems together. Therefore, a generic streaming graph library will bring great value to the community. We first give a clear statement about the input, output and criteria of this library.

**Input** Since the library is built on top of Apache Flink, which is an event-driven stream processing platform, the library mainly accepts two types of input events: graph events and query events. Based on the definition of streaming property graph model (see Section 3.1), each graph event is a labeled edge with a unique ID. Each edge event includes a source and target vertex, which also has a unique ID and label. All edges and vertices can have different properties. As for the query event, it can be in any format such as *String, Json* or *XML*. These query events should offer enough information for formulating a customized query. Users can then parse these query events by implementing their own *parse* function (see Section 3.2). Let us consider social network as an example, sending messages from users to users can be represented as edge events, so is building friendship between users also belongs to edge events. If we want to know how many friends a user has or if there is a friend-path between two users, we can forward query events including information such as user ID and query type to a Flink job created by this library.

**Output** The output of the library is also a datastream. Graph events are simply stored in Flink’s distributed environment and do not forward any event to the output stream. Each query event strictly corresponds to a result in the output stream.

**Criteria** The library should meet the following criteria.

- Efficient memory footprint. In graph theory, a cut is to divide the graph into two subsets [16]. Partitioning a graph into several subgraphs will lead to redundancy whether we cut the graph by edge or vertex (see Section 2.1). We should reduce such redundancy because they can overwhelm the system and even cause synchronization problems between copies on different partitions.

- Fast execution speed. The library should reduce the time usage for completing a query event as much as possible. The fast execution speed is a key performance indicator.

- Ease of use. The library should provide an ease of use API or other well-defined query model (see Section 4.2).

**Project goal:** Developing a generic streaming graph library, which can perform user-defined queries within a Flink job.

1.3 Contributions

The thesis is done during author’s internship in ING, which is a Dutch multinational banking and financial services corporation headquartered in Amsterdam, Netherlands. The contributions mainly include following two parts:

**Academic contribution** To solve ING’s real-world issues, we did a quick review of today’s popular graph libraries. However, these libraries are insufficient for solving our problems. Therefore, we designed two query abstractions for our library to process graph-related queries (see Section 4.2). Additionally, we also analyzed the impact of different graph partition strategies. Ideally, if we partition the streaming graph properly, all necessary information for a specific query should be stored on the same partition. Then no communication is needed between these partitions, which is the ideal way to execute a particular query. However, there is no such partition strategy that works perfectly with all graph-related queries. By benchmarking the library with different graph partition strategies (see Section 5.1), we made two conclusions about how to choose a particular strategy for a specific query and a general strategy for all queries.
CHAPTER 1. INTRODUCTION

Engineering contribution The engineering contribution is a generic streaming graph library on top of Apache Flink. An ING-specific version was first implemented, which includes 6 graph-related queries for private use in ING. We also benchmarked these 6 different queries to show their performance under different configurations. In the end, we further generalized these 6 queries and expose a Cypher-like API to the users.

We also tried different ways to generalize the query computation model (see Section 4.2) and implemented a replication mechanism to further accelerate a particular type of queries. In the end, most queries can be completed within hundreds of milliseconds, and some complex queries can be done within a few seconds.

1.4 Thesis organization

The rest of the thesis is organized as follows.

In Chapter 2, we review the current researches on graph processing. Section 2.1 shows how to process a graph in a distributed environment. The main difficulty is how to partition a graph. We need to choose among vertex-cut, edge-cut or hybrid-cut and use different partition strategies such as hash-based strategy, random partition strategy, and round-robin partition strategy. Besides, we also discuss the graph computation model for distributed graphs, such as signal collect model. Section 2.2 introduces two mainstream methods for processing streaming graphs as well as their representative library. We investigate the difficulties in streaming graph processing and the possible solutions. In Section 2.3, we first introduce Flink’s Gelly library, which enables graph processing with three different graph computation models, and why these models do not work in a streaming environment. In Section 2.4, we introduce Neo4j, which is an open-source NoSQL native graph database that provides an ACID-compliant transactional backend [9]. Neo4j also defines a clear property graph model and provides an SQL-inspired query language called Cypher. In fact, Neo4j provides a good inspiration for implementing our own library. In Section 2.5, we make a comparative study of the reviewed methods.

In Chapter 3, we present the foundations of our library. Section 3.1 provides the definition of our streaming property graph model. The model specifies the details of our library’s graph input. In Section 3.2, we introduce the two-layer abstraction of our query computation model. The first layer is parameter-based and gives users more flexibility. The second layer is an experimental Cypher-like API. In Section 3.3, we discuss the design of iterative scatter-gather model. The model is the basis of the two query abstractions discussed in previous section. In Section 3.4, we compare the differences between three graph partition strategies and explain why they are included in our library.

Chapter 4 is the core of this thesis. We first introduce the restrictions within the design of this library in Section 4.1. Subject to these restrictions, we explain the key design decisions of this library in Section 4.2. These decisions include how to support different partition strategies, how to cut and store the streaming graph, and how to implement the query computation models, redundancy mechanism and Cypher-like API. All these decisions make the library generic and efficient. In the last Section 4.3, we show our Flink execution plan, which indicates how we organize the library. The organization determines how the event stream flows in a Flink job.

In Chapter 5, we first introduce a large-scale graph dataset in Section 5.1. The dataset is used for benchmarking this library. In Section 5.2, we describe the design and system specification of all experiments. In Section 5.3, we discuss the benchmark results from four aspects. The first step is to discuss the general performance of this library. The general performance is evaluated by throughput, load balancing and possible cross-partition communication cost. Besides the general performance, we also test the time usage of different queries under different partition strategies, graph sizes and query models. In Section 5.4, we conclude the benchmark results. According to these benchmark result, we make an exploration on how to execute a graph query efficiently with our library.

In Chapter 6, we conclude by providing an overview of the main contributions of this thesis and also possible future works.
Chapter 2

Related work

A graph is a mathematical object that captures the notion of connection [20]. Connections are everywhere in real-life scenarios, but these different application scenarios can be represented by different types of graph. A connected graph means all vertices in the graph are connected directly or indirectly. A bipartite graph can be split into two sets of vertices such that edges only go between sets, not within them. Multi-graph is a graph in which there are multiple edges between any pair of vertices or there are edges from a vertex to itself, which is also called a loop [6]. With different types of graph representing different real-world problems, some specially optimized graph algorithms or solutions are always necessary to improve the performance. Moreover, the fast-growing graph size leads to the need for distributed processing, and the concept of streaming graph brings real-time characteristic to graph processing.

2.1 Distributed graph processing

Distributed graph processing is proposed due to the limitation of scalability. It is important to clarify that there is clear-cut distinction between distributed graph processing and parallel graph processing. In parallel graph processing, the computation is distributed among the system, the parallel jobs may communicate with each other through shared memory. But in distributed graph processing, the work is done with geographically distributed systems [30]. So the application of parallel graph algorithms is more or less the same as that of the sequential algorithms. But when the nodes of a graph are geographically distributed, it is impossible to compute structures like shortest path since no node has global topology, and the topology is ever-changing with the presence of node and link failures [30]. Therefore, it is important to avoid cross-partition communication which will lead to expensive network overhead.

Another difficulty in distributed graph processing is the imbalanced workload. A graph consists of edges and vertices. Figure 2.1 shows the two mainstream methods to distribute a graph. Many distributed graph systems partition the graph by cutting edges. They assign vertices to machines uniquely, so edges will exist in different computation nodes. In this case, the overhead mainly lies on the communication of cut nodes, and is positively correlated to the number of edges that are cut. So we can minimize the number of cut edges to reduce the overhead [18]. However, it is difficult to get a balanced edge-cut for most real-world graphs. These graphs are also called natural graphs, which usually follow power-law distribution. Power-law natural graphs usually have a small number of high-degree vertices and a large number of low-degree vertices [27].

Figure 2.2 shows that most of the vertices in a natural graph are low-degree vertices. They only have a few in degrees or out degrees. Only a few vertices are high-degree vertices, which are frequently connected in the graph. Edge-cut approaches can evenly assign the vertices to machines, and maintain a consistent local state by replicating vertices and edges. However, for skewed graphs, edge-cut suffers from imbalanced computation and communication caused by high-degree vertices [27].
CHAPTER 2. RELATED WORK

Figure 2.1: **Edge-cut vs Vertex-cut**: An edge-cut (a) splits the graph along edges while a vertex-cut (b) splits the graph along vertices. In this illustration we partition the graph across three machines (corresponding to color), taken from [39].

![Edge-cut vs Vertex-cut illustration](image)

Figure 2.2: The power-law distribution of the vertex degree in the real-world dataset in log-log scale, taken from [27].

![Power-law distribution](image)

Vertex-cut methods partition a graph by cutting vertices and evenly assigning edges to machines, and distribute the edges of high-degree vertices to multiple machines to balance the load. Vertex-cut suffers from high replication factor and high communication cost among replicated low-degree vertices [27]. Although a number of improvements for vertex-cut have been proposed, these variations of vertex-cut essentially distribute the edges of low-degree vertices to multiple machines, and thus either have significant overhead for graph placement with high ingress time, or have high cost of communication (between low-degree vertices and their mirrors) for graph computation with excessive execution time [19], [22].

Besides vertex-cut and edge-cut approaches, a hybrid-cut approach is also proposed for solving the issues we have mentioned above. The approach distinguishes the processing of low-degree and high-degree vertices. It evenly distributes the low-degree vertices, as well as the edges of high-degree vertices among machines. However, the hybrid design has the following problems [27].

- It is difficult to decide the differentiation threshold, which may lead to suboptimal choice for the considerable quantity of moderate-degree vertices.

- Hybrid-cut counts the degree of every vertex to decide whether it is high-degree or low-degree. For large-scale disordered datasets, this substantially increases the communication cost for graph loading since the entire graph has to be traversed to count the degrees of vertices.

- Differentiated computation and partitioning remarkably increase the complexity of the design and implementation [27].

Table 2.1 summarizes the differences between these three approaches.

With a specific cut approach, it is also important to propose a good partition strategy. Graph partitioning has a rich history. It encompasses many problems and has many proposed solutions,
CHAPTER 2. RELATED WORK

Table 2.1: Comparison between different graph cut approaches

<table>
<thead>
<tr>
<th>Approach</th>
<th>Edge-cut</th>
<th>Vertex-cut</th>
<th>Hybrid-cut</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cut edges</td>
<td>Cut high-degree vertices</td>
<td>Cut low-degree vertices and edges of high-degree vertices</td>
</tr>
<tr>
<td>Weakness</td>
<td>Load imbalance</td>
<td>High communication cost</td>
<td>Difficulties to decide</td>
</tr>
<tr>
<td></td>
<td>Communication caused by high-degree vertices</td>
<td>between (partitioned) low-degree vertices</td>
<td>the most appropriate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>threshold and count</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>the degrees of</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>vertices beforehand</td>
</tr>
<tr>
<td>Libraries</td>
<td>Pregel[31], GraphLab[28]</td>
<td>PowerGraph[21], GraphX[39]</td>
<td>PowerLyra[19]</td>
</tr>
</tbody>
</table>

from the very simple to the very sophisticated [35]. The goal is, given a graph $G$ as input and a number $k$, to cut $G$ into $k$ balanced pieces while minimizing the number of edges or vertices cut. To achieve the goal, multiple complex partitioning algorithms have been designed including finding balanced cuts, $k$-way partitions, and hierarchical clusterings [26], [34], [33], [23], [17]. However, it is still difficult to figure out a perfect graph partitioning algorithms, and different algorithms are usually designed for particular graph problems.

2.2 Streaming graph processing

Different from distributed graph processing, the overall understanding of the graph is insufficient in streaming graph processing. We can consider a simple streaming graph model mentioned in Isabelle Stanton and Gabriel Kliot’s work [35]. We have a cluster of $k$ machines, each with memory capacity $C$, such that the total capacity, $kC$, is large enough to hold the whole graph. The graph is $G = (V,E)$ where $V$ is the vertices, and $E$ the edges. The graph may be either directed or undirected. The vertices or edges arrive in a stream. In this situation, we need to design a partitioner that can decide which machine the incoming vertex or edge should be distributed to. Theoretically, it is impossible to come up with a perfect streaming partitioning algorithm for all graph algorithms. It is easy to create graphs and orderings for any algorithm that will cause it to perform poorly [35]. A simple example is a bank transaction network. Each account of bank customers can be seen as a vertex. The edges are the transactions between customer accounts. If we want to calculate the amount of money transferred between any two customers, we can sum the ID of transferrer and receiver and use a modulo function to decide which machine the transaction should be partitioned to. However, if we want to calculate the amount of money that one customer has received in the past month, the transactions may appear in any machine based on the previous partition strategy.

Heuristic approaches

It is worth noting that the state-of-the-art work on graph partitioning seems to roughly divide in two main lines of research: rigorous approximation algorithms mathematically work but do not scale to massive graphs, and heuristics algorithms are used in practice [36]. In our work, we are more interested in the practical work as our major goal is to develop a library for solving real-world problems. Based on Isabelle Stanton and Gabriel Kliot’s summarization of some existing heuristics [35], we can group them into three classes. $ind$ represents the index of selected partition.

In the following equations, notation $P^t$ refers to the set of partitions at time $t$. Each individual partition is referred to by its index $P^t(i)$ so $\bigcup_{i=1}^k P^t(i)$ is equal to all of the vertices placed so far. Let $v$ denote the vertex that arrives at time $t$ in the stream, $\Gamma(v)$ refers to the set of vertices that $v$ neighbors and $|S|$ refers to the number of elements in a set $S$. $C$ is the capacity constraint on each partition. Each of the heuristics gives an algorithm for selecting the index $ind$ of the partition.
CHAPTER 2. RELATED WORK

where \( \nu \) is assigned.

The first kind of strategies mainly aims at load balancing.

1. **Balancing.** Assign \( \nu \) to a partition of minimal size, breaking ties randomly.

\[
\text{ind} = \arg \min_{i \in [k]} \{|P^t(i)|\}
\]

2. **Chunking.** Divide the stream into chunks of a specific size and fill the partitions completely in order. It can also be seen as a round-robin strategy.

\[
\text{ind} = \lceil t/C \rceil
\]

The second kind of strategies focuses more on minimizing the number of vertex-cuts.

3. **Hashing.** Given a hash function \( H : V \rightarrow \{1 \cdots k\} \), assign vertex \( \nu \) to \( \text{ind} = H(\nu) \). We use:

\[
H(\nu) = (\nu \mod k) + 1
\]

4. **(Weighted) Deterministic Greedy.** Assign \( \nu \) to the partition where it has the most edges. Weight this by a penalty function based on the capacity of the partition, penalizing larger partitions. Break ties using Balanced.

\[
\text{ind} = \arg \max_{i \in [k]} \{|P^t(i) \cap \Gamma(\nu)|w(t, i)|\},
\]

where \( w(t, i) \) is a weighted penalty function:

- \( w(t, i) = 1 \) for unweighted greedy
- \( w(t, i) = 1 - \frac{P^t(i)}{C} \) for linear weighted
- \( w(t, i) = 1 - \exp\{|P^t(i)| - C\} \) for exponentially weighted

5. **(Weighted) Randomized Greedy.** Assign \( \nu \) according to the distribution defined by

\[
Pr(i) = |P^t(i) \cap \Gamma(\nu)|w(t, i)/Z,
\]

where \( Z \) is the normalizing constant and \( w(t; i) \) is the above three penalty functions.

6. **(Weighted) Triangles.** Assign \( \nu \) according to

\[
\arg \max_{i \in [k]} \frac{|E(P^t(\nu) \cap \Gamma^t(\nu), P^t(\nu) \cap \Gamma(\nu))|w(t, i)|}{(|P^t(\nu) \cap \Gamma(\nu)|)^2},
\]

where \( w(t, i) \) is the above 3 penalty functions and \( E(S, T) \) is the set of edges between the nodes in \( S \) and \( T \).

The third type of strategies differentiates high-degree and low-degree vertices.

7. **Balance Big.** Given a way of differentiating high and low degree nodes, if \( \nu \) is high-degree, use Balanced. If it is low-degree, use Deterministic Greedy.

8. **Prefer Big.** Maintain a buffer of size \( C \). Assign all high degree nodes with Balanced, and then stream in more nodes. If the buffer is entirely low degree nodes, then use Deterministic Greedy to clear the buffer.
CHAPTER 2. RELATED WORK

9. **Avoid Big.** Maintain a buffer of size \( C \) and a threshold on large nodes. Greedily assign all small nodes in the buffer. When the buffer is entirely large nodes, use **Deterministic Greedy** to clear the buffer.

10. **Greedy EvoCut.** Use EvoCut [14] on the buffer to find small Nibbles with good conductance. Select a partition for each Nibble using **Deterministic Greedy**.

Each of the above heuristics has a different motivation with some arguably more natural than others. Among all these heuristics, the third strategy **hashing** is most widely used. The benefit of this strategy is that every vertex or edge can be quickly found, without the need to maintain a distributed mapping table. Moreover, the greedy strategy is also useful in many cases. Because we can make a fast decision just based on all information that is already collected. The fast response time determines the performance of a real-time processing.

The interesting heuristic is called FENNEL [36]. In a distributed system, by assigning one partition to one worker node, we get a direct implementation of FENNEL processing model. For every newly arrived vertex \( v \), a proxy node will broadcast the vertex data, including its neighbor list to all \( K \) worker nodes. The workers will cache that data firstly, and use a greedy vertex assignment algorithm to calculate the gradient \( \delta_g(v, S_i) = |N(v) \cap S| - \alpha(|S| + 1)^\gamma - |S|^\gamma) \), which gives the outcome if vertex \( v \) is allocated to this worker (partition), then return the value to proxy. After proxy has gathered all the returned values \( \delta_g(v, S) \), it will choose \( \delta_{\text{max}}g(v, S_i) \), and broadcast the decided optimal partition \( i \) back to \( K \) workers. Then, for every worker, it will check whether it holds the optimal partition \( i \) or not. If yes, the worker takes corresponding vertex data from cache and puts in local storage. Otherwise, the worker removes corresponding vertex data from cache and puts a key-value pair \( \langle v, i \rangle \) into local table for future reference [32].

However, the FENNEL model has two major problems. On one hand, synchronous processing will lead to low network efficiency. On the other hand, the star-shaped network topology has a limitation for scalability. Therefore, an asynchronous version of FENNEL was introduced in [32]. Figure 2.3 shows the asynchronous graph processing model for FENNEL.

![Figure 2.3: Asynchronous vertex data processing model, taken from [32]](image)

In this model, a proxy will send vertices with their adjacency lists to all workers continuously, without waiting for the return of their gradients. Considering a concurrency of \( N \), on the worker side, the gradients of no more than \( N \) vertices can be calculated and sent back to the proxy simultaneously.

**Two adaptive approaches**

To solve the load balancing problem in streaming graph processing, there are another two interesting adaptive approaches.

The first approach evolves from the hashing approach. Suppose an incoming graph stream consists of massive graph edges, each edge has a source vertex and target vertex which can be identified by a unique id. If we partition each edge based on source or target vertex id, all edges related to one specific vertex will go to the same partition, which can reduce the possible
communication cost between partitions. However, a natural graph has a few high-degree vertices which own most of the edges in the graph. In such situation, those edges connected with high-degree vertices will be distributed to the same partition. And there is no guarantee that these high-degree vertices will be evenly distributed. Because the vertex IDs are usually randomly generated. Apart from partitioning based on only source or target vertex id, we can also partition the edges based on the sum of the source and target id. However, the edges connected with a specific vertex may be distributed to any partition, which will lead to massive cross-partition communication. Figure 2.4 shows how to use multiple incremental hash functions to evenly distribute the high-degree vertices.

Figure 2.4: Adaptive Partitioning with incremental hash functions: $\eta = 3$ (number of hash functions), taken from [41]

In this approach, we need to maintain a global partition table. The table will record the hash function being used for each vertex. If the partition computed by current hash function has surpassed the maximal capacity for storing this vertex, the vertex will use next hash function and update the partition table. With this approach, the high-degree vertices will be evenly distributed among partitions. The low-degree vertices will only appear on a few partitions. The benefit of this approach is to get a balanced workload and bring an acceptable amount of communication cost. There is always a trade-off between load balancing and communication cost. The selection of partition strategy depends on the particular graph structure and specific graph algorithms.

Till now, we have discussed many approaches motivated by different application scenarios. An assumption for these approaches is that the distributed graph is immutable. The assigned edges and vertices will never move after being located. However, these approaches still face some challenges.

- The graph structure changes are not predictable,
- The computational overhead from the partitioning optimization must be low,
- Synchronizing the distributed state at a large scale is very costly in the dynamic environment.

Therefore, a repartitioning strategy is proposed in [38]. The goal of this approach is to prevent performance degradation as the graph size grows rapidly. The repartition approach is based on label propagation. After assigning each vertex to a partition, we need to check if any vertex can be migrated to another partition where it has maximum neighbors. The partition strategy is called greedy vertex migration. On every iteration $t^1$ after the initial partitioning, each vertex will make
a decision to either remain in the current partition, or to migrate to a different one. The candidate partitions for each vertex are those where the highest number of its neighbours are located. Since migrating a vertex potentially introduces an overhead, the heuristic will preferentially choose to stay in the current partition if it is one of the candidates. At the end of the iteration, all vertices who decided to migrate will change to their desired partitions. The heuristic relies on local information, as each vertex $v$ chooses its destination based only on the location of its neighbours. Dynamism comes natively in this iterative approach. New vertices are initially assigned a partition according to a strategy (we opted for the de facto standard, hash modulo) and the heuristic will automatically attempt to move them closer to their neighbours. Figure 2.5 shows the detail of this strategy.

Figure 2.5: Deferred Vertex Migration to Ensure Message Delivery. Top: Failed message delivery due to incorrect synchronization. Bottom: Correct delivery. The dashed-red circle indicates when the vertex is in a “migrating” state waiting for one iteration (step) before actually migrating, taken from [38]

The solution will read a stable convergence after some iterations. The major advantage of this approach is to enable dynamic graph structure change. However, the iterative graph changes will significantly increase the complexity and slow down the speed of computation.

**GraphStream vs GraphWindow**

The heuristics mentioned before are mainly for processing graph streams. However, it is also possible to process the streaming graph as graph windows. The streaming graph is sliced into multiple pieces of graph, then distributed graph algorithms can be adopted on these windows.

GraphStream has the following features:

- A representation of a data stream of edges,
- Edges can have a state (e.g. weights),
- Supports property streams, transformations and aggregations.
GraphWindow has following features:

- A “time-slice” of a graph stream.
- It enables neighborhood aggregations.

The GraphWindow solution is actually a special case of GraphStream, which processes a streaming graph in batches. However, the latency highly depends on the batch size. The performance is not acceptable in many real applications and the limited support for neighborhood aggregations is also a major drawback of GraphWindow.

2.3 Graph processing with Flink

Graph processing will get much more complicated when the graph is geographically distributed across multiple machines. Synchronization, communication, message delivery guarantee are all necessary in such a distributed system. If we further consider the graph as a stream, the system will be even more complex. Therefore, a better choice to design such a system is taking advantage of some existing tools. There are many popular open-source platforms aiming at such a task. Apache Spark is a lightning-fast unified analytics engine, which has already integrated GraphX as its parallel-graph computation API [12]. However, Spark is mainly designed for batch processing. Spark’s streaming processing mechanism is supported by simulating micro-batches as data streams. The idea has an excellent output in many cases, but it is not enough when there is a high demand for real-time performance. Apache Storm is a free and open source distributed real-time computation system. Storm makes it easy to reliably process unbounded streams of data, doing for real-time processing what Hadoop did for batch processing. Storm is fast: a benchmark clocked it at over a million tuples processed per second per node. It is scalable, fault-tolerant, guarantees your data will be processed, and is easy to set up and operate [13]. Currently, there is still no API supported for graph processing in Apache Storm. Apart from Apache Storm, there is a better solution currently. Apache Flink is an open source platform for distributed stream and batch data processing. Flink’s core is a streaming dataflow engine that provides data distribution, communication, and fault tolerance for distributed computations over data streams. Flink builds batch processing on top of the streaming engine, overlaying native iteration support, managed memory, and program optimization [1]. Figure 2.6 shows the stack of Apache Flink.

Figure 2.6: As a software stack, Flink is a layered system. The different layers of the stack build on top of each other and raise the abstraction level of the program representations they accept: The runtime layer receives a program in the form of a JobGraph, both the DataStream API and the DataSet API generate JobGraphs through separate compilation processes, taken from [38].

In such streaming engines, the design of fault tolerance mechanism is of great importance, not only because of the need for fault-tolerant processing, but also because of the additional system
overhead caused by fault-tolerant processing. Apache Flink offers a fault tolerance mechanism to consistently recover the state of data streaming applications. The mechanism ensures that even in the presence of failures, the programs state will eventually reflect every record from the data stream exactly once [2]. The fault tolerance mechanism continuously draws snapshots of the distributed streaming data flow. For streaming applications with small state, these snapshots are very light-weight and can be drawn frequently without much impact on performance.

Gelly and its computation models

Gelly is a Graph API for Flink. It contains a set of methods and utilities which aim to simplify the development of graph analysis applications in Flink. In Gelly, graphs can be transformed and modified using high-level functions similar to the ones provided by the batch processing API. Gelly provides methods to create, transform and modify graphs, as well as a library of graph algorithms [3].

Gelly already supports a wide arrange of graph algorithms, including community detection, label propagation, connected components, single source shortest path, triangle enumerator, summarization, clustering, pagerank, etc [4].

Gelly exploits Flink’s efficient iteration operators to support large-scale iterative graph processing. Currently, Gelly supports three computation models, including vertex-centric, scatter-gather, and gather-sum-apply models. The basic idea of these three models is similar to each other. They all treat a graph vertex or edge as a computation unit. In vertex-centric model, we need to define our own vertex compute function, which will be executed by all vertices in each iteration. In the scatter-gather model, the difference is that we need to define scatter and gather function separately. The process of sending messages to and receiving messages from connected vertices are distinguished. This model may be a better choice for some use cases. The last GSA model also proceeds in synchronized iterative steps, called supersteps. Each superstep consists of the following three phases:

- **Gather**: a user-defined function is invoked in parallel on the edges and neighbors of each vertex, producing a partial value.
- **Sum**: the partial values produced in the Gather phase are aggregated to a single value, using a user-defined reducer.
- **Apply**: each vertex value is updated by applying a function to the current value and the aggregated value produced by the Sum phase.

Let us consider computing Single-Source-Shortest-Paths on the following graph in Figure 2.7 and let vertex 1 be the source. Figures 2.7, 2.8 and 2.9 illustrate how to do it with each computation model.

Actually, the three iteration abstractions in Gelly are quite similar. The user needs to decide which model is a better choice for their particular use case. Table 2.2 shows the difference between these three models.

<table>
<thead>
<tr>
<th>Iteration Model</th>
<th>Update Function</th>
<th>Update Logic</th>
<th>Communication Scope</th>
<th>Communication Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex-Centric</td>
<td>arbitrary</td>
<td>arbitrary</td>
<td>any vertex</td>
<td>arbitrary</td>
</tr>
<tr>
<td>Scatter-Gather</td>
<td>arbitrary</td>
<td>based on</td>
<td>any vertex</td>
<td>based on vertex state</td>
</tr>
<tr>
<td>Gather-Sum-Apply</td>
<td>associative &amp; commutative</td>
<td>based on neighbors’ values</td>
<td>neighborhood</td>
<td>based on vertex state</td>
</tr>
</tbody>
</table>
CHAPTER 2. RELATED WORK

Figure 2.7: **Vertex-centric model**: Initially, each vertex has a value of infinite distance, except from the source vertex, which has a value of zero. During the first superstep, the source propagates distances to its neighbors. During the following supersteps, each vertex checks its received messages and chooses the minimum distance among them. If this distance is smaller than its current value, it updates its state and produces messages for its neighbors. If a vertex does not change its value during a superstep, then it does not produce any messages for its neighbors for the next superstep. The algorithm converges when there are no value updates or the maximum number of supersteps has been reached. The figure is taken from [8].

Figure 2.8: **Scatter-gather model**: In each superstep, each vertex sends a candidate distance message to all its neighbors. The message value is the sum of the current value of the vertex and the edge weight connecting this vertex with its neighbor. Upon receiving candidate distance messages, each vertex calculates the minimum distance and, if a shorter path has been discovered, it updates its value. If a vertex does not change its value during a superstep, then it does not produce messages for its neighbors for the next superstep. The algorithm converges when there are no value updates. The figure is taken from [8].

**An experimental streaming graph API**

Since Gelly is implemented on top of Flink’s batch processing API, we have to load the whole graph before executing any graph algorithms. As a result, Gelly is not for processing streaming graphs. Table 2.3 shows the differences between batch processing and stream processing. We have to design another library on top of Flink’s stream processing API. Currently, Vasia Kalavri has designed an experimental streaming graph API for Single-Pass Graph Streaming Analytics [5]. Single pass algorithm is a streaming algorithm which reads its input exactly once, in order, without unbounded buffering. Vasia’s experimental library has implemented such algorithms like Connected Components, k-Spanner, Continuous Degree Aggregate, Weighted Matching, etc.
CHAPTER 2. RELATED WORK

Figure 2.9: Gather-sum-apply model: During the Gather phase, we calculate the new candidate distances, by adding each vertex value with the edge weight. In Sum, the candidate distances are grouped by vertex ID and the minimum distance is chosen. In Apply, the newly calculated distance is compared to the current vertex value and the minimum of the two is assigned as the new value of the vertex. Notice that, if a vertex does not change its value during a superstep, it will not calculate candidate distance during the next superstep. The algorithm converges when no vertex changes the value. The figure is taken from [8]

Table 2.3: Comparison between batch and stream graph processing, taken from [5]

<table>
<thead>
<tr>
<th>Batch processing</th>
<th>Stream processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Graphs</td>
<td>Dynamic Graphs</td>
</tr>
<tr>
<td>Multi-Pass Algorithms</td>
<td>Single-Pass Algorithms</td>
</tr>
<tr>
<td>Full Computations</td>
<td>Approximate Computations</td>
</tr>
</tbody>
</table>

2.4 Graph processing with Neo4j

In Section 2.3, we introduced three iterative graph processing models. These models are usually used when the graph algorithms can be easily abstracted into iterative steps. After the iterative calculation converges, we will automatically get the result we want. However, there is another type of graph-related queries. Such queries need not only SQL-like functionality for querying structured data, but also intrinsic support for typical graph-style applications: reachability analysis, path finding and graph construction [37]. Apparently, iterative graph processing cannot handle these queries. Therefore, some SQL-like graph query languages are naturally proposed. Among all these designs, we find that Neo4j is a powerful tool which implements a native graph database and an ease-to-use graph query language.

Neo4j property graph model

Neo4j defines a labeled property graph model. In this model, nodes are the entities in the graph. They can hold any number of attributes (key-value-pairs) called properties. Nodes can be tagged with labels representing their different roles in your domain. In addition to contextualizing node and relationship properties, labels may also serve to attach metadata-index or constraint information to certain nodes.

Relationships provide directed, named, semantically relevant connections between two node-entities (e.g. Employee WORKS_FOR Company). A relationship always has a direction, a type, a start node, and an end node. Like nodes, relationships can also have properties. In most cases, relationships have quantitative properties, such as weights, costs, distances, ratings, time intervals, or strengths. As relationships are stored efficiently, two nodes can share any number or type of relationships without sacrificing performance. Note that although they are directed, relationships can always be navigated efficiently in either direction. Figure 2.10 shows a typical relationship and its related nodes.
CHAPTER 2. RELATED WORK

Figure 2.10: A typical block in Neo4j. The green node is an employee who works for the pink node, which is a company. The company is located in a city represented by the blue node, taken from [9]

Basically, Neo4j is a native database which stores graph data in a number of different store files. Neo4j does not support database sharding. The support of clustering is mainly for high availability. The organization of graph storage is shown below.

- Properties are stored as a linked list of property records, each holding key+value.
- Each node/relationship references its first property record.
- The nodes also reference the first node in its relationship chain.
- Each relationship references its start and end node.
- Each relationship also references the prev/next relationship record for the start/end node respectively.

Cypher query language

Cypher is a declarative query language similar to SQL, but optimized for graphs [9]. Cypher supports not only basic syntax like match, update and delete, but also enables user-defined procedures and functions to extend Cypher. Here is a simple example of Cypher query.

Listing 2.1: A simple example of the Cypher query. The query will find out all accounts that used to transfer money to any account that has transferred money to Alice.

```
match (n) -[:hasAccount]->(a:Account) -[:transferTo]->(b:Account)<-[[:hasAccount]}-(x:Customer)
where x.Name='Alice'
return n
```

2.5 Comparative study

As a conclusion of this chapter, we will compare the features and performance of several popular graph processing engines. The development of graph processing system starts from native graph processing. Most graph algorithms are sequential algorithms. Soon researchers try to design some parallel graph algorithms to accelerate the execution. Neo4j is such a graph database which enables fast graph queries by parallel processing. However, the fast-growing graph size makes it harder and harder to scale vertically. Therefore, horizontal scaling becomes a better and feasible solution. By scaling horizontally, graph data is geographically distributed on separate partitions. As these separated partitions can only communicate with each other over expensive networking, it is necessary to minimize these synchronization costs. Considering different real-world use cases, researchers have proposed different approaches to cut a graph. In this section, we will mainly compare three vertex cut approaches, three edge-cut approaches and one hybrid-cut approach.
PowerGraph [21], GraphX [39] and GraphA [27] cut a graph by vertex.

**PowerGraph:** To address the challenges of computation on power-law graphs, Joseph E. Gonzalez, Yucheng Low and Haitie Gu proposed a new graph-parallel abstraction that eliminates the degree dependence of the vertex-program by directly exploiting the GAS decomposition to factor vertex-programs over edges. By lifting the Gather and Scatter phases into the abstraction, PowerGraph is able to retain the natural think-like-a-vertex philosophy [31] while distributing the computation of a single vertex-program over the entire cluster. PowerGraph combines the best features from both Pregel and GraphLab.

**GraphX:** Existing data-parallel frameworks like Pregel and GraphLab share many common properties, but it is difficult to compose these abstractions because each framework relies on a separate runtime. Therefore, if we can make use of some existing data-parallel systems like MapReduce and Spark [40], features such as high scalability and fault-tolerance can be easily supported based on common runtime. However, naively expressing graph computation and graph algorithms in these data-parallel abstractions can be challenging and typically leads to complex joins and excessive data movement that does not exploit the graph structure. GraphX is a graph computation system which runs in the Spark data parallel framework. By extending Spark’s Resilient Distributed Dataset (RDD) abstraction to introduce the Resilient Distributed Graph (RDG), we can associates records with vertices and edges in a graph and provides a collection of expressive computational primitives.

**GraphA:** To efficiently store the fast-growing graph and satisfy the performance requirements of on-machine storage operations like creation and deletion, Dongsheng Li, Chengfei Zhang, etc designed an adaptive approach to efficient partitioning, storage and computation for large-scale natural graphs. GraphA provides a uniform partitioning algorithm that partitions the datasets in a load-balanced manner. GraphA adopts vertex-cut to evenly distribute edges to machines by using an incremental number of hash functions on the vertices. The functions distribute edges of the same (low-degree) vertex to one machine, while constraining the number of edges stored in each machine to the machines limitation. The incremental hash algorithm of GraphA minimizes communication cost for low-degree vertices and achieves load balancing for high-degree vertices. At the same time, it does not complex differentiation like hybrid-cut. GraphA is implemented as a separate engine on top of Apache Spark.

Pregel [31], GraphLab [28] and FENNEL [36] are three models using edge-cut approach.

**Pregel:** Graph algorithms often exhibit poor locality of memory access, very little work per vertex, and a varying degree of parallelism during execution [29]. Distribution over many machines exacerbates the locality issue, and increases the probability that a machine will fail during computation. To address these issues, Grzegorz Malewicz, Matthew H. Austern, etc built a scalable and fault-tolerant platform with an API that is sufficiently flexible to express arbitrary graph algorithms. A typical Pregel computation begins with input when the graph is initialized, followed by a sequence of supersteps separated by global synchronization points until the algorithm terminates, and finishes with output. Within each superstep the vertices compute in parallel, each executing the same user-defined function that expresses the logic of a given algorithm.

**GraphLab:** GraphLab is originally a parallel framework for machine learning which exploits the sparse structure and common computational patterns of machine learning algorithms, because GraphLab supports asynchronous iterative computation, it can also be used as a graph processing framework. GraphLab implements a graph-based data model which simultaneously represents data and computational dependencies. GraphLab also includes a set of concurrent access models which provide a range of sequential-consistency guarantees. In addition, a sophisticated modular scheduling mechanism and an aggregation framework to manage global state are also included in GraphLab.

**FENNEL:** It is always challenging to find a balanced graph partition while processing a large-scale graph. Furthermore, dynamic graphs in big data era make it even more difficult to find an efficient partition strategy. For example, a company account needs to transfer salaries to its employees on a fixed date every month. Without a proper partition strategy, such intensive transactions related with one account may lead to imbalanced partitions. FENNEL is a framework for graph partitioning that resolves the computational complexity of the traditional balanced
CHAPTER 2. RELATED WORK

graph partitioning problem. Specifically, in the traditional balanced graph partitioning problem, the goal is to minimize the number of edges cut subject to hard constraints on the number of vertices in a cluster [24]. FENNEL loosens the hard cardinality constraints by formulating the graph partitioning objective function to consist of two elements: one that accounts for the cost of edges cut and another for the cost related to the sizes of individual clusters.

PowerLyra [19] exploits both the advantages of vertex-cut and edge-cut approach.

**PowerLyra** Above vertex-cut and edge-cut approaches have invented many creative mechanisms and abstract models to improve the performance in graph processing systems. However, these solutions are usually specifically designed for a particular group of graph algorithms. To incorporate the best of existing systems, PowerLyra distinguishes the processing of low-degree and high-degree vertices: it uses centralized computation for low-degree vertices to avoid frequent communication and only distributes the computation for high-degree vertices. PowerLyra follows the GAS (Gather, Apply and Scatter) model and can seamlessly support existing graph algorithms under such a model.

<table>
<thead>
<tr>
<th>Vertex-cut approach</th>
<th>PowerGraph</th>
<th>GraphX</th>
<th>GraphA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Comp. Pattern</td>
<td>distributed</td>
<td>distributed</td>
<td>distributed</td>
</tr>
<tr>
<td>Comm. Cost</td>
<td>high</td>
<td>high</td>
<td>average</td>
</tr>
<tr>
<td>Dynamic Comp.</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Load Balance</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Comp. Model</td>
<td>vertex-centric</td>
<td>GAS</td>
<td>GAS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Edge-cut approach</th>
<th>Pregel</th>
<th>GraphLab</th>
<th>FENNEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Comp. Pattern</td>
<td>local</td>
<td>local</td>
<td>distributed</td>
</tr>
<tr>
<td>Comm. Cost</td>
<td>low</td>
<td>high</td>
<td>average</td>
</tr>
<tr>
<td>Dynamic Comp.</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Load Balance</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Comp. Model</td>
<td>vertex-centric</td>
<td>scatter-gather</td>
<td>Streaming</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hybrid-cut approach</th>
<th>PowerLyra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Comp. Pattern</td>
<td>L: local</td>
</tr>
<tr>
<td></td>
<td>H: distributed</td>
</tr>
<tr>
<td>Comm. Cost</td>
<td>average</td>
</tr>
<tr>
<td>Dynamic Comp.</td>
<td>yes</td>
</tr>
<tr>
<td>Load Balance</td>
<td>yes</td>
</tr>
<tr>
<td>Comp. Model</td>
<td>GAS</td>
</tr>
</tbody>
</table>

Table 2.4: Comparison between seven typical graph processing libraries/models
Chapter 3
Library foundations

Based on the existing works mentioned in Chapter 2, we develop a streaming graph library. The library is built on top of Apache Flink due to the reasons explained in Section 2.3. This chapter defines the foundations of the library. As shown in Figures 2.7, 2.8 and 2.9, vertices or edges can be seen as computation nodes which can send or receive messages from neighbors in each superstep. However, there is no clear superstep in streaming graph processing, because vertices and edges may be added or removed at any time and thus no way to synchronize messages between them in a specific time interval. Furthermore, the computation will never converge because the graph is constantly changing. The lack of boundary between supersteps in stream processing makes it impossible to work with existing computation models such as vertex-centric, scatter-gather or gather-sum-apply. Therefore, we divide the past computation models into two parts: graph model and query model. Section 3.1 defines the graph model which illustrates the type of graph can be processed by this library. These graph data are abstracted as Edge or Vertex events in data stream. We introduce a two-layer query abstraction model in Section 3.2. For the first layer, users can define the processing logic by implementing the abstract functions in the abstract Query class, and the second layer provides an ease-to-use Cypher-like API to define customized API. With these abstractions, a Query event can be parsed into a Query object and thus compute the result based on information available on each graph partition. To implement the graph and query model, we introduce our iterative scatter-gather model in Section 3.3, which is an improved version of the existing iterative graph processing model. The last Section 3.4 illustrates three different graph partition strategies supported in our library. Each strategy usually performs well with a specific group of queries, the performance difference between these strategies will be further discussed in Section 5.3.

3.1 Streaming property graph model

There are various types of graph such as natural graph, bipartite graph or connected graph. Each specific graph is modeled for different real-world scenarios. There is no unified solution for all graph problems and thus researchers have proposed a wide range of algorithms, theories and tools from either theoretical or practical perspective. In this thesis, a streaming graph library is designed on top of Apache Flink. The library can be used for executing queries on streaming property graph.

Before discussing streaming property graph model, let us first consider the property graph model. The idea of property graph model comes from Neo4j’s property graph model.

Graph \( G = \{V, E\} \) includes a set of vertices \( V \) and edges \( E \). A vertex \( V \) is identified by a unique ID and label. The vertex label shows the type of a vertex. For example, a vertex can be a customer, an account or even an address in a bank network. A vertex also includes a list of properties such as customer age, account open time, postcode of an address. The difference between vertex label and property is that label is immutable but properties can be added, changed or removed. Vertices are connected through edges, and an edge is identified by its source vertex,
target vertex and label. An edge explicitly indicates a path from its source vertex to its target vertex, and the label specifies the type of connection. For example, Figure 3.1 shows that Bob has a salary account and a family account. He used to transfers money from his salary account to his family account.

![Figure 3.1: An example of vertices linked with edges](image)

Figure 3.1: An example of vertices linked with edges: The green circle indicates a customer vertex, whereas the purple circle represents customer Bob’s accounts. The word on each edge specifies the edge label.

An edge also has a list of properties. Similar to the vertex property, edge property is also mutable, but edge label is not. For example, Alice opened one of her accounts five years ago. Then there is an edge that Alice -hasAccount-> account1. Then five years ago can be one of this edge’s properties, which is account open time. Later, Alice may decide to close this account, then this edge will be removed from the graph.

Figure 3.2 shows a bank network represented by our property graph model.

![Figure 3.2: A simple example of bank network](image)

Figure 3.2: A simple example of bank network: The green circles represent customers, and the blue circles are the accounts owned by these customers, the graph is generated by Neo4j [9].

Now we consider adding the streaming feature to this property graph model. In the real world, a graph is not created at once. The graph size grows by connecting vertices and edges to itself. Therefore, we just treat each vertex or edge as an event in a streaming graph. For example, an
event including a new vertex will be added to the data stream when new accounts are created in social network. If we consider a bank transaction network, a transaction between customers can be seen as the streaming edges in our streaming graph model.

Given the above, our streaming property graph model can be defined as below. The model is inspired by Angles et al’s graph data model [15].

• \(L\) is an infinite set of (node and edge) labels,
• \(P\) is an infinite set of property names,
• \(V\) is an infinite set of values,
• \(T\) is an infinite set of timestamps.

A streaming property graph model is a tuple \(G = (V, E, \rho, \lambda, \sigma, \tau)\).

• \(N\) is a finite set of vertices;
• \(E\) is a finite set of edges such that \(N\) and \(E\) have no elements in common;
• \(\rho : E \rightarrow (N \times N)\) is a total function between edges and vertices, thus every edge connects two vertices. The edges are all directed.
• \(\lambda : (N \cup E) \rightarrow P(L)\) where \(P(L)\) is the powerset of \(L\), is a total function from vertices and edges to sets of labels.
• \(\sigma : (N \cup E) \times P \rightarrow V\) is a partial function from the property names of vertices and edges to their values.
• \(\tau : (N \cup E \cup P) \rightarrow T\) is a total function from the vertices, edges and properties to the time windows. Timestamp of vertices and edges are confirmed once they are first loaded, timestamp of properties is the time when the property is stored.

### 3.2 Abstraction of query model

For most existing graph libraries or tools, the graph computation task is usually delegated to vertices or edges and from this comes three popular graph computation models discussed in Section 2.3: vertex-centric, scatter-gather and gather-sum-apply model. These models iteratively compute and synchronize the status of each computation unit until the computation converges. However, this iterative graph processing model is not feasible in streaming graph processing, because the streaming graph will never converge due to the streaming graph being dynamically changing.

Another issue of iterative processing model is that we usually do not apply global computation on streaming graph. Because computations like pagerank algorithms will cause too much traffic in data stream and thus result in congestion, our library is designed to handle graph and computation processing separately. The query model is abstracted as below.

Each query is an event in data stream. As mentioned in last section, graph is stored distributively when graph events go through the stateful operators. These query events are mixed with graph events, and they can simply collect the necessary information when they flow through the corresponding graph storage partition.

**First layer: parameter-based query model**

In our library, the graph query model includes two layers’ abstraction. The first-layer abstraction is parameter-based. Each query event contains a well-structured text such as JSON or XML. These texts can be easily parsed and provide the necessary information for formulating a query. In this way, a user can define their own queries by implementing their own Query Parse Function. Listing 3.1 shows two examples.
Listing 3.1: Two examples of parameter-based query. The first query is to calculate how much money account 1 has transferred to all other accounts between 2014 and 2019. The second query is to check if there is a transaction path from account 1 to account 4 and the length is no more than 4.

Second layer: Cypher-like query model

With the parameter-based query model, we can implement various queries with flexible parameters defined by users. However, users also need to implement the parse function for every parameter and thus decide how to use these parameters in the following computation. This will result in situations where users need to reimplement a new query model even there is a slight change. For example, if we have two queries: the first one is to calculate how much money Alice transferred to other people, and the second one is to compute how much money Alice received from other people. The only difference is the direction of the edge. It is much better if we can generalize such queries and provide a unified API for the users of our library. Inspiring by Cypher [7], a SQL-inspired graph query language, we implement the second layer of abstraction for our query model. The basic idea is to define a streaming graph query language, which enables user-defined queries with a group of pre-defined syntax.

The followings are statements supported in our experimental streaming graph query language.

- **Select**: The select statement defines a graph structure that we want to find out from the current streaming graph.
- **Ask**: The ask statement defines a graph structure but only checks if it exists in the current streaming graph.
- **Where**: The where statement adds conditions to the edges or vertices defined in select or ask statement.
- **Return**: The return statement is to define the needed result from the found structures in select statement. This statement is only required while using select statement.
- **GroupBy**: The groupby statement is to group the found structures in select statement.
- **Having**: The having statement is usually used with the groupby statement. It can add conditions to filter partial grouping results.
- **OrderBy**: The orderby statement is to sort the order of output result from select statement.

Additionally, our library also supports some useful functions for aggregating or calculating the results.

- **Sum**: The sum function is to calculate the aggregated value of some numeric values.
- **Count**: The count function is to count the times that vertices or edges appear in found graph structures from select statement. The function also supports distinct keyword to filter duplicate records.
- **Concat**: The concat function is to concatenate the character values in return statement.
- **group_concat**: Similar to concat function, the group_concat function is just to concatenate the character values belongs to same group. It is usually used with the groupby statement.
CHAPTER 3. LIBRARY FOUNDATIONS

Listing 3.2: Examples of Cypher-like query. The first query is to check if there is a path between account 1 and account 10 that the length of this path is no more than 4? The second query is to calculate how much money Bob has received from all other bank accounts?

3.3 Iterative scatter-gather model

The streaming property graph model and two different query models defined in previous sections show the input of our library. In this section, we introduce an iterative scatter-gather model to process these graph and query events. The iterative scatter-gather model can be divided into two models: the scatter-gather model and iterative processing model.

Scatter-gather model

The idea of scatter-gather model comes from the Map-Reduce computation model [25]. Let us consider World Wide Web as an example: if we want to know if two websites are linked with each other, we can check if website A is linked to website B and if website B is linked to website A at the same time. In this way, we can increase the parallelism and execute the query more efficiently. Given any graph-related queries, if the graph structure that we want to detect is already completely known to us, we can fully parallelize the query by scattering the queries to all graph partitions containing necessary information. Later, these graph information collected from different partitions can be aggregated in \textit{gather} phase.

Iterative processing model

The scatter-gather model is for graph-related queries without any hidden graph structure; those with hidden graph structure cannot be done in one step. For example, the shortest path algorithm is a common problem in graph computation. But we have no information about the intermediate vertices between two vertices and thus no way to parallelize the query. The query can not continue until the source vertex finds out all reachable vertices. To use the information obtained from last iteration, we develop this iterative processing model. After the \textit{gather} phase, we need to define a rule to determine if the currently collected information is enough for calculating the final results. Otherwise, the query should go to next iteration until there is no more information can be retrieved from the graph partitions.

Figure 3.3 shows the iterative scatter-gather model.
CHAPTER 3. LIBRARY FOUNDATIONS

Figure 3.3: Iterative scatter-gather model: The computation model mainly contains three phases. Scatter phase is to broadcast the graph and query events to the partitions where they should go. Iterate phase is to store the graph events and execute the query according to these stored graph events. Gather phase is to aggregate the query results from different partitions in iterate phase and thus decide if the query enters next iteration.

3.4 Graph partition strategy

With the definition of streaming property graph model, we further discuss how to store a streaming graph in distributed partitions. When a graph is divided into a series of vertex or edge events, it increases the difficulty to store the graph efficiently for further computation. Therefore, we need efficient graph partition strategies to decide the partition of each incoming graph events. The major goals of graph partition strategy are load balancing and minimum communication cost. Load balancing means the graph vertices or edges are evenly distributed on each partition. To reduce the possible communication cost in further graph computation, we need to minimize the number of vertex or edge cuts. A cut is generated when a vertex or edge is distributed to more than one partition, and thus they need to communicate or synchronize with each other. With more cuts generated by a partition strategy, we can expect a corresponding amount of communication caused by these cuts. However, there is no perfect graph partition strategy for all graph algorithms or queries. For any strategy works perfectly with some queries, counterexamples always exist for certain other cases.

Figure 3.4 shows three different hash partition strategies.

Besides avoiding unnecessary graph cuts, the graph partition strategy should also be fast and easy to scale. The strategy should not take too much time to decide the location of incoming graph events, because it will easily lead to congestion. Furthermore, an incoming query also needs to efficiently decide which partition it should be delivered to. These fast decisions are crucial for reducing the latency to process graph and query events. Scaling is also important for a streaming graph partition strategy, because streaming graph events are usually loaded in parallel, a good partition strategy should be able to make decisions independently.

Moreover, there are also some limitations while partitioning a streaming graph.

• The decision-making is only based on partial graph. There is no global understanding before the whole graph is completely loaded.

• The order of streaming graph events has a huge influence on the performance. The worst case scenario would be that, if all following graph events are distributed to the same partition, the workload will be extremely imbalanced. Therefore, a good partition strategy should adapt to different order of streaming graph events.

• A graph event is usually immutable after being delivered to a graph partition. This immutable graph can lead to significant performance loss. Although some repartition strategies
Figure 3.4: **Three different hash partition strategies:** Assume these vertices are bank accounts and the edges are transactions. The first strategy is good for calculating how much money a specific account transferred to others. The second strategy is efficient for computing how much money a specific account received from all other accounts. As for the last strategy, it is useful for counting all historical transactions between two specific accounts.

Based on these requirements and limitations, our library supports three typical graph partition strategies: **simple hash strategy**, **incremental hash strategy** and **repartition hash strategy**. Each of these strategies performs well for a particular type of graph-related queries. Chapter 5 will further discuss the performance of these different partition strategies. The details of these three strategies are explained as follows.

**Simple hash strategy**

All these three partition strategies are hash-based. In this way, the strategy can decide the location of each graph event quickly. The simple hash strategy is easy to understand: for each incoming edge, it computes the partition ID using a modulo function. The sparse graph is usually stored as an adjacency list, which contains each vertex and its connected vertices as key-value pairs. So the hash function is based on source vertex ID.

\[
\text{PartitionID} = \text{SourceVertexID} \mod \text{TotalPartitionNumber} \quad (3.1)
\]

In this way, all edges with the same source vertex are distributed to the same partition. The strategy is good for fast decision and need not store additional partition information. With such partition strategy, if a query wants to find out all edges with a specific source vertex, the query can also be delivered to the corresponding partition.

**Incremental hash strategy**

The simple hash strategy is fast and easy to understand. However, the strategy can lead to imbalanced workload if all incoming edge events get same partition ID after being hashed by same hash function. To obtain a balanced workload regardless of order of incoming edges, we develop
CHAPTER 3. LIBRARY FOUNDATIONS

Figure 3.5: Simple Hash Strategy: The blue circles represent the source vertices of each edge, which are used to decide the location of each edge.

incremental hash strategy. This strategy comes from an existing graph processing system, GraphA introduced in Section 2.2.

The incremental hash function is also based on the source vertex ID, but it uses multiple hash functions instead of one. The strategy maintains a global partition table, and for each partition, there is a capacity for each source vertex. When a source vertex has reached its capacity on one partition, it will start using the next hash function. The partition table records the ID of hash function being used by each vertex. In this way, the problem of high-degree vertices is solved. The edges related with a high-degree vertex can be evenly distributed. The edges related with a low-degree vertex are distributed to the same partition. With such partition strategy, a query needs to check the partition table before being delivered to the corresponding graph partitions.

Figure 3.6: Comparison between Simple and Incremental Hash Strategy: The left figure is using simple hash strategy, all edges start from vertex 1 are distributed to the first partition. The right figure is using incremental hash function. The capacity for each partition is 4. Therefore, each partition can store at most 4 edges start from vertex 1. Now vertex 1 is using the third hash function.

However, there is also a bottleneck for incremental hash strategy, because we need to maintain a global partition table, we need to record all the incoming edges as well as the corresponding hash function ID. But the incoming edges are loaded in parallel, these parallel data streams are

A streaming graph library for Apache Flink
independent of each other. Therefore, we have to disable the parallel processing or synchronize partition table between each data stream, and both solutions can cause significant performance loss.

Repartition hash strategy

Incremental hash strategy is an adaptive approach because it can adjust well to high-degree vertices. However, the graph is still immutable once the location of an incoming edge is decided. Therefore, we try to come up with a new strategy which enables graph repartition if the performance of current graph partitions is unacceptable. The idea of repartition hash strategy is inspired by Vaquero LM et al [38]. The core of repartition hash strategy is label propagation. For any vertex, it prefers to move to a partition that contains more of its neighbors. In this way, each partition can include as much information as possible for a specific query. In order to implement repartition hash strategy, each partition needs to maintain a local partition table. Then an edge event can decide if it should be stored on this partition according to the local partition table. Furthermore, the repartition hash strategy uses a probability function to calculate the possibility if a vertex should migrate to another partition. Apparently, the more neighbors a vertex has on another partition, it is more likely to migrate to that partition.

\[ P = 1 - P_0^N \]  

(3.2)

\( P_0 \) is the base probability defined by users, a smaller base probability means a vertex is more likely to migrate. \( N \) is the number of neighbors appear on another partition. With this probability function, the strategy can avoid unnecessary system overhead.

Figure 3.7: An example of repartition hash strategy: The edge events colored red are already loaded. For the left figure, the next incoming edge event is (1,2). Since vertex 2 is already on partition 2, vertex 1 is more likely to migrate to second partition because its neighbor vertex 2 is on that partition. As for the right figure, the next incoming edge event is (2,4), which will be distributed to second partition. Since one of its neighbor vertex 4 is on first partition and another neighbor vertex 5 is on second partition, it will remain stored on the second partition.

The repartition hash strategy needs to avoid an issue called chase each other. For example, two incoming events (1,2) and (2,1) arrive on first and second partition at the same time. As vertex 2 and vertex 1 are stored on second and first partition, these two events both want to migrate to the other partition. The probability function mentioned above is also useful in this situation.
Chapter 4

Design

This chapter mainly discusses the design of our library from practical perspective. Based on the graph computation model defined in Chapter 3, we review the restrictions within the design in Section 4.1. We will see how to fit our solution into Apache Flink’s streaming processing API. Section 4.2 illustrates the key design decisions in our library. These decisions are beneficial to either ease-of-use or efficiency. The last Section 4.3 shows the organization of our library. We will give an overview of our library’s visualized Flink plan.

4.1 Restrictions within the design

As our library is built on top of Apache Flink, and therefore, leverages its advanced features such as distributed processing, resource management and fault tolerance. However, Apache Flink also has some restrictions that limit the design of our library.

- Similar to other data processing platform, Apache Flink uses a directed acyclic graph (DAG) to organize the processing logic. DAG is a directed acyclic graph which only supports sequential data flow. Therefore, the downstream operators cannot provide feedback to upstream operators.

- Although Apache Flink only supports DAG, it enables a feature called iterative streaming program. The idea is to split or filter the downstream data, and part of these data will be added to an iteration sink. This iteration sink serves as part of upstream’s data source.

- Apache Flink can support distributed processing by setting the number of parallelism. In this way, the incoming data stream is loaded in parallel. Each data stream is assigned to one TaskManager created by JobManager in Apache Flink. These separate TaskManager are physically isolated. Therefore, shared memory is not supported between parallel TaskManager.

4.2 Key design decisions

To fit within Apache Flink’s boundaries, we make several key design decisions. Since Apache Flink is an event-driven data processing platform, we design a basic Event class, each event contains a unique ID which is randomly generated. A timestamp is also assigned to an event when the event is first created. Additionally, each event has a partitionID, which is responsible for controlling the data flow.

```java
public class Event implements Serializable {
    private long eventId;
    private long timestamp;
}
```

A streaming graph library for Apache Flink 27
CHAPTER 4. DESIGN

Based on these common properties in \textit{Event} class, we further design the \textit{Vertex}, \textit{Edge} and \textit{Query} class. All these three classes extend the \textit{Event} class.

\begin{verbatim}
public class Vertex extends Event {
    private long vertexId;
    private String vertexLabel;
    private Map<String, List<Property<Object>>> properties;
}
\end{verbatim}

\textbf{Listing 4.2: Vertex Class}

\begin{verbatim}
public class Edge extends Event {
    private String edgeId;
    private Vertex source;
    private Vertex target;
    private String edgeLabel;
    private Map<String, List<Property<Object>>> properties;
    private boolean isLoopback;
    private long loopbackTargetPartitionId;
}
\end{verbatim}

\textbf{Listing 4.3: Edge Class}

\subsection{4.2.1 Vertex-cut or Edge-cut}

Different distributed graph processing systems choose different methods to cut a graph. As we have discussed in Section 2.1, edge-cut method is good for load balance but may cause excess communication between high-degree vertices. Vertex-cut method reduces the communication between high-degree vertices but the communication between low-degree vertices becomes a major issue. Either method can lead to a synchronization problem between duplicated vertices or edges. Considering the streaming property graph model defined in Section 3.1, our library makes two distributions of a streaming property graph. The first distribution is an edge distribution which cut a graph by vertices, and the second one is a vertex distribution that cuts a graph by edges. In this way, each edge or vertex only appears on one partition in its respective distribution. Although the additional distribution brings redundancy to our library, we avoid the efforts for synchronizing edge or vertex properties.

The edge distribution is covered by the red circle in Figure 4.2, and the vertex distribution is covered by the brown circle. Such an organization is good for edge-oriented queries. A query can compute the qualified graph structure and filter the unqualified ones with edge properties. Then these qualified graph structures are forwarded to vertex distribution for further filtering or vertex property aggregation.

\begin{verbatim}
public Graph() {
    // key is all source vertex IDs stored on this graph partition, value is a set of vertex IDs connected to the key vertex
    private Map<Long, Set<Long>> adjacencyList;
    // key is all vertex IDs stored on this graph partition, value is the label of key vertex
    private Map<Long, String> vertexList; // ignore vertex properties
    // key is the ID of each edge, identified by source, target vertex and edge label, value is the corresponding edge including all edge properties
    private Map<String, Edge> edgeList;
}
\end{verbatim}

\textbf{Listing 4.4: Graph storage: Edge distribution}
4.2.2 Implementation of partition strategy

In order to store a large-scale streaming graph efficiently and enable fast query execution, our library supports three typical graph partition strategies. User needs to choose a partition strategy before launching Flink’s execution plan. Since Flink provides the ability to change physical partitioning of data streams, we can implement different graph partition strategies with the provided \( \text{partitionCustom} \) function.

In general, all these strategies use source vertex ID of each edge event to compute the target partition ID. This computation is done on the Flink Operator ahead of storing edge events, we call this operator \( \text{scatter queries and edges} \), which is the \( \text{operator 11} \) in Figure 4.2. The operator for storing edge events is \( \text{operator 13} \), which is named as \( \text{edge distribution} \).

- **Simple Hash Strategy**: With simple hash strategy, there is no need to store any global partition table or local partition table. Because the partitioning completely depends on the computed target partition ID, \( \text{Operator 13} \) only stores the edge events from upstream.

- **Incremental Hash Strategy**: With incremental hash strategy, we want to evenly distribute those edges starting with high-degree vertices. Therefore, we introduce multiple incremental hash functions. The number of incremental hash functions is the same as the global parallelism or the number of subgraph partitions. We call them incremental hash functions because they are continuous modular functions. Each hash function simply adds 1 to the result of its previous hash function. Users of the library also need to define a capacity for this strategy. The capacity limits the maximum number of edges starting from each vertex storing in each subgraph partition. Once a partition has reached its capacity, following edge events starting with this vertex will use the next hash function. Therefore, we need to maintain a global partition table in \( \text{operator 11} \). The table records the hash function being used by each vertex. Furthermore, the parallelism of \( \text{operator 11} \) has to be set to 1. Otherwise, this operator will generate multiple partition tables because each parallel pipeline is independent of each other. And Flink does not allow communication or synchronization between these partition tables. Obviously, this design will become a bottleneck, but it can also provide good performance for breadth-first search. Users need to weigh the pros and cons to determine the strategy.

- **Repartition Hash Strategy**: This strategy is more complex than the other two in that it is more adaptive to the dynamic changes in streaming graph. With repartition hash strategy, \( \text{operator 11} \) will broadcast all edge events to the downstream \( \text{operator 13} \). The partition ID of each edge event is set to \(-1\), which is the symbol for broadcasting. \( \text{Operator 11} \) will replicate the same number of edge events as parallelism. The partition ID of these duplicate events will be set from minus one to minus the number of parallelism. Then the \( \text{partitionCustom} \) function will use the absolute number to compute the partitioning ID. In this way, all downstream pipelines can receive the same edge events, these pipelines first store the edge events into a local partition table, which records the location of all existing vertices. Afterward, the edge event will compute the number of neighbors of its source vertex on each subgraph partition. It will use the number of neighbors to compute a probability which decides if this edge event should be redistributed to the partition with maximum number of neighbors. The probability function is defined in Section 3.4. Users can adjust the base
probability in this function. A smaller base probability will lead to more repartition events. However, the corresponding edge distribution can perform better for depth-first search.

```java
public class GeneralHashKeySelector implements KeySelector<Event, Long> {
    @Override
    public Long getKey(Event event) throws Exception {
        return event.getPartitionId();
    }
}
```

```java
public class HashPartitioner implements Partitioner<Long> {
    @Override
    public int partition(Long key, int numPartitions) {
        if (key < 0)
            return (int) (-key % numPartitions);
        else
            return (int) (key % numPartitions);
    }
}
```

Listing 4.6: KeySelector and HashPartitioner: These two classes define our customized partition strategy.

### 4.2.3 A redundancy mechanism to accelerate queries

We introduce multiple graph partition strategies to adapt to different queries. Furthermore, we also design a redundancy mechanism to accelerate a particular type of query. As we have discussed in the beginning of this section, each subgraph is stored as an adjacency list in edge distribution. Adjacency list is actually a HashMap data structure, the key is all events’ source vertex ID, and the value is a set of each key’s linked vertex ID. In this way, we can easily find out all out-degrees of a vertex. However, if we want to find out the in-degrees of a vertex, we have to foreach all values of the HashMap. If the value HashSet contains the vertex, the corresponding key is the in-degree of this vertex. This is inefficient especially for processing the large-scale streaming graph, because it will take a long time to check all adjacency lists. Therefore, we introduce a redundancy mechanism that generates an edge event with reverse direction for each incoming edge events. In this way, we can also store an edge in both directions in adjacency list. However, this redundancy mechanism cannot be applied to all edge events, because it will double the size of edge events. Therefore, the library decides to make it optional. Users can state a list of edge labels and their reverse labels before launching the execution plan. Once the Flink job receives an edge labeled as the user stated, Flink will generate a reverse edge event with the reverse label and add it to the datastream.

```java
public class EdgeDuplication extends RichFlatMapFunction<Event, Event> {
    @Override
    public void flatMap(Event event, Collector<Event> collector) throws Exception {
        ParameterTool pt = (ParameterTool) this.getRuntimeContext().getExecutionConfig().getGlobalJobParameters();
        collector.collect(event);
        String eventType = Utility.getEventType(event);
        if (eventType.equals("Edge")) {
            Edge e = (Edge) event;
            String reverse = pt.get(e.getEdgeLabel(), "none");
            if (!reverse.equals("none")) {
                Event duplicate = Utility.reverseEdge(e, reverse);
                ((Edge) duplicate).setProperties(e.getProperties());
                collector.collect(duplicate);
            }
        }
    }
}
```

30 A streaming graph library for Apache Flink
4.2.4 Implementation of iterative scatter-gather model

In order to implement a generic library, we need to abstract the iterative scatter-gather model. Therefore, we define an abstract Query class, which extends the Event class. Users can extend this abstract Query class and define their own graph-related queries accordingly. We give a list of instructions to implement user-defined queries.

- The first step is to implement the parseString function, which can transform a query string into a query class. The customized query can also include additional parameters, which can be retrieved from the query string.

- If a customized query includes additional parameters, it should also implement the default and copy constructor. Because these constructors are frequently used while scattering the queries.

- The core step is to implement scatter, iterate and gather function. If a query can collect necessary information from multiple downstream graph partitions, scatter function should be implemented to scatter the queries. Otherwise, the scatter function just pass the query event to downstream. The iterate function must be implemented, which is responsible for executing the major query task. The gather function should be implemented if the query event is scattered in scatter function.

- Additionally, the function scatterQueryResultForVertexPropertyAggregation, aggregateVertexProperty and gatherAggregatedVertices are for vertex property aggregation. Since the vertices as well as vertex properties are stored in vertex distribution, we need to aggregate these vertex properties with the query results.

- After aggregating the vertex properties, some of the query results may be unqualified and thus should be filtered. Therefore, filterByVertexProperty should be implemented.

- The function OutputResult is for generating query results. Users need to define their own logic to output the qualified or unqualified results. Users can also implement the toQueryString function to formulate a query string.

```java
public abstract class Query extends Event {
    // parameters for scatter–gather model
    private boolean isScattered;
    private int scatteredNum;

    // parameters for iterative model
    private boolean isIterativeQuery;
    private int step = 0;

    // parameters for vertex properties aggregation and filtering
    private int scatteredVertexNumForPropertyAggregation;
    private boolean isFilteredByVertexProperty;

    // constructor
    public Query() {
        super();
    }

    // copy constructor
    public Query(Query q) {
        super(q);
        this.isScattered = q.isScattered;
    }
}
```
CHAPTER 4. DESIGN

Listing 4.8: Query abstraction: Users should extend the Query class and thus define their own query logic

4.2.5 An experimental Cypher-like Query API

On top of the iterative scatter-gather model, we further encapsulate this Cypher-like query API and expose it to users. It is difficult to come out with a unified logic for detecting various graph patterns. After a lot of optimization attempts, we finally come up with this experimental API, many aspects of which are still to be further optimized.

Figure 4.1 shows the processing logic of this query model. Initially, it will parse a user-defined query string to a list of edges. Each edge’s target vertex is its following edge’s source vertex. In this way, we check one edge in one iteration and naturally move to the next edge in next iteration. For example, we know the source vertex ID of first edge can be 1, 3, 5. Then we can replicate three queries in the scatter phase of iterative scatter-gather model and set partition ID as 1, 3, 5 respectively. These queries will be sent to the corresponding graph partitions based on the choice of graph partition strategy. In iterate phase, each query looks up the linked vertices and add it to mapping’s value side. The gather phase is responsible for gathering the results collected by different scattered queries. Then one iteration is done, and we get the result that 1 is linked to 4 and 5, 2 is linked to 1 and 3, 3 is linked to 4. If there are more edges in the list, the query will enter next iteration and start with values found in last iteration. In our case, it will start from vertex 1, 3, 4, 5, 6. If no values is found in last iteration, the query is failed and will be forwarded to the output stream. Furthermore, if an edge contains hidden graph structure, the step number will not change until it reaches its maximum length of hidden graph structure. For example, if we want to check if there is a path between vertex A and B that the length is no more than 4. The edge has to store four mappings which collect all possible paths between A and B. Through this
method, we can get a tree structure that contains all graph patterns in accordance with user’s query.

![Figure 4.1: Process logic of Cypher-like Query API](image)

Figure 4.1: **Process logic of Cypher-like Query API**: A user-defined query is parsed into a list of edge, each edge is a step within the query. The mappings included in each edge are the detected graph patterns. An edge with hidden structure can contain multiple layers of mappings, otherwise, it only has one layer of mapping.

However, there are some problems with this process logic. Firstly, if the source vertex of first edge is unknown, it will collect all edges in the edge distribution and thus produce a huge traffic in datastream. We have 2 solutions for this problem. On the one hand, we should not do such global query in stream processing, because stream processing is usually designed for the lightweight task that requires low latency. Such global summarization should be executed against historic database. On the other hand, if we know properties of these vertices, we can query the qualified vertices based on these properties in vertex distribution. This method helps filter the unqualified patterns from the very beginning. It is also a good way to cut the overwhelming branches of the tree-structure. In this way, we can further improve the performance of this query model.

### 4.3 Library organization

Since our library is built on top of Apache Flink, this section gives an overview of our library. Figure 4.2 shows a Flink visualized plan, which is an overall process logic for our library. The processing logic can be divided into four parts: data preprocessing, edge distribution, vertex distribution, and finally the output of the query result.

**Data preprocessing**

Since Apache Flink is an event-driven data processing platform, our library is designed to process two types of events including graph edge events and query events. In order to parse the user inputs into edge or query events, the data preprocessing part gives the flexibility to users where they can define their own parse function for edge and query events. These two parsers are `operator`
CHAPTER 4. DESIGN

Figure 4.2: An overview of Flink visualized plan: The execution plan covered by green circle is data preprocessing. The plan covered by red circle is edge distribution. The brown circle covers the vertex distribution part. The blue circle covers the query result output. The execution plan is generated by Flink plan visualizer.

3 and operator 4. Additionally, the graph parser can divide an edge event into one edge event and two vertex events. The edge events including edge properties are forwarded to operator 8, which is used for duplicating certain edge events. Mixed with query events, these edge events are forwarded to edge distribution. The vertex events including vertex properties are forwarded to vertex distribution.

**Edge distribution**

The edge distribution part includes operator 11, operator 13 and operator 15. Operator 11 is used to scatter edge and query events to operator 13. In operator 13, all edge events are stored as part of the graph. The query events collect the related graph information and then flow to operator 15, which is responsible for gathering the query results. Operator 15 decides if each query event should be looped back and continue next iteration. These query events will be forwarded to the vertex distribution part.

**Vertex distribution**

Vertex distribution part receives vertex events from data preprocessing part and stores these vertices distributively. Besides passing vertex events to operator 18, operator 16 also scatters the query events received from edge distribution. These scattered query events are forwarded to corresponding graph vertex partition, and thus aggregate vertex properties in operator 18. Operator 20 assembles these scattered query events with their vertex properties. These assembling query events are forwarded to operator 21, which can filter the unsatisfied query events.

**Query result output**

The combination of edge distribution and vertex distribution formulates the iterative scatter-gather model. The query events are forwarded to the last part after a number of iterations. Operator 24 will output the query results whether these queries are qualified or unqualified.
Chapter 5

Evaluation

This chapter shows the evaluation results of our library. The goal of evaluation is to show if this library is in line with our expectations. The library is supposed to process graph events with a high throughput and execute customized queries with a low latency. In the meantime, the library also needs to show its scalability. We use a dataset called UK-2007 to benchmark the library. The details of this dataset will be described in Section 5.1. Since the order of incoming graph events also influences the performance of a streaming graph library, we introduce the design of our experiments to obtain a general performance of our library in Section 5.2. Based on the evaluation results, we further discuss the performance and limitation of our library in Section 5.3. In the end, we make a conclusion about the results and an exploration about how to choose a graph partition strategy for different user scenarios in Section 5.4.

5.1 Datasets

To evaluate the performance of our library, we use a dataset called UK-2007 [10]. The dataset is a large-scale web graph including 105,896,555 nodes representing pages, connected by approximately 3.7 billion edges representing hyperlinks. As this dataset crawls tons of real-world websites, it should be a natural graph, which is the target of our library. Therefore, we should first verify if it is a natural graph. Based on our discussion in Section 2.1, a natural graph should follow power-law distribution.

![Power-law distribution of UK-2007 dataset](image)

Figure 5.1: **Power-law distribution of UK-2007 dataset**: The x-axis is the number of out degree for each vertex, y-axis is the frequency. The figure shows low-degree vertices have much higher frequency than high-degree vertices.

In natural graph, a small subset of the vertices have a large number of neighbors and most
vertices have a small number of neighbors. This is also what Figure 5.1 shows to us. Since our library is designed to process natural graphs, we use this UK-2007 dataset to generate graphs with different skewness levels to evaluate the adaptability of our library. Also, we use this UK-2007 dataset to generate graphs with different sizes to evaluate the scalability of our library.

5.2 Experiment design

System Specifications Before introducing the experiment design, we first specify the system configuration. All experiments were tested on a server with an Intel 2.60 GHz Core Xeon(R) CPU (E5-2640) which has 8 physical cores and 32 hyper-threads. The machine has 94.1 GB of Installed memory and runs a 64-bit Operating system of Centos Linux 7. All experiments were run with Java 1.8.0 181, Apache Flink 1.4.2 and Apache Kafka 2.11.

Many factors can influence the performance of a streaming graph library, so we need to design different experiments to test the influence of multiple variables. Potential factors that may significantly impact the performance are listed below.

- **Parallelism**: Parallelism is the setting of Flink execution plan. By increasing the parallelism, Flink can launch more TaskManger to complete a task in parallel. In order to demonstrate the good scalability of our library, we need to benchmark the library with different parallelism.

- **Size of Input Graph**: A streaming graph library is usually deployed in production environment and thus should be able to process large-scale streaming graph. As the size of input graph grows, we need to evaluate if a graph edge or vertex can be stored efficiently and a query event can still retrieve the required data with a low latency.

- **Streaming order of Input Graph**: The streaming order of input graph also makes a difference. We assume our library partitions a graph by vertex. In worst case, if all incoming edges are related with one specific vertex, all these edges will be distributed to the same partition. Therefore, the library should be adaptive enough to balance such workload.

- **Graph Partition Strategy**: Although graph partition strategies are usually designed for particular graph queries, we still benchmark the three implemented partition strategies with different queries and discuss their optimal and average performance.

- **Choice of query abstraction model**: We introduce two abstraction layers for the query computation model in this library (see Section 3.2). The first layer is more flexible and hence user can optimize the processing logic as they want. The second layer provides an ease-to-use API; however, the generic API may lead to performance loss for some particular queries. Therefore, we need to compare the performance with different query abstraction models.

Taking all these factors into consideration, we can design our experiments in four steps.

The **first step** is to benchmark the general performance of our library. Since our library is designed to process graph events and query events separately, we should evaluate the general performance of this library even though no queries are predefined. We use three evaluation standards to measure the general performance.

- **Number of graph cuts**: Graph cuts mean potential cross-partition communication. By limiting the number of graph cuts, we can expect fewer communication needs.

- **Number of edges on each partition**: The number of edge on each partition shows if the workload is well balanced. An imbalanced graph distribution can also introduce heavy query workload to one partition. Actually, we make two graph distribution in Section 4.3, and edge distribution is the major cause of imbalanced workload. Therefore, we count the number of edges in edge distribution.
• **Throughput**: The throughput is also evaluated as it is not simply the sum of graph edges on each partition. A repartition graph partition strategy can generate iterative graph events and hence the real throughput could be much higher than the sum of graph edges. Also, multi-edges are stored as one edge with multiple properties in our library, which can also cause inequality between throughput and sum of edges.

The **second step** is to evaluate the performance with different graph sizes. Since the library is designed for large-scale streaming graph, scalability is important for our library. Therefore, we execute different queries against different sizes of graph.

The **third step** is to benchmark the library by using different query abstraction models. We execute different queries implemented with different abstractions and compare the performance difference between abstractions.

The **fourth step** is to execute queries on top of different graph partition strategies. As discussed in Section 3.4, a query event can collect all necessary information on a graph partition if the graph is well partitioned. However, there is no such ideal strategy, we need to find a strategy which can be generally good for all queries or specifically optimized for a particular query. By evaluating different strategies, we make an exploration of how to choose a graph partition strategy.

We execute three different types of queries within the above three steps. Each type of query refers to a specific graph pattern.

• **Query Category 1**: The first type is to detect graph pattern with hidden structure. For example, such a query can be *Is there a shortest path between vertex A and B that the length is no more than 4?*. The in-between vertices between vertex A and B are the hidden structure we want to detect.

• **Query Category 2**: The second type is to detect graph pattern without hidden structure. Specifically, the starting vertex is defined, for example, such a query can be *how many distinct vertices are linked to this start vertex?*.

• **Query Category 3**: The third type is similar to the second type. The only difference is that the end vertex is defined but the start vertex is not. We try to benchmark such queries because we store each subgraph as an adjacency list. We can easily search edges with a starting vertex but have to traverse all key-value pairs if only end vertex is defined. To accelerate such queries, we design a redundancy mechanism, this type of query can verify the validity of this mechanism.

For each type of query, we generate 1000 queries with random vertex IDs or properties and collect the following statistics on the query completion time.

• **Average Completion Time**: We count the completion time of each query instance and calculate the average completion time.

• **Standard deviation of Completion Time**: We also calculate the standard deviation of these query instances. Standard deviation evaluates if an execution configuration can provide stable performance for a particular type of query.

• **Maximum Completion Time**: The maximum completion time shows the worst case for a particular query.

• **Minimum Completion Time**: The minimum completion time shows the optimum case for a particular query.

To summarize, we evaluate our library by following different configurations.

• **Graph size**: We consider three graph dataset with 1 million, 10 million and 100 million edges. These datasets are subgraph of the UK-2007 dataset.
• **Streaming order**: We design a balanced and skewed graph stream.

• **Graph partition strategy**: We can choose simple hash, incremental hash or repartition hash strategy.

• **Flink parallelism**: The parallelism is set from 1, 2, 4, 8 to 16.

• **Query abstraction model**: The models are parameter-based query model and Cypher-like API.

### 5.3 Results

We evaluate the results from four aspects. The first step is to discuss the general performance of the library. The general performance is evaluated based on throughput, total number of vertex cuts and number of edges on each graph partition. The general performance shows if a library can store and index a streaming graph quickly and efficiently. The other steps are to test the performance with different graph related queries under different execution settings. The graph queries are grouped into three categories. The settings include choice of graph partition strategy, size of streaming graph and different implementation of query model. The performance is evaluated based on the query completion time.

#### General performance

To benchmark the general performance, we first discuss the throughput of our library. For a streaming graph library, it should be able to process large amounts of streaming events continuously. Figure 5.2 shows the throughput with different parallelism and different graph partition strategies. Parallelism means the number of parallel executors which are responsible for processing the data pipeline. The throughput increases as the parallelism becomes larger, we get the maximum throughput when the parallelism is set as 8. The throughput does not increase anymore when the parallelism is set as 16 because our testing machine has 8 CPU cores. The choice of graph partition strategy also decides the throughput of our library. Figure 5.2 shows that simple hash strategy has the highest throughput because there is no need to store any partition table or repartition any streaming events. With simple hash strategy and a parallelism of 8, we get a maximum throughput which is 200,000 events per second. In real-world cases, we can improve the throughput by increasing the parallelism. As for the partition strategy, the simple hash strategy provides the fastest way to decide the partition of an incoming event. However, throughput is not the only evaluation standard, we also need to consider if a strategy is good for a future graph-related query.

The performance of a graph partition strategy is usually evaluated by load balancing and cross-partition communication cost. Load balancing means if a graph is evenly distributed on all graph partitions, because a skewed partition result can easily lead to imbalanced workload and thus become a bottleneck of a library. In our case, the load balancing is evaluated by counting the number of edge events on each parallel task executors. Figure 5.3 shows the edge distribution of three different graph partition strategies. We can see that the simple hash strategy and incremental hash strategy have a similar performance, because all their 8 parallel task executors receive similar number of edge events all the time. But the repartition hash strategy gets an imbalanced edge distribution. Because the repartition hash strategy can redistribute an edge event based on label propagation, which provides no guarantee for load balancing. Therefore, it is not a good idea to choose repartition hash strategy if balanced workload is the most important standard.

However, the distribution of a graph also has a huge impact on load balancing, because our library uses source vertex ID of each edge to decide the belonging partition of each event, if these source vertex IDs do not follow a balanced distribution, the edge events are not evenly distributed accordingly. For example, if we distribute a streaming graph into 4 graph partitions, and all high-degree vertices get the same remainder after dividing by 4, then all edges that start with these...
CHAPTER 5. EVALUATION

Figure 5.2: Total throughput with different parallelism: The x-axis is time, y-axis is the events being loaded. The dotted lines show the throughput with different graph partition strategies. The solid lines show the throughput of simple hash partition strategy with different parallelism.

Figure 5.3: Edge distribution with a balanced graph stream: The x-axis is time, y-axis is the number of edges on each partition. The test is based on a parallelism of 8. So each partition strategy has 8 solid lines in the figure. The orange one represents the simple hash strategy, the blue represents the incremental hash strategy and the green one represents the repartition hash strategy.
high-degree vertices will be distributed to the same partition. This partition will therefore become heavily loaded. Additionally, even the distribution of a graph is balanced, but the streaming order of edge events can still cause an imbalanced workload at some time points. Therefore, our library solves the problem by supporting incremental hash strategies. Figure 5.4 compares the performance of simple hash strategy and incremental hash strategy with a skewed streaming graph. We can see that the incremental hash strategy has a much better performance because all the 8 blue solid lines are almost overlapped.

![Edge distribution under parallelism 8 with skewed data stream](image)

**Figure 5.4: Edge distribution with a skewed graph stream:** The x-axis is time, y-axis is the number of edges on each partition. The test is based on a parallelism of 8. The orange lines represent the simple hash strategy and the blue lines represent the incremental hash strategy.

Besides load balancing, we also need to consider cross-partition communication cost. Because a graph is partitioned into multiple subgraphs, which are usually insufficient for providing enough information a graph query or computation. The potential communication cost can be evaluated by the number of graph cuts, because these cuts bring the need for communication. Since our library mainly cuts the graph by vertex, we can evaluate the performance by counting the number of vertex-cuts. Figure 5.5 shows the number of vertex-cuts under different parallelisms with simple hash strategy. We can see that a higher parallelism greatly increases the number of vertex-cuts. So there is a trade-off between throughput and cross-partition communication cost. We should choose a moderate parallelism for our Flink execution plan.

Figure 5.6 shows the number of vertex-cuts generated by different graph partition strategies. We can see that the simple hash strategy produces the least number of vertex-cuts. The repartition hash strategy produces a large number of vertex-cuts because the strategy depends on label propagation.

**Performance under different graph sizes**

In this part, we test the performance of our library with different sizes of streaming graph. We mainly use three queries to do the benchmark, that details of which can be found in Appendix B.

Table 5.1 shows the statistics of the benchmark result. As Query 1 contains hidden structure, we can see that the performance gets worse and worse as the size of graph grows. When we load 100 million edge events, the average completion time is 2.65 seconds, with the maximum completion time even reaches 23.8 seconds. The poor performance is caused by the unified logic
CHAPTER 5. EVALUATION

Figure 5.5: **Total vertex-cuts under different parallelism with simple hash strategy:** The x-axis is time, y-axis is total number of vertex-cuts. Each line represents different settings of parallelism.

Figure 5.6: **Total number of vertex-cuts with different partition strategies:** The x-axis is the number of loaded edge events, y-axis is the total number of vertex-cuts. The green bar represents the simple hash strategy which has the minimum vertex-cuts. The blue bar represents the incremental hash strategy. The orange bar represents the repartition hash strategy which has the maximum number of vertex-cuts.

of Cypher-like API, which offers poor optimization on such kind of query. As for Query 2, we can see that the performance remains the same even the graph contains 100 million edges. All query instances can be done in one second and the average completion time is about 0.4 second. We can make a comparison between Query 2 and Query 3. Neither of these two queries includes a hidden structure, but the performance of Query 3 is getting worse as the graph size grows. The difference is that the target vertex in Query 2’s first iteration is unknown, whereas the source vertex is unknown in Query 3. Due to the way we store each subgraph, Query 2 apparently has a better performance than Query 3. This is also evidence that the redundancy mechanism we...
designed is useful. With it, we can easily look up any edge with either its source vertex or target vertex once we store two replications with reverse directions for each edge.

The first 3 figures in Figure 5.7 shows the distribution of each query when the graph contains 100 million edges. For Query 1, we can see two peaks in the histogram. Because the query is actually a shortest path query, and we generate random vertex IDs as the starting and end point of the shortest path query. Therefore, there is no existing paths for most of the queries, which simply fails and terminate the iterations in the beginning. These queries are usually done in 1 second. The second peak appears around the coordinate of 5 seconds. This peak shows the average time usage of successful queries that actually find a shortest path. However, we can also see some long tails on the right of the histogram. Some of the queries even take more than 20 seconds. Generally, there are two methods to improve the poor performance. The first one is to use another graph partition strategy, which can produce a better edge distribution for this query. The second one is to use the parameter-based query model, where we can apply some optimizations in the processing logic. For Query 2 and Query 3, the distributions are close to normal distribution. Because there is no hidden structure in these queries, and hence they can be completed after certain iterations.

| Table 5.1: Performance Statistics under different graph sizes (Unit: ms) using parallelism 8, simple hash strategy and Cypher-like API |
|---------------------------------|-------|------|----------|-----|
| Query 1 (10,000 edge events)    | 239   | 1412 | 902.45   | 2356.78 |
| Query 2 (10,000 edge events)    | 120   | 897  | 357.72   | 99.28  |
| Query 3 (10,000 edge events)    | 189   | 854  | 467      | 112.47 |
| Query 1 (1 million edge events) | 178   | 7045 | 1757.92  | 3024.32 |
| Query 2 (1 million edge events) | 140   | 998  | 376.24   | 108.92 |
| Query 3 (1 million edge events) | 732   | 2199 | 1231.24  | 278.23 |
| Query 1 (100 million edge events)| 267   | 23805| 2654.75  | 3758.53 |
| Query 2 (100 million edge events)| 131   | 1088 | 389.65   | 104.95 |
| Query 3 (100 million edge events)| 961   | 3017 | 1766.57  | 328.47 |

Performance under different query models

In this part, we benchmark the performance of different query models. As the parameter-based query model is good for optimizing process logic, we decide to make an optimized query for shortest path query. In the optimized version, the query can immediately quit the iterative stream once it finds out a path in one iteration. In this case, a query does not need to retrieve all paths and sort out the shortest one. The details of the queries can be found in Appendix B.

Table 5.2 shows that the performance is significantly improved with the parameter-based query model. The average time usage for Query 1 is 0.7 second, and the maximum time usage is 2.2 seconds. The standard deviation also indicates the improvement of stability. Although parameter-based query model offers a better performance, it actually relies on users’ own effort to optimize the processing logic. The parameter-based query model is a good option when the Cypher-like API does not fulfill users’ performance requirement. However, the goal of our library is to provide an ease-of-use and generic API to the users, we should continue working on the optimization of Cypher-like API.

The last figure in Figure 5.7 shows the distribution of time usage when we use the parameter-based query model. We can still see two peaks which fall on 0.3 second and 1 second respectively. Analyzing this two peaks, we can say that most failed queries can be done in around 0.3 second and those successful queries are usually completed in around 1 second. Furthermore, the distribution shows a shorter tail in the right side comparing to that in the first figure. Because the shortest path query does not need to retrieve all connected paths as it does in Cypher-like API.
Table 5.2: Performance Statistics under different query models (Unit: ms) using parallelism 8, simple hash strategy and 100 million edge events

<table>
<thead>
<tr>
<th>Query 1 (Parameter-based query model)</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>Std</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query 1 (Cypher-like query model)</td>
<td>267</td>
<td>23805</td>
<td>2654.75</td>
<td>3758.53</td>
</tr>
</tbody>
</table>

Figure 5.7: Four histograms of query time usage with different configurations

Performance under different graph partition strategies

In this part, we try to verify the performance of different graph partition strategies. We did the benchmark with 2 queries with hidden graph structure. The first one is shortest path query. The second one is to find out all paths between two vertices. The details of the queries can be found in Appendix B.

In Section 3.4, we have discussed the expected performance of three different graph partition strategies. To verify these expectations, we use two queries in this set of experiment. These 2 queries belong to query category 1 defined in Section 5.2. Query 1-2 is to check if there is an existing path between two vertices that the length of the path is less than 4. In this case, we prefer to use the depth-first search, because the query can simply return true once it detects a path. In this way, the query does not need to enter next iteration to check the paths on other graph partitions. As for Query 1-3, the goal is to find out all existing paths between two vertices that the length of the path is less than 4. Under such context, it is better to use the breadth-first search, because we need to list all possible connected vertices in each iteration. The statistics in Table 5.3 confirm our expectations for each partition strategy. We can see that Query 1-3 has the best performance while using the incremental hash strategy, because incremental hash strategy performs well for breadth-first search. Query 1-2 has the best performance under repartition hash strategy. With repartition hash strategy, an edge is tending to move to a graph partition that contains maximum number of its neighbors. Therefore, it is much easier to detect a path on one partition without communicating with other graph partitions. With repartition hash strategy, it only takes an average time of 0.5 second to complete an instance of Query 1-2.

The experiments with different partition strategies clearly demonstrate the significant performance improvement when we choose a proper strategy for a particular query. However, we should always notice that there is no strategy works perfectly for all queries. Especially when a query is specifically designed for a particular type of query, it usually has a poor performance on other queries. The idea of our library is to offer different options to the users. With the flexibility, users can combine a proper configuration for their own situations.
### 5.4 Conclusions

Based on the experiments in previous section, we can make the following conclusions.

The library is designed to process large-scale streaming graph events, therefore we evaluate the throughput of this library.

- **The library achieves linear scalability while using simple hash partition strategy.** Currently, the library can maximumly process 200,000 events per second on a single machine with 8 physical CPU cores.

- **The library has a poor scalability while using incremental hash strategy or repartition hash strategy,** because these strategies maintain a global or multiple local partition tables to reduce the latency for particular queries.

Besides processing unbounded graph events, the library also supports customized queries against the large-scale streaming graph. We evaluate the latency of different types of queries.

- **Query with hidden structure.** For queries with hidden structure, the latency depends on graph size, graph partition strategy and query computation model. The latency of such queries is mainly determined by the number of iterations. Obviously, a larger graph will lead to more iterations for a query with hidden structures, the query will also detect much more potential qualified graph patterns with more iterations. With a proper partition strategy, the query is more likely to find all necessary information in one iteration, and thus significantly reduce the latency. With the parameter-based query model, users can flexibly define their own process logic and specify additional rules to terminate the queries. However, the Cypher-like API uses a unified logic to detect all qualified graph patterns and then output what the users want. So the Cypher-like API has a higher latency than the parameter-based query model.

- **Query with certain source vertex.** For queries with certain source vertex, they will produce a limited number of scattered queries. Therefore, the latency is always low regardless of graph sizes, partition strategy or query computation model.

- **Query with uncertain source vertex.** For queries with uncertain source vertex, we have to check all existing vertices in the current streaming graph. Therefore, the latency could be extremely high when the streaming graph is huge. However, the library implements a redundancy mechanism to store some specific graph edge events bidirectionally. In this way, the query can be completed quickly with the known target vertex.

Furthermore, we make an exploration about how to combine a proper configuration for different use cases.

- As the size of streaming graph grows, the benefits of choosing a simple hash partition strategy are getting bigger and bigger, because the strategy achieves linear scalability and does not require the storage of any additional partition tables.
• The incremental hash partition strategy performs well for bread-first search. The repartition hash partition strategy is optimal for depth-first search. Apart from these two strategies, users can also design and implement their own strategies to accelerate their own customized queries.

• The Cypher-like query model provides an ease-to-use API for the users of the library. However, the API uses a uniform process logic so that no optimization space is left. Therefore, users need to decide if they use this convenient API or design an optimized process logic for their own queries.

• The redundancy mechanism can accelerate the queries with uncertain source vertex. However, there is a tradeoff between redundancy and latency. We recommend users to replicate those graph events which lead to moderate redundancy but significantly reduce the latency of particular queries.
Chapter 6

Conclusions

In this thesis, we developed a streaming graph library which supports fast and flexible user-defined queries. Contrary to most of the existing tools, our library is not specifically designed for processing a particular type of graph or any specific graph algorithms. The library provides different options of graph partition strategies and graph query abstractions. Users can choose their preferred strategy to store a streaming graph in a distributed environment, and then define customized queries accordingly.

The graph partition strategies include simple hash strategy, incremental hash strategy and repartition hash strategy. The simple hash strategy is more practical for processing large-scale streaming graph, because it can quickly determine partitioning of graph events and does not require any additional partition information to be stored for future queries. The incremental hash strategy provides a guarantee of load balancing for any type of graph or even a skewed graph stream. Furthermore, the strategy is also efficient for breadth-first search. The last repartition strategy has a poor throughput and may lead to imbalanced workload. However, it can significantly reduce the delay of some depth-first search.

Users can define customized queries by using two different query computation models. The first model is parameter-based, where users have to implement their own \texttt{Query} class which inherits the abstract \texttt{Query} class. By implementing the \texttt{parse} function, the library can extract the parameters in a query event and then formulate a \texttt{Query} object. The \texttt{scatter}, \texttt{iterate} and \texttt{gather} function decides the processing logic of a query event. There are also other abstract functions that can be implemented to enable vertex property aggregation or filtering. In this way, the library makes it flexible to customize and optimize a query. However, some users are unfamiliar with programming or feel inconvenient to implement a \texttt{Query} class for each new type of query. So we propose a Cypher-like query API, which is an experimental API built on top of the first layer query abstraction. The experimental API currently supports the basic syntax and some advanced functions (See Section 3.2). With this model, user can easily learn the necessary syntax and define their own queries with these syntaxes. This API provides a more generic way to define a query. The disadvantage of this API is that a user cannot make specific optimizations based on their own requirement because the API uses a unified logic to process all queries.

In the end, we carried out some experiments to evaluate the performance of different graph partition strategies and graph query abstractions. The evaluation results show that the library can provide a high throughput and low latency with specific configuration. The library evaluation results are in line with our expectations.

6.1 Contributions

We made several contributions. We designed and implemented a streaming graph library for Apache Flink. The library is able to execute user-defined queries against graphs representing different real-world use cases. For this reason, the library includes three typical graph partition
strategies aiming at different scenarios. Additionally, the library also provides two abstraction layers for users to define a customized query. The first layer is parameter-based and users can implement their own queries in a more flexible processing logic. The second layer is a Cypher-like query language, which provides a more generic and ease of use API. Finally, we evaluated the performance of this library and offered a couple of useful advice to the users of this library. In this way, users can combine an optimum configuration, thereby making it possible to execute queries against large-scale streaming graph.

6.2 Future work

Graph is a popular and practical structure to solve many real-world problems. Especially in the era of big data, streaming graph model becomes more and more attractive for processing large-scale graphs. Inspired by some real-world scenarios, we first developed a library for processing some typical graph queries. For example, What is shortest path between two vertices in a streaming graph?; How many vertices are connected to a certain vertex? Later, we decided to extend this library and implemented a more generic version. The generic version is an open-source project such that everyone can contribute. We list a couple of things that can be further supported in the future.

- **Support more graph partition strategies.** The library currently supports three different graph partition strategies which can optimize the graph distribution for different graph queries. It would be beneficial to support more graph partition strategies or even enable user-defined strategies. The library will become more generic, thus making it a great platform for processing different streaming graph and corresponding queries.

- **Enable more syntax in the Cypher-like query language.** To help users easily define a customized graph-related queries, we learned from Cypher, a graph query language developed for Neo4j. However, the current version is still an experimental API, which only supports several basic syntax such as select, where, return, orderBy, sum, etc. Therefore, we can further develop this graph query API and make it more powerful to define a wider range of queries.

- **Performance optimization.** We developed this library on top of Apache Flink, which is a next generation stream processing platform. Apache Flink designs an efficient and fault-tolerant memory management mechanism, which helps avoid the common OutOfMemory problem while using Java to process large-scale data. Stream processing is usually deployed in production environment and therefore imposes high demand on low latency. We still need to do a lot of engineering effort to further optimize the performance of this library.

- **Support different languages.** We developed this library using Java, but Flink is also available in Scala and Python. Scala and Python are currently two popular scripting languages in the field of data processing. To improve the ease of use, the library should also expose APIs to support Scala and Python in the future.

- **Add Checkpointing.** Every function and operator in Flink can be stateful, and therefore Flink needs to checkpoint the state. The checkpoint mechanism is crucial for the production environment deployment. Otherwise, the system will be fragile and provide no guarantee on fault tolerance. Therefore, we need to enable the checkpointing mechanism and make sure it does not have any negative influence on the performance.
Bibliography


[30] Aditi Majumder. Parallel and distributed graph algorithms and their applications. 4


---

A streaming graph library for Apache Flink


A streaming graph library for Apache Flink 51
Appendix A

How to use the library?

In this section, we write a guide about how to use this library.

We encapsulate the processing logic into a Plan class. Therefore, users need to create an instance with a parameter Tool.

```
ParameterTool parameterTool = ParameterTool.fromArgs(args);
Plan plan = new Plan(parameterTool);
```

Parameter Tool is a class provided by Flink, it can help users manage the configurations for a Flink job. The following parameters are required in our Flink execution plan.

```
-parallelism 4 //the global parallelism of the execution plan
-outputlog false //if the execution logs will be printed
-outputlinkplan true //if the Flink execution plan will be printed
-strategy hash //you can choose hash, incrementalhash or repartitionhash here
-capacity 3 //only used when using the incremental hash strategy, the parameter decides the maximum capacity for each vertex on each partition
-haveFriend friendOf //only added when you want to replicate the edges labeled as haveFriend and generate a reverse edge labeled as friendOf. You can add as many pairs as you want.
```

After creating the plan, we need to set the data source first. We need to set both the graph stream source and the query stream source. For example, we can use Kafka consumer as the data source.

```
FlinkKafkaConsumer010<String> queryStream =
  new FlinkKafkaConsumer010<> (java.util.regex.Pattern.compile("query"), new SimpleStringSchema(), properties);
FlinkKafkaConsumer010<String> graphStream =
  new FlinkKafkaConsumer010<> (java.util.regex.Pattern.compile("graph"), new SimpleStringSchema(), properties);
plan.registerGraphSource(graphStream);
plan.registerQuerySource(queryStream);
```

After setting the data source, we need to assign parsers to parse the events from data source. Users can implement the parsers by themselves or use the provided parsers. If the query parser is set as CypherLikeQueryParser, user can use our Cypher-like Query API.

```
plan.registerGraphParser(new GraphParser());
plan.registerQueryParser(new ShortestPathQueryParser());
```

In the end, you can launch the execution plan.

```
plan.run();
```
Appendix B

Queries used in benchmark

All these queries are written using the syntax of Cypher-like Query API. Due to confidential reason, we replace all vertex and edge labels with VertexLabel and EdgeLabel.

• Query 1: Is there a shortest path between vertex v1 and v2 that the length of the path is no more than 4?

```
select (v1:VertexLabel) -[:EdgeLabel|0..4]- (v3:VertexLabel) -[:EdgeLabel]- (v2:VertexLabel)
where v1.id = 1 and v2.id = 2
orderBy count(?a3) ASC
return a1.id, concat(?a3.id), a2.id
```

• Query 2: How many distinct vertices that vertex v1 is connected to?

```
select (v1:VertexLabel) -[t:EdgeLabel]- (v2:VertexLabel)
where v1.id = 2
return count(distinct v2)
```

• Query 3: How many distinct vertices are connected to vertex v2?

```
select (v1:VertexLabel) -[t:EdgeLabel]- (v2:VertexLabel)
where v2.id = 2
return count(distinct v1)
```

• Query 1-2: Is there a path between vertex v1 and v2 that the length of the path is less than 4?

```
ask (v1:VertexLabel) -[:EdgeLabel][0..4]- (:VertexLabel) -[:EdgeLabel]- (v2:VertexLabel)
where v1.id = 1 and v2.id = 2
```

• Query 1-3: What are the paths between vertex v1 and v2 that the length of the path is less than 4?

```
select (v1:VertexLabel) -[:EdgeLabel][0..4]- (?v3:VertexLabel) -[:EdgeLabel]- (v2:VertexLabel)
where v1.id = 1 and v2.id = 2
return v1.id, concat(?v3.id), v2.id
```

Hints: all vertex IDs in where statement are replaced with random numbers while doing the benchmark.