GraphX Query System

Ahmadreza Jeddi
David R. Cherition School of Computer Science, University of Waterloo
Waterloo, Ontario
a2jeddi@uwaterloo.ca

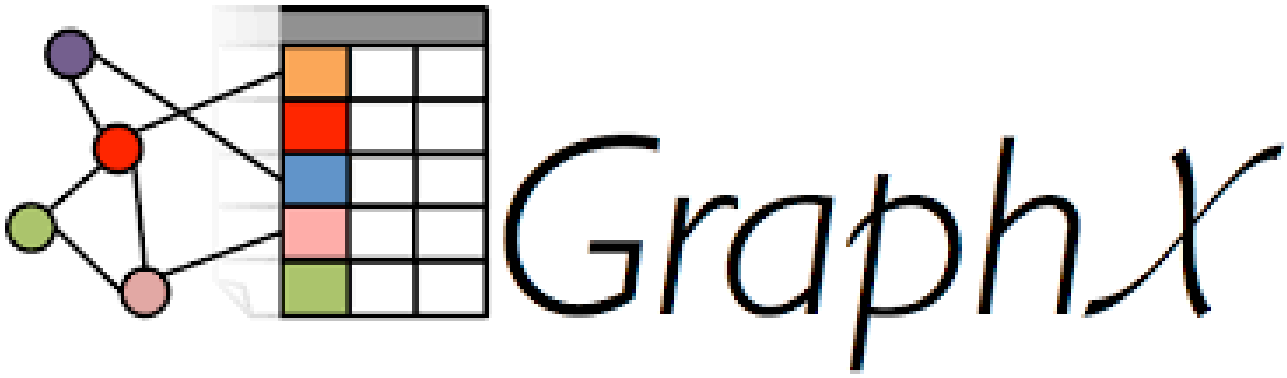

Figure 1: GraphX: a Spark component for graph-parallel computation.

ABSTRACT

Graph-structured data analytics is now very vital for many applications such as Web graph page ranking, social network link analysis and prediction, and language modeling. The increasing volume of such graph data has sparked the need for special large-scale graph processing frameworks; as a result, in the past decade numerous such systems have been developed. Spark GraphX is one of the most promising systems developed to meet these graph processing needs. By providing a rich high-level Spark API and utilizing the Hadoop ecosystem components, GraphX offers operators and implemented algorithms that can be used to solve the graph-related problems. In this article, I go a bit under the surface of the GraphX interface, and describe its key features; utilizing these features, I propose and implement a novel ranking-based graph query system that runs SQL-like queries on the nodes of a large-scale directed graph and ranks the selected nodes based on graph specific metrics and reports the results to the user. This method can help a user of the graph, find other nodes that fulfill their search preferences; this ranked list of found nodes, could be used for applications like recommendation systems, and connection suggestions on social networks. A video accompanying this article would demonstrate such an example.

CS-651, Winter 2020, University of Waterloo, Waterloo, Ontario

CCS CONCEPTS

• **Information systems → Data management systems**; *MapReduce languages*; Graph-parallel distributed computation.

KEYWORDS

big data, Hadoop, Spark, distributed computing, MapReduce, query system, graph analytics

ACM Reference Format:
Ahmadreza Jeddi. 2020. GraphX Query System. In *(CS-651).* 6 pages.

1 INTRODUCTION

The new century has seen rapidly increasing data volumes, providing large scales of different types of data, so we live in the Big Data era. Fortunately, a large body of work has been done by numerous researchers and engineers to meet the requirements of large-scale data processing [9, 12, 17, 18]. Distributed computing by using clusters of commodity servers is the most common approach to tackle this challenge [11]. MapReduce [7], a distributed computing programming model, lies in the heart of almost every big data processing framework and algorithm. Hadoop [1, 14] was one of the first and promising solutions for the need of better computing tools, and the different aspects of Hadoop framework, like the distributed file system, the resource managers, and the scheduling algorithms form the very basis of almost any other framework in this field, like Spark [3, 18].

Although Spark is a cluster computing framework on its own, big data practitioners usually consider it as a part of the Hadoop ecosystem, as Spark continues to borrow many of Hadoop's subsytems, especially the Hadoop Distributed File System (HDFS). Furthermore, both of these frameworks have various computing libraries for different sorts of computing domains. Detailed discussion about these libraries and subsystems is beyond the scope of this article. Reader can refer to [5] for more informations. **Q1:** considering these facts, which one is a better framework, Hadoop or Spark?

In order to be able to answer Q1, I would like to rephrase it like this: which one is better, Hadoop's MapReduce or Spark's RDD (Resilient Distributed Dataset) based actions and transactions? In my opinion Spark is the better option, as its in memory processing makes it much more efficient, and it is more flexible, and provides more options than the simple map and reduce.

RDDs are collections of elements partitioned across the nodes of the cluster and can be operated on in parallel. RDDs and libraries that can manage their processing and storage form the core of Spark, and every Spark application consists of a driver program that runs the user's main function and executes various parallel operations on the RDDs. In addition to the core library, Spark distributions have an increasing body of built-in libraries accompanying the core library. These libraries include SQL, MLlib, Streaming, and GraphX. They extend the RDD concept into various applications, and provide flexible interfaces for distributed large-scale processing.

From here on, the focus of this article would be on the GraphX library of the Spark framework. The reason I choose GraphX is that I am very interested in graphical modeling, and statistical inference within a graph structured dataset. During past few years, I have done research on various graph related topics, like, belief propagation and message passing algorithms within graphs, probabilistic graphical models and the application of conditional random fields and graph cuts within the domain of semantic image segmentation, and representation learning of graph nodes and edges for link prediction in dynamic graphs by using node2vec framework [8]. While working on the dynamic node2vec project, I worked with the GraphX library components; for more details on this project, reader can refer to this github repository.

GraphX extends the Spark RDD abstraction to graphs and graph-parallel computation. It works with both collections and graphs, and supports the graph ETL (Extract, Transform, and Load), so graphs can be generated from various types of sources. Furthermore, it has implementations for some well-known graph algorithms like PageRank and connected components, and provides exploratory analysis tools such as transforming and joining different graphs and various views of graph elements via RDDs. All of these features are explained in detail in section 2.2.

Utilizing the features of Spark and GraphX, I propose a graph query system, in which a user can run queries over a graph. For example, consider the graph of users on LinkedIn: each node of this graph represents a user with their profile information (in this case public information that is available to other users like their research or work interests or the institute of their education), and edges represent the connections between the users (or endorsements). A user in this graph might like to find other users and make connections with them, based on their preferences. For instance, they might like to find users among their own connections and connections of their connections who have a certain set of skills and level of education; my proposed graph query system is able to perform such queries on large scale graphs, and report the results based on a ranking algorithm, and this ranking ranking can be calculated based on different attributes, like the distance to the user, user's personalized PageRank, and other such measures. To the best of author's knowledge there is no such framework for providing SQL like queries on large-scale graphs. Moreover, the author cannot verify whether graph based query is feasible for very large graphs in practice. To this end, my contributions can be folded as below:

- A detailed discussion about various features of GraphX library, including the class structure and overview of its variables and methods, especially its more important operators, as well as the Pregel system, and its implemented algorithms
- A novel graph query system, which can perform SQL-like queries on property graphs by using GraphX and Spark available features.

The article is organized as follows: section 2 provides a a detailed discussion about GraphX; the proposed graph query system is presented in Section 3, and followed by a conclusion.

2 GRAPHX

GraphX introduces Resilient Distributed Property Graph as an extension to Spark RDD abstraction. A Property Graph refers to a directed multigraph in which vertices and edges have properties. Multigraph refers to having parallel edges [2]. Figure 2 shows such a graph; this graph is a *Graph[(String, Int), Int]* property graph. In this section, I will first present a brief background on graph parallel computation, and after that I will continue with different types of GraphX operators, and at the end of this section, I will discuss Pregel API [12], a system developed for iterative parallel computation.

2.1 Graph parallel computation background

Social networks [15], academic citations [16], graph of web pages and links [13], and language modeling are just few examples of the growing scale and importance of the graph structured data that has driven the development of numerous graph parallel systems. Based on the specific graph related goal of such a system (managing databases or graph analytics), these systems introduce new graph partitioning techniques, and restrict their computation types; as a result, these graph parallel systems execute sophisticated graph algorithms much more efficient and faster than other data parallel ones [4].

In [6], Ammar and Ozsu compare 8 different graph parallel distributed computing systems under 4 graph algorithm (PageRank, weakly connected components, single source shortest path, and K-hop, another version of shortest path but with K differnt sources simultaneously), and they provide exhaustive evaluations on 4 very large graph datasets (Twitter, WRN, ClueWeb, and UK200750). According to the results, they report the pros and cons of each system. From this very detailed study, here, only table 1 is reported which shows the key characteristics of each of these systems. Detailed analysis of the experiments in [6] is beyond the scope of this article; however, here, I report their findings about GraphX:

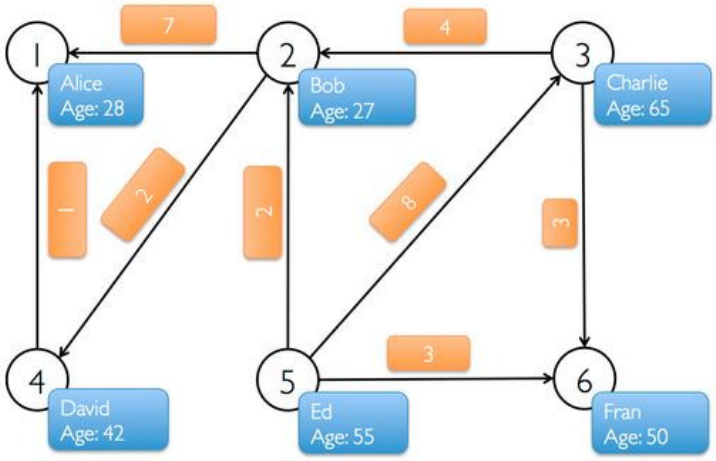

Figure 2: A simple small social network property graph; nodes have a (String, Int) property pair representing the name and the age of a user, and edges have an Int property showing the number of times a user has liked another user's post [2]

- GraphX is not suitable for graph workloads or datasets that require large number of iterations.
- General data parallelization frameworks such as Hadoop and Spark have additional computation overhead that carry over to their graph systems (HaLoop, GraphX). However, they could be useful when processing large graphs over shared resources or when resources are limited.
- GraphX suffers from Spark overheads, such as data shuffling, long RDD lineages, and checkpointing.

As Table 1 shows, graph partitioning in GraphX is done with vertex cut. Figure 3, shows vertex cut and edge cut; in the former, edges are assigned to nodes and vertices can span multiple machines, and in the later, the opposite happens, with vertices assigned to nodes and edges spanning (two) machines. Figure 4 shows a realization of the vertex cut graph partitioning done on a simple property graph. The partitioning algorithm assigns each edge to a node, and after that, the challenge is routing vertex attributes and joining them with the edges.

2.2 GraphX operators

At this point, GraphX is only available in scala language, and this github repository contains all of the GraphX classes and objects. Among them, the Graph abstract class and its realization (Graph object) are the main point of entry to the GraphX, and together with the GraphOps class, they provide all of the GraphX operations. Every property graph modeled with a GraphX Graph object contains two very important RDDs: EdgeRDD[ED] and VertexRDD[VD]. ED and VD show the types of properties the edges and the vertices have, respectively. Together, these two RDDs, form a Graph[VD, ED] object. For example, the property graph illustrated in figure 2 is *Graph[(String, Int) Int]*, which means property of vertices is of the *(String, Int)* Tuple type, while edges have an *Int* value as their property. Some of the GraphX operators like, fromEdgeTuples, fromEdges, and the Graph constructor, provide the means to create Graph objects. Once a Graph[VD, ED] object is created, we can have triplets RDD as RDD[EdgeTriplet[VD, ED]]; what this does is to merge properties of each edge and the pair of vertices it connects, into an EdgeTriplet object which has both the vertices property (VD) and the edge property (ED) into a single object. Figure 5 shows these three types of RDDs for a GraphX graph object. Any other GraphX operator and algorithm (e.g. PageRank) works by applying some Spark transactions or actions on these three RDDs: vertices: VertexRDD[VD], edges: EdgeRDD[Edge[ED]], and triplets: RDD[EdgeTriplet[VD, ED]].

The operators can be divided into these categories:

- **Basic grpah information:** methods like *numEdges*, and *numVertices* that return the size of each set, or methods like *degrees*, *inDegrees*, and *outDegrees* that return VertexRDDs.
- **New graph generators:** These operators return new graphs. Some of them change the structure of the graph; *reverse*, *subgraph*, and *groupEdges* are of this type. Others like *mapVertices*, *mapEdges*, *outerJoinVertices*, and *mapReduceTriplets* keep the same graph structure, but modify the properties of either edges or vertices. Join operator are the key to merge the information from different graphs into a single graph, so for example, we can merge the result of the PageRank algorithm (which structurally is the same graph, with only PageRank value as the vertex property), into the graph itself. They are also very important for my graph query system.
- **Graph algorithms:** the growing library of GraphX now has implementations of these algorithms: PageRank, shortest path, connected components, label propagation, triangle count, and SVD++. The result of these is another graph with the same structure as the original one, with vertex or edge properties modified.
- **Pregel:** a system that simplifies the iterative graph computational algorithms like PageRank and shortest path. This is done via formulating the algorithm like a message passing problem. More details about the Pregel is provided in the next subsection.

2.3 Pregel

In [12], Malewicz *et. al* propose Pregel and describe it as a system for large-scale graph processing. Pregel is a bulk-synchronous

System	Memory/Disk	Architecture	Computing paradigm	Declarative Language	Partitioning	Synchronization	Fault Tolerance
Hadoop	Disk	Parallel	BSP	×	Random	Synchronous	re-execution
HaLoop	Disk	Parallel	BSP-extension	×	Random	Synchronous	re-execution
Pregel/Giraph/GPS	Memory	Parallel	Vertex-Centric	×	Random	Synchronous	global checkpoint
GraphLab	Memory	Parallel	Vertex-Centric	×	Random Vertex-cut	(A)Synchronous	global checkpoint
Spark/GraphX	Memory/Disk	Parallel	BSP-extension	×	Random Vertex-cut	Synchronous	global checkpoint
Giraph++	Memory	Parallel	Block-Centric	×	METIS	(A)Synchronous	global checkpoint
Blogel	Memory	Parallel	Block-Centric	×	Voronoi 2D	Synchronous	global checkpoint
Vertica	Disk	Parallel	Relational	✓(SQL)	Random	Synchronous	N/A

Table 1: Graph processing systems and their characteristics [6]

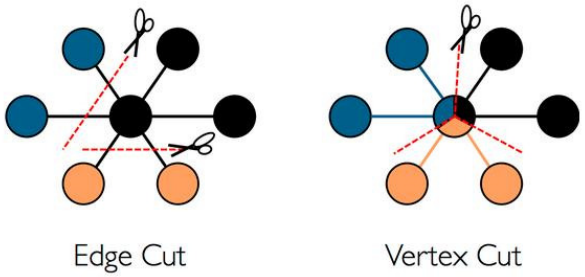

Figure 3: Graph partitioning for distributed computing [4].

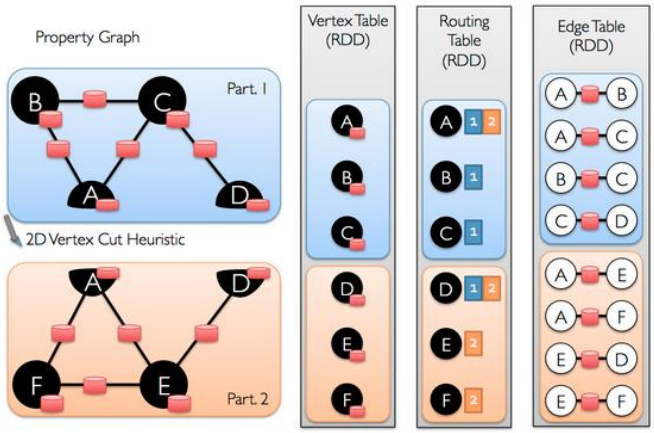

Figure 4: Example of GraphX partitioning a property graph using the vertex cut strategy [4].

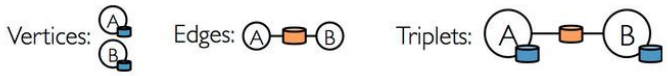

Figure 5: Three most important RDDs of every graph object (from left to right): VertexRDD (vertex property), EdgeRDD (edge property), and RDD[EdgeTriplet] (joining edge and vertex property for each edge) [4].

message-passing system, which operates over a sequence of iterations, called supersteps. In each superstep, S_i, the framework evokes a user defined function on each vertex (in parallel), which consist of 3 parts: 1) **message combining:** defines how to deal with the received messages. For example, PageRank computes the sum of the received messages which are the r_js sent to this node, and shortest path computes the minimum of the received messages. 2) **Update the node's property:** based on the received messages and combined result and the current state, update the property values. For PageRank, it means to run the PageRank equation, and for the shortest path to choose the minimum between the current property value and the minimum of the received messages computed by the combine method. 3) **message sending:** send messages through outgoing edges, based on the defined message protocol (*activeDirection*). based on the protocol, a vertex may or may not send a message at superstep S_i. The messages sent in this superstep, will be processed in superstep S_{i+1}; for example, in PageRank, each node j sends a $\frac{r_j}{n_j}$ message via its outgoing edges, where r_j and n_j show the node's page rank value and number of outgoing edges, respectively, and in shortest path, each node sends its own path length + 1 via its outgoing edges.

As described in the previous paragraph, every Pregel operation needs message sending, messing combining, and node program (how to update the node properties). Other than these functions, we should also define an initial message and a termination criterion. the Initial message is the one that each vertex receives at the

first superstep. For PageRank, the initial message is 0, and for the shortest path, the initial message is $+\infty$. The termination criterion can be either a specific iteration number or checking a state in the supersteps. In [12], authors terminate the process when all vertices are inactive. A vertex gets inactive when it has no work, i.e. if it has nothing updated and no messages to be sent. If an inactive vertex receives a message, it processes the message, and if its state changes, it sends messages through its outgoing edges; after this, it should explicitly deactivate itself. At the first superstep, all vertices are active.

3 GRAPH QUERY SYSTEM

In this section, I will describe my proposed graph query system. Let's assume that we are given a large-scale graph, and a node in this graph wants to perform a query on the rest of the graph, and get a ranked list of the other nodes, based on his/her query specifications. A very good example for such a scenario is to consider a social network platform like LinkedIn; where a user might look for other individuals who have a specific set of skills or a specific background such as their field of study, and the institute of their studies. Such a user may have other graph specific preferences as well; for example they may want the closer nodes (in terms of shortest path) have higher ranks. In other cases, they may want to use node's PageRank value; so more important users may get higher ranks. GraphX gives a graph abstraction to the table-like data collections (RDDs), and provides operators and optimizations, so graph-specific features like PageRank and shortest path can efficiently be computed; these graph-specific features are central to the graph query system.

There are two main components for the query system; the first one is the BuildGraph object, which receives a file containing graph edges in the form of source node and destination node pairs. This object creates a property graph RDD and saves this graph into the Hadoop's file system. More details on how to create the property grah will be elaborated in the experiment subsection. The second component of the query system is the querying part, which takes the query preferences and the node which runs the query. The result of the process is the ranked list of the nodes.

3.1 Experiment

In order to demonstrate the effectiveness of the proposed system, here, I provide a toy example: for the graph, I choose Wikipedia Talk Network [10]. In this graph, nodes show the Wikipedia registered users, and an edge from a node i to another node j means that user i has edited a talk page of user j at least once. It has 2,394,385 nodes and 5,021,410 edges. However, in this toy example, I only use the structure of this graph; the final property graph that I create has *Graph[User, Double]* as its signature, which means its vertices's property is *User* and its edges's property is *Double*. *User* is a case class that I define, and for this example it has two fields: *age* and *interests*, respectively showing the age of the user which can be anything between 18 and 75, and a list of academic interests that are randomly chosen from a list of 15 computer science related research fields. A user can have 2 to 5 interests, randomly chosen. While creating the graph from the edge file, for every new node, an User instance is created and then, the node added to vertex RDD. The edges's property is a *Double* value. If the graph is weighted, edges take the weight value as their property, otherwise, it is simply put to 1.0, meaning unweighted distance between the nodes.

Figure 6 shows the graphical user interface for such a scenario. The upper half in this figure provides the interface to build a graph; providing the path to the edges file, which in this example is the wiki talk data, and path to save the graph's edges and vertices, so we can load it later to perform the queries. There are two other options: weighted graph and undirected graph. The former is for when the graph is weighted and the later for when the graph is undirected. When the *build button* in figure 6 is clicked, the following command is executed:

```
$ spark-submit --driver-memory 12g --class
↪  ca.uwaterloo.cs451.project.BuildGraph
↪  target/assignments-1.0.jar --input
↪  data/wikiTalk.txt --output graph1 --weighted
↪ --undirected
```

The lower half shows the interface for performing the query. A path to load the graph and a node to run the query from. And at the end, we can select the range of user age that we are interested in, and if we would like to use PageRank as an evaluation metric. When the *search button* in 6 is clicked, this command is executed:

```
$ spark-submit --driver-memory 12g --class
↪  ca.uwaterloo.cs451.project.GraphQuery
↪  target/assignments-1.0.jar --input graph1
↪ --srcnode 2 -- agerange 30-50 --pagerank
```

Assuming that we select to have PageRank in the process, in this toy scenario, the shortest path to the search node and PageRank algorithms are run and the values are computed for every node; these values are then joined to the User graph, the nodes are then filtered according to the given *age* limit, and finally each node's score is calculated based on the following function:

$$score(v) = exp(i + p - d) \tag{1}$$

where for a given node v, i shows how much node v covers the interests of the start node, so $i = \frac{|intersection(v,start-node)|}{|start-node|}$ (cardinality refers to the size of the interests). p and d show PageRank value and distance to the start node, respectively. PageRank values are all normalized to be in $[0, 1]$ range. If PageRank option is not selected then p is equal to 0.

After the node scores are computed, the nodes are sorted based on these scores and those with higher scores get higher ranks. At the end of the process, 50 top nodes are reported; this can be seen at the bottom of the figure 6.

4 CONCLUSION

In this article, I describe the GraphX library of the Spark framework; a brief background on graph-parallel compuatation, then GraphX components, as well as the Pregel system and its usage within the GraphX library. And finally I implement graph query system, a novel method to run queries on large-scale property graphs and rank the graph nodes based on query preferences and graph specific metrics like PageRank and distance between the nodes. This system can be used for node rankings.

REFERENCES

[1] 2006. *Apache Hadoop.* http://hadoop.apache.org/

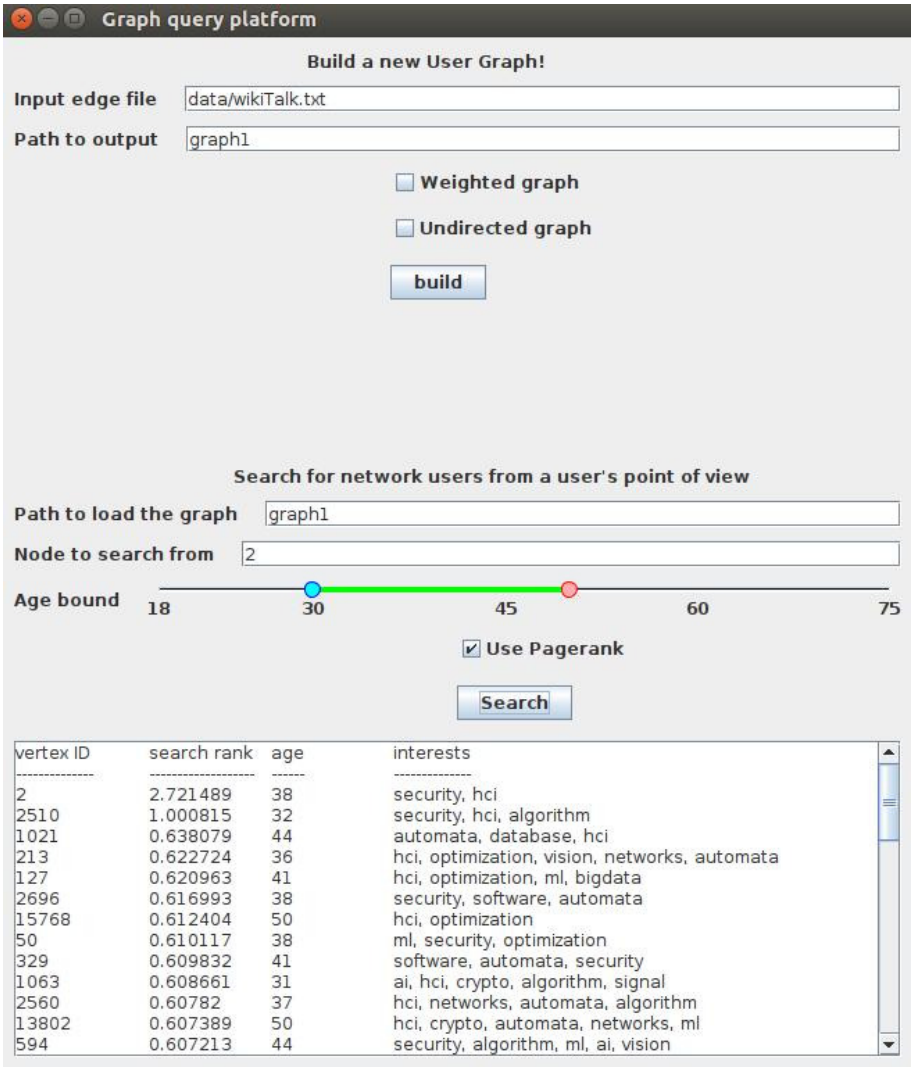

Figure 6: The graphical user interface that I have developed to give a better experience of graph query system.

[2] 2009. *AMP lab documentation for GraphX.* http://ampcamp.berkeley.edu/big-data-mini-course/graph-analytics-with-graphx.html

[3] 2009. *Apache Spark.* http://spark.apache.org/

[4] 2020. *AMP lab documentation for GraphX.* https://spark.apache.org/docs/latest/graphx-programming-guide.html

[5] 2020. *Hadoop vs Spark.* https://data-flair.training/blogs/spark-vs-hadoop-mapreduce/

[6] Khaled Ammar and M Tamer Özsu. 2018. Experimental analysis of distributed graph systems. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1151–1164.

[7] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[8] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 855–864.

[9] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center.. In *NSDI*, Vol. 11. 22–22.

[10] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Signed networks in social media. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 1361–1370.

[11] Jimmy Lin and Chris Dyer. 2010. Data-intensive text processing with MapReduce. *Synthesis Lectures on Human Language Technologies* 3, 1 (2010), 1–177.

[12] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[13] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The pagerank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[14] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 1–10.

[15] Wei Tan, M Brian Blake, Iman Saleh, and Schahram Dustdar. 2013. Social-network-sourced big data analytics. *IEEE Internet Computing* 17, 5 (2013), 62–69.

[16] Feng Xia, Wei Wang, Teshome Megersa Bekele, and Huan Liu. 2017. Big scholarly data: A survey. *IEEE Transactions on Big Data* 3, 1 (2017), 18–35.

[17] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 15–28.

[18] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.