GraphFlow: Workflow-based Big Graph Processing

Sara Riazi, Boyana Norris
Department of Information and Computer Science
University of Oregon
Eugene,OR 97403
sara@cs.uoregon.edu, norris@cs.uoregon.edu

***Abstract*—We introduce GraphFlow, a big graph framework that is able to encode complex data science experiments as a set of high-level workflows. GraphFlow combines the Spark big data processing platform and the Galaxy workflow management system to offer a set of components for graph processing using a novel interaction model for creating and using complex workflows. GraphFlow contributes an easy-to-use interface and scalable algorithms for big graph analytics. We demonstrate GraphFlow use in large social network analysis with several case studies.**

***Keywords*-graph analysis, workflow, big data**

I. Introduction

In recent years, analysis of large-scale networks has become an integral research component in a wide variety disciplines including bioinformatics [1], social sciences [2], and epidemiology [3]. Networks are graph-based models of complex systems of interacting entities. The entities are represented as vertices in the network and their pairwise interactions as edges. Analyzing the properties of networks helps us understand the characteristics of the underlying systems. For example, vertices with high centrality map to lethal proteins in protein-protein interaction networks [4], and groups of closely connected vertices in social networks map to people with similar interests [5].

Unfortunately, in practice there is a significant gap between the services provided by the network analysis toolkits and the actual needs of the domain experts. Most network analysis packages only provide a set of algorithms that can be used as a black box. However, with the increasing diversity of data formats and solution requirements, there are no high-level reusable solution approaches. Instead, each data analysis instance can have a different workflow based on the underlying analysis framework, typically requiring domain expert involvement at each step. Current network analysis frameworks do not provide sufficient support for creating, reusing, and extending complex workflows required for analyzing large diverse datasets. Moreover, they rarely provide support for auxiliary, but necessary tasks, such as creating graphs from raw data, filtering metadata, selecting heuristics and comparing multiple results.

Processing large datasets presents yet another challenge. Very few network analysis software tools support parallel algorithms, and the set of available methods is also small. Some approaches even implement out of core algorithms [6], [7] to enable the analysis of large-scale graphs, incurring significant performance penalty for disk I/O (even when using SSDs). Even though any tasks, such as finding the correlation coefficients over a set of entities or running multiple algorithms over the same dataset, are trivially parallel, most software tools do not allow simultaneous execution of these tasks, even when there are parallel resources available. Our approach aims to maximize parallel resource utilization by exploiting parallelism at a much finer-grained level, within individual analysis tasks.

The research contributions described in this paper is a new framework for large-scale graph analysis that combines the Galaxy workflow engine and interface with a new distributed Spark-based implementation, including the following.

- A Galaxy-based workflow front end to Spark-based distributed graph analysis tools.
- High-level abstractions for Spark components and a well-defined interaction model.
- Examples of reusable GraphFlow workflows.

We illustrate GraphFlow capabilities with realistic large-scale graph analysis use cases.

II. Background

In this section we overview the concepts and software infrastructure, on which GraphFlow is based, as well as related work on parallelizing Galaxy workflows.

A. Large-Scale Graph Processing Approaches

One of the most significant advances in distributed data processing is the MapReduce programming model [8]. In MapReduce, data is converted to key-value pairs and then partitioned to nodes. A MapReduce system consists of a set of workers that are coordinated by a master process. Computation consists of multiple distributed map and reduce operations, with intermediate results stored as key-value pairs stored on local disks.

Many graph frameworks such as Preglix [9], GraphX [10], and PEGASUS [11] are developed on top of data-parallel systems to benefit from their optimized parallel processing. The other major benefits of using data-parallel system to represent graph data structure and graph operation are graph mutation and out-of-core computation.

GraphX [10] is a distributed graph processing framework developed on top of Apache Spark[1], which is a fast growing distributed computing framework. In Spark applications, a driver "main" program runs on a master process and coordinates the execution of distributed worker processes. The most important concept in Spark is the resilient distributed dataset (RDD) data structure [12], an immutable collection of objects that is partitioned across different Spark workers in the network. Because Spark is a data-parallel computation system, GraphX implements graph operations based on data-parallel operations available in Spark such as join, map, and various reductions. GraphX represents graphs using two RDDs, one for vertices and another for edges.

However, efficient processing of graphs in a distributed environment requires more than simple MapReduce operations because vertices are processed in the context of their neighbors. Hence, GraphX represents vertices and edges using a triplet construct that stores the value of an edge and the values of vertices incident to that edge. Therefore, by grouping triplets on id of the source or destination vertex, one can access the value of all the neighbors of each vertex. Moreover, because the triplets are distributed, if the neighbors of a vertex are located on different machines, then Spark workers communicate to each other to construct the group by result.

B. Galaxy Workflow

Galaxy [13] is a public, user-friendly data integration and workflow management system, mostly used in the biomedical sciences. Recently, it also has been used for social sciences [14], but the provided tools are based on common statistical algorithms, and are not intended for big data processing. Galaxy does not have any abstraction for graph data. Galaxy enables users to graphically describe workflows by drawing pipes to connect tools. Then, the system manages the execution of the workflow by running the tools over input datasets in a defined order. The Galaxy *toolshed* supplies different, mostly bioinformatics tools, for gathering and processing data, and also allows users to introduce new tools and include them in the workflows. Galaxy workflows can be stored, which enables reproducibility. Galaxy runs each tool as a separate sequential program providing the output of a tool as the input to the next tool in the workflow.

Galaxy tools are described using XML files that contain the input, parameters, output of the tools, and execution specifications.

Many bioinformatics applications are data intensive; hence, running the Galaxy workflows in the cloud can increase the amount of available resources and can also potentially speed up the evaluation of workflows. In a cloud-based environment, tools in a workflow that do not depend on each other run in parallel on different machines [15]. In these platforms Galaxy is offered as service on purchasable high-performance computational resources.

BioBlend [16] is another approach to parallelizing Galaxy workflows, which offers a rich API for accessing Galaxy workflows and jobs on cloud resources. Nevertheless, these platform and frameworks do not parallelize the execution of each tool (algorithm) over different machines, so many data science experiments that are usually expressed as a pipeline of single tools and scripts would not benefit from these cloud-based coarse-grained parallelization approaches.

III. GraphFlow

In this section, we introduce GraphFlow, a workflow-based big graph processing toolkit. The GraphFlow toolkit is a set of new Galaxy compatible tools that offers the rich GraphX graph algorithm API through the higher level of abstraction of Galaxy workflows, which improves usability, reuse, and reproducibility of graph analysis tasks, while adding fine-grained parallelism to Galaxy for the first time.

Using GraphFlow we can construct complex data science experiments as a workflow of Spark-based components. Although throughout this paper we focus on Spark as the data-processing engine, we can incorporate other data-processing frameworks in future.

Figure 1 shows the general architecture of GraphFlow. Each new Galaxy tool submits a Spark application to cluster systems through the cluster-adapter or runs it on a local machine. The cluster-adapter we developed provides access to tools by managing the Spark master node and other cluster dependent configuration. This new architecture enables GraphFlow to separate the workflow interface from the data processing. Therefore, Galaxy workflow can be placed on a local machine, e.g., a laptop, while the data engine resides on the cluster system.

A. Data Description

The input data provided by Galaxy must be made accessible to Spark applications and output data generated by Spark applications must be accessible to Galaxy. The new cluster-adapter is responsible for this data migration.

In addition, Galaxy expects the input data to be stored as single local file in a conventional file system (not a distributed file system such as HDFS). By contrast, Spark partitions data into multiple files, which may also be distributed over many separate machines (virtual or real).

To address this inconsistency in a MapReduce context, Pireddu et al. [17] introduce a new functional and extensible integration layer, which enables the users of Galaxy to combine Hadoop-based tools with conventional sequential tools in their workflows.

Their adaptation layer combines the HDFS address of input data files as a *pathset* and passes the constructed pathset to a Hadoop-based tool, which outputs another

[1] https://spark.apache.org/

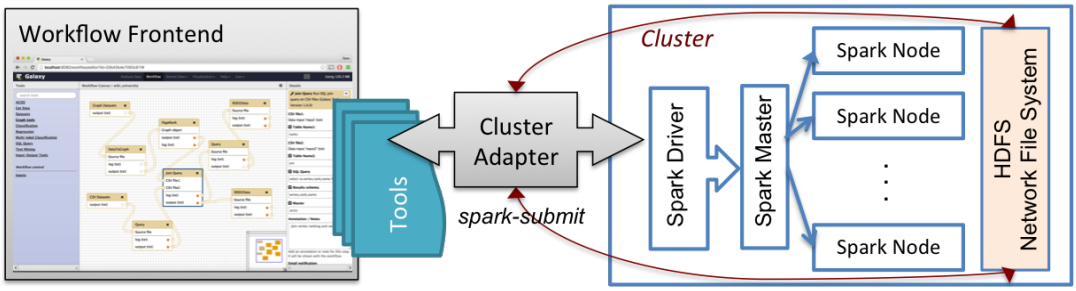

Figure 1. The architecture of GraphFlow. The Spark-based tools in Galaxy interact with Spark nodes on the cluster system using a cluster-adapter.

pathset as results. The output pathset can be the input of another Hadoop-based tool.

We build on this indirect referencing and introduce the *Metafile* as the input and output format of GraphFlow components. A Metafile is an XML description of the objects, the address type, and object address. By using the information about the address type, the cluster-adaptor can determine whether the object is stored locally, on HDFS, or on a network file system, and can then post the application to the requested cluster system or local machines if the data is available to it. Moreover, to avoid data migration, the address type is used for allocating space for the output data at the same file server as the input data.

Metafiles also include the schema of the data, which helps users attain general understanding about the underlying values because only Metafiles are accessible to users through the Galaxy experiment history.

B. Interaction Model

The ultimate goal of GraphFlow is to provide a workflow-based environment that is capable of encoding complex graph analytic experiments. Each GraphFlow component is a Spark application that manipulates a distributed collection of objects stored as dataframes. By using this representation, we define each GraphFlow component as either: (a) a complex map function that transforms a dataframe or a graph object to another dataframe, graph or a combination of these; or (b) a reduce function of a dataframe or a graph into a single data file, a set of aggregated values or charts.

Loading and storing typed collection objects such as RDDs reduces the generalization of the each component because RDDs have to be manipulated differently based on the type of the objects they are encapsulating. For better generality, each GraphFlow component expects the input to be in a named column format, such as a CSV file. Each GraphFlow component loads the input CSV file into a dataframe and maps it to another dataframe. Finally, the component stores the dataframe as another CSV file. The CSV files are multi-part files, so GraphFlow components expect a Metafile as input that contains the schema of these CSV files and their addresses, and outputs another Metafile. The schema of an output Metafile may be different from the schema of the input Metafile. We use the Spark-CSV library[2] for I/O of dataframes.

C. GraphFlow Components

The GraphFlow components are grouped into general input/output tools, graph analytic tools, relational tools, and plotting tools. All GraphFlow components return a log file in addition to their expected output. This log output is a single text file understandable by Galaxy. The log files usually includes a small sample of output dataframes and the execution log of the tool (useful for debugging). For simplicity, we do not explicitly mention the log output in the description of each tool. Next, we describe GraphFlow components in more detail.

GraphFlow's **I/O tools** include components used to convert single-file data into dataframes and graph objects, and also to convert them back to single-file data. Since the aim of GraphFlow is to process big graph data, we expect GraphFlow's users to upload their big data files directly to the cloud storage (e.g., Amazon S3 if using Amazon AWS) instead of uploading through the Galaxy Web interface, and use their corresponding Metafile of their data as input to the GraphFlow's components. Therefore, we provide a basic *MetaLoader* component, which takes the file information from users and constructs a Metafile for it. The *MetaLoader* component can be used as the initial component of any workflow. *DFDump* can be used for converting a distributed dataframe back to a single file, which is downloadable through Galaxy interface. GraphFlow has two more similar components *GraphLoader* and *GraphDump* for loading a distributed graph object from a single file and for dumping a graph object into a single file, respectively.

GraphFlow provides a collection of **graph tools** that include algorithms for generating and processing big graphs: *GraphGen*, *PageRank*, *DegreeCount*, *TriangleCount*, *Subgraph*, *LargestCC*, *GraphCluster*, *ClusterEval*, and *GraphCoarsen*. The *GraphGen* components support generating random graphs using log-normal degree distribution and RMAT [18].

[2]https://github.com/databricks/spark-csv

PageRank is a well-known graph vertex ranking algorithm introduced by Google for ranking Web pages. GraphFlow's *PageRank* component takes a graph object and outputs a dataframe with two columns of vertex IDs and rank value, for which the rank values are computed using the PageRank algorithm provided by Apache Spark's GraphX library.

Similar to the *PageRank* component, the *DegreeCount* and *TriangleCount* components take a graph object and return a dataframe of vertex IDs and degree counts, and a dataframe of vertex IDs and triangle counts, respectively.

The subgraph function in GraphX constructs a subgraph of the original graph. The user must provide either an edge or a vertex indicator function. The purpose of the indicator function is to determine whether the given edge or vertex belongs to the resulting subgraph. In order to utilize the indicator function, we represent any discrete function f as a dataframe of x and $f(x)$. For the vertex indicator, x is a vertex ID, and f is a boolean function. The *Subgraph* component in GraphFlow takes a graph object and a dataframe representing an indicator function, and returns two graph objects: one for the subgraph corresponding to the indicator function and the other for the complement of that.

Another component is *LargestCC*, which takes a graph object and outputs the subgraph of its largest connected components.

GraphFlow also includes a set of graph clustering algorithms such as PIC [19], spectral clustering [20], and label propagation. We rely on the Spark implementation for PIC and label propagation, and add our implementation for spectral clustering. For *GraphCluster* takes a graph as its input and returns two outputs. The first output is a graph object called a cluster graph, in which the attribute of each vertex is the cluster number of that vertex. The other output of *GraphCluster* is a dataframe of vertex IDs and cluster numbers, which can be transform to an indicator function using query component as we describe later, so one can easily create a subgraph of nodes belonging to a particular cluster.

To measure the quality of a clustering, we created a GraphFlow *ClusterEval* component that implements two clustering metrics, modularity [21] and normalized cut [22]. This tool takes a cluster graph (as described above) and computes the modularity and normalized cut. We can consider the *ClusterEval* component as a reduce function that reduces a distributed graph object to a single value. We implemented the modularity and normalized cut computations using the Spark GraphX API.

Finally, we created a *GraphCoarsen* component that can be used to simplify a big graph. *GraphCoarsen* takes a cluster graph as input and replaces a set of vertices in a cluster with a super vertex. The output is a graph object where each super vertex attribute is the number of vertices that form the supper vertex. The coarsening implementation is based on the pseudocode provided in [10].

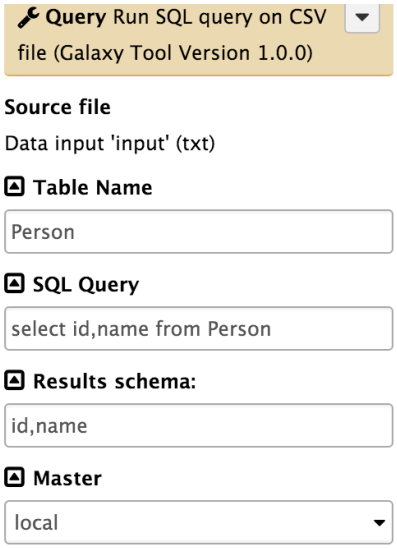

Figure 2. The Query tool expects a table name and query on the given table name. Providing the Query tool with the output schema is optional.

Relational tools consists of *Info*, *Query*, *JoinQuery*, and *PredefinedQueries* components. These relation components are a very important part of every workflow represented in GraphFlow because we can use them to transform or constrain dataframes or to join the output of multiple components.

The *Info* tool collects the schema, the number of available data points, and some samples of data points from the given dataframe in order to guide the user in constructing valid queries.

The *Query* component runs an SQL query over the given input dataframe. In order to run a query over a dataframe, it first registers the input dataframe as a relational table with the given name, and then executes the query on the relational table. Figure 2 shows the parameter of page of the *Query* tool and an example SQL query. The *Query* component also expects the schema of the output dataframe in order to construct appropriate named columns. The given names are specifically useful when we want to run other queries on the output dataframe. To simplify using the relational queries, we provide a set of common queries in *PredefinedQueries*.

The *JoinQuery* component is similar to *Query*, except it accepts two dataframes as inputs, so we can run join queries on both dataframes. Similar to *Query*, we provide names for the tables, schema for the results, and the SQL query. *JoinQuery* is specifically useful when we want to combine the information of two dataframes.

Statistics tools: the goal of these components is to collect statistics from the dataframe. Cumulative density function (CDF) has been extensively used in practice to study the data distribution, which is also provided here.

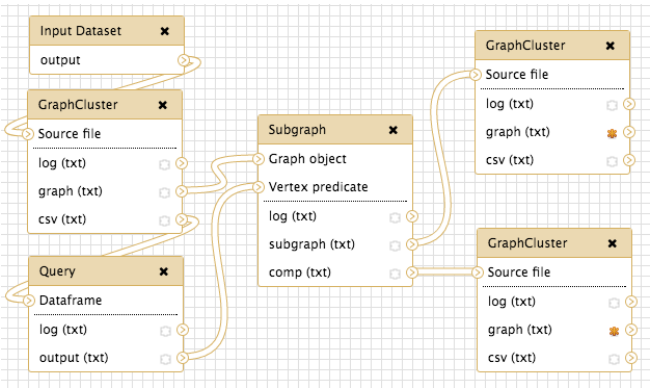

Figure 3. The workflow of hierarchical clustering using Subgraph, GraphCluster, and Query.

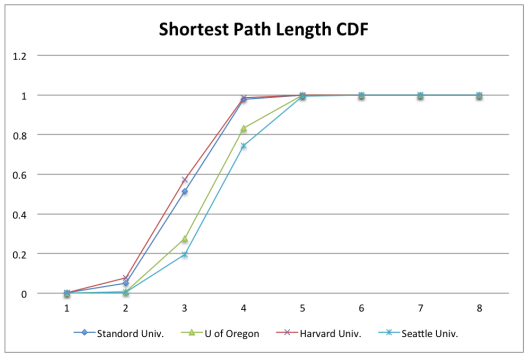

Figure 4. CDF of the shortest path length from the all nodes of the graph to vertices of Harvard University, Stanford University, University of Oregon, and Seattle University.

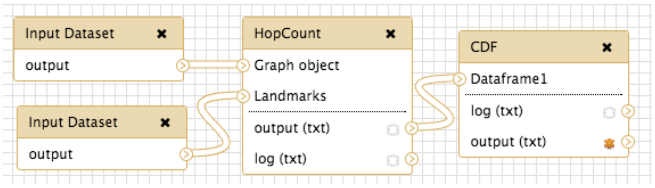

Figure 5. The workflow of finding the CDFs of shortest paths.

Plotting tools: includes different plotting components such as ScatterPlot, and HistogramPlot which summarizes a dataframe for further analytic studies.

We can also create more complex components by combining these tools, for example, we can create a hierarchical clustering workflow using the *GraphCluster*, *Subgraph*, and *Query* components as shown in Figure 3. In this workflow, the *GraphCluster* is configured with a maximum of two clusters. Then, we use the cluster assignment to select the vertices that belong to one cluster and feed that to *Subgraph* along with the cluster graph. *Subgraph* partitions the given graph into two subgraphs, each belonging to one cluster. Finally, we apply *GraphCluster* to each of these subgraphs.

IV. Use Cases

In order to show the expressiveness of the GraphFlow components, we construct different workflows to study the structural properties of graphs constructed from the Wikipedia datasets[3]. This dataset is a crowd-source gathered information from Wikipedia and includes several data files such as page links and abstracts. Each line in the page link dataset contains a pair of URIs such that the second URI appears in the Wikipedia Web page of the first URI. As an example of URIs, "http://dbpedia.org/resource/Stanford_University" is the URI of Stanford University Wikipedia page. The abstracts includes the URI of a Wikipedia page and the main section of each page.

In order, to construct the Wikipedia graph, we assign a unique ID to each URI, which identifies a vertex in the graph. Two vertices are connected if the pairs of their URIs appear together in the page links dataset. We simply ignore the order of URIs in each pair, so the final graph is undirected. The constructed graph consists of more than 20 million vertices and 159 million edges. Moreover, we keep the URIs and the assigned IDs in a CSV file as URI data file, which we use for finding the corresponding URI assigned to each vertex.

For these experiments, we ran the cluster system (Figure 1) on the ACISS cluster[4], and we ran the Galaxy frontend on a laptop. We used five Spark nodes, each running on an Intel(R) Xeon(R) CPU X5650@2.67GHz with access to a total of 50GB of memory.

A. Degree Distribution

Degree distribution is well-studied metric for graphs. In order to find the degree distribution, we first use the Node Degree components to get the degree of each vertex as a CSV file with schema "vertex,degree", then the following SQL queries gives us the distribution:

```
SELECT degree, count(degree)
FROM degreeTable
GROUP BY degree,
```

where the degreeTable is the relation name that we use to register the input degree CSV file. Finally we redirect the output to the plotting component.

The degree distribution of Wikipedia graphs mostly follows power-law degree distribution, Figure 8.

B. Shortest-Path Length Distribution

Shortest paths length in a graph has been used for defining the closeness centrality, which shows the relative positions of a given vertex with respect to all other vertices in the graph.

[3]http://wiki.dbpedia.org/

[4]http://aciss-computing.uoregon.edu/

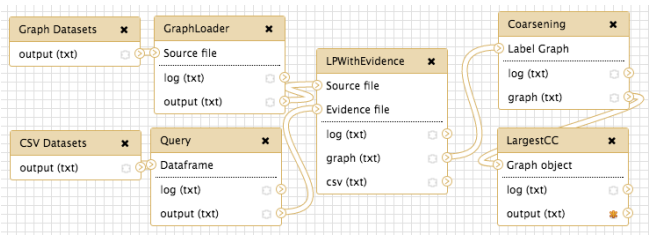

Figure 6. The workflow of coarsening a graph using clustering.

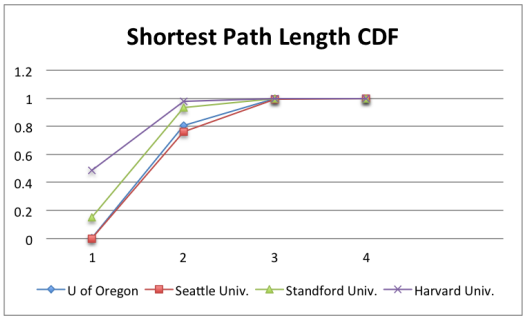

Figure 7. CDF of the shortest path length in the coarse graph.

However, looking at the shortest-path length distribution is more informative. The *ShortestPath* component generates the shortest path lengths from each vertex in the graph to a set of predefined landmarks. We use the set of vertices corresponding to different universities as landmarks, and generate the cumulative density function (CDF) for each of these universities. Figure 4 shows these CDFs, which indicate the closeness of the landmarks with respect to other vertices in the graph. For example, approximately 45% of shortest paths toward the Standford University page have length smaller or equal to 3, while this value is only 20% for Seattle University.

C. Coarsening

Coarsening of very large graphs enables analysis with fewer resources. However, the coarsening process should preserve the properties of the original graph. For example, suppose we are interested in a subgraph of the Wikipedia graph that includes the pages of universities, colleges, institutes, and related pages. We select the pages if their URIs include University, Institute, or College, and refer to them as academic pages. Using the *Subgraph* component may result in removing all pages not belonging to set of vertices of the interest and ignoring their effect on the coarse graph. For example, the Oregon Ducks Football team page will not appear in the set of vertices and *Subgraph* ignores the paths that connected University of Oregon to universities thorough their football pages. Therefore, we need to find a community around each page of interest. This is similar to local clustering. For this purpose, we modify the label propagation algorithm and put a weight on each label. Setting uniform weights reduces the local label propagation to original label propagation. For our purpose, we set the weights of labels belonging the academic pages to large values, while all other weights are set to one. This forces communities to be formed around the academic pages.

We feed the output of the local label propagation algorithm, which is a cluster graph (where the attribute of every vertex is its cluster id) to the coarsening component and obtain its largest connected components. This workflow is shown in Figure 6. The academic pages include about 140K pages, however, the largest connected component of the coarse graph has only 14K vertices, comparing to the 20M vertices of the original Wikipedia graph. To check whether the coarse graph preserves the structure, we look at the CDF of the shortest paths of the same universities studied in previous section. To do so, we can easily feed the output of the coarsening workflow, Figure 6, to the shortest path workflow, Figure 5. Figure 7 shows the resulting CDF of the shortest paths to the given landmarks, indicating that the coarse graph has structure similar to that of the original graph.

D. Pagerank Centrality

PageRank is a well-known variation of eigenvector centrality. With PageRank, we can sort the vertices based on their rank score. Our goal in this use case is to rank universities based on their appearance in the Wikipedia using the PageRank algorithm [23]. Therefore, the rank of a university depends on the Wikipedia pages that have links to the Wikipedia page of the university and importance of those pages based on the ranking.

To find the Wikipedia pages of universities we simply use the URI name and look for related words such as University, Institute, or College. An alternative approach would be to use the abstract file, but here the URI name seems sufficient. Therefore the result of the search is a dataframe that includes the ID and URI of universities.

Figure 9 shows the workflow of the experiment. The graph dataset points to the edge-view of the Wikipedia graph constructed from the Pagelink file, and the CSV dataset points to the URI data file.

The *PageRank* tool ranks the vertices of the Wikipedia graph, and the output dataframe is given to *JoinQuery* tool. The SQL query given to the *JoinQuery* joins two dataframes, so we access the URI of each vertex as well as its rank. We can then restrict the results to the URIs. The *JoinQuery* register the output of the *PageRank* and *Query* tools as relational tables *ranks* and *univ*, respectively, and runs the following SQL query on them:

```
SELECT name, rank from uri, ranks
WHERE ranks.vertex = uri.vertex
AND (name LIKE "%University%"
OR name LIKE "%Institute%"
```

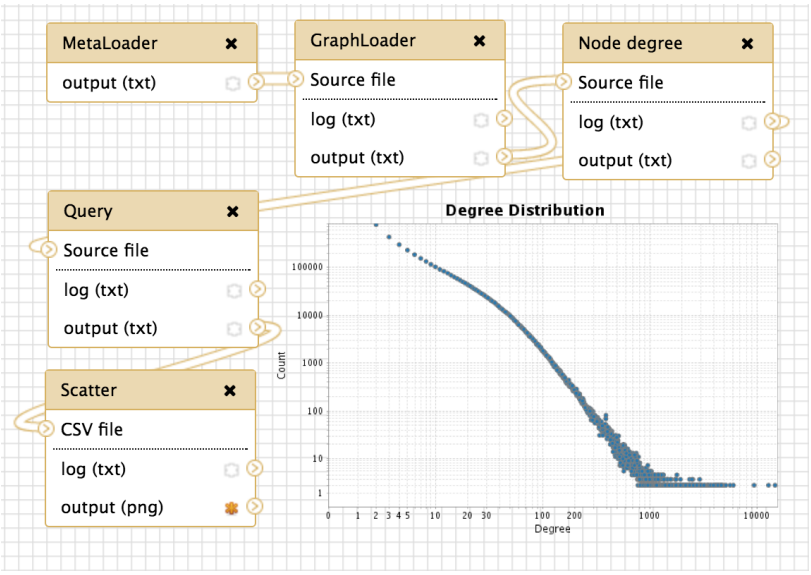

Figure 8. The degree distribution of Wikepedia graph along with the corresponding workflow.

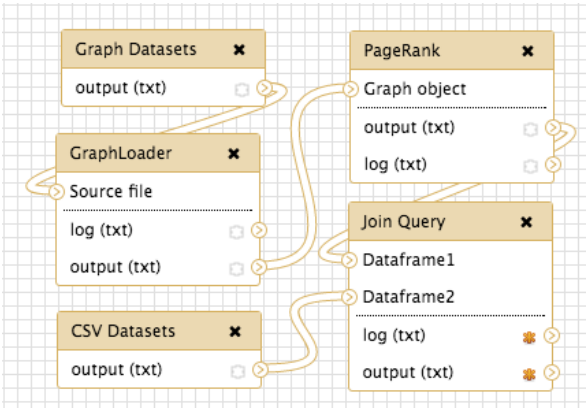

Figure 9. The workflow of ranking universities using Wikipedia graph.

```
OR name LIKE "%College%" )
ORDER BY ranks.rank DESC limit 100
```

Table I includes the top 10 of the final ranking result produced by our example workflow, the Wikipedia ranking reported by Lages et al. [23], and the QS survey-based ranking [24]. The Wikipedia-based top 10 lists have nine common entries. The difference in Wikipedia-based rankings is most likely attributable to the fact that we only used English Wikipedia pages while Lages et. al use all provided Wikipedia pages.

V. Conclusion

We introduced the GraphFlow toolkit, a workflow-based system for large-scale distributed graph analysis. GraphFlow provides the user with a set of Spark-based tools that can be combined together using the intuitive Galaxy workflow manger in order to describe complex data science experiments. Using GraphFlow, researchers can re-run their complex experiments with different parameter settings and over different input data. Moreover, workflows can be shared, reused, or composed into larger applications, as shown in the case studies. GraphFlow hides the complexity of interacting with cluster systems and data-parallel processing frameworks, significantly simplifying large-scale graph analysis.

References

[1] B. H. Junker and F. Schreiber, *Analysis of Biological Networks (Wiley Series in Bioinformatics)*. New York, NY, USA: Wiley-Interscience, 2008.

[2] S. Wasserman and K. Faust, *Social network analysis: Methods and applications*. Cambridge university press, 1994, vol. 8. [Online]. Available: http://scholar.google.com/scholar.bib?q=info:gET6m8icitMJ:scholar.google.com/&output=citation&hl=en&as_sdt=0,5&as_vis=1&ct=citation&cd=0

[3] E. Stattner and N. Vidot, "Social network analysis in epidemiology: Current trends and perspectives," in *Research Challenges in Information Science (RCIS), 2011 Fifth International Conference on*, May 2011, pp. 1–11.

[4] H. Jeong, S. P. Mason, A. L. Barabasi, and Z. N. Oltvai, "Lethality and centrality in protein networks," *Nature*, vol. 411, no. 6833, pp. 41–42, May 2001. [Online]. Available: http://dx.doi.org/10.1038/35075138

[5] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '07. New York, NY, USA: ACM, 2007, pp. 29–42. [Online]. Available: http://doi.acm.org/10.1145/1298306.1298311

Table I
TOP 10 UNIVERSITIES FOUND USING WORKFLOW OF FIGURE 9 COMPARED TO WIKIPEDIA UNIVERSITY RANKING FROM [23] AND THE SURVEY-BASED RANKINGS [24].

	Ranking from [23]	GraphFlow	QS Ranking [24]
1st	University of Cambridge	Harvard University	Massachusetts Institute of Technology
2nd	University of Oxford	University of Oxford	Stanford University
3rd	Harvard University	Columbia University	Harvard University
4th	Columbia University	University of Cambridge	University of Cambridge
5th	Princeton University	Yale University	California Institute of Technology
6th	Massachusetts Institute of Technology	Stanford University	University of Oxford
7th	University of Chicago	University of California, Berkeley	University College London
8th	Stanford University	Massachusetts Institute of Technology	ETH Zurich
9th	Yale University	University of Michigan	Imperial College London
10th	University of California, Berkeley	Princeton University	University of Chicago

[6] K. Mehlhorn and U. Meyer, "External-memory breadth-first search with sublinear I/O," in *Proceedings of the 10th Annual European Symposium on Algorithms*, ser. ESA '02. London, UK, UK: Springer-Verlag, 2002, pp. 723–735. [Online]. Available: http://dl.acm.org/citation.cfm?id=647912.740673

[7] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu, "Turbograph: A fast parallel graph engine handling billion-scale graphs in a single PC," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, Chicago, IL, USA, 2013, pp. 77–85.

[8] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[9] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie, "Pregelix: Big(ger) graph analytics on a dataflow engine," *Proceedings of the VLDB Endowment*, vol. 8, no. 2, pp. 161–172, 2014.

[10] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI 14, Broomfield, CO, USA, 2014, pp. 599–613.

[11] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A peta-scale graph mining system implementation and observations," in *Proceedings of the 9th IEEE International Conference on Data Mining*, ser. ICDM '09, Miami, FL, USA, 2009, pp. 229–238.

[12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.

[13] J. Goecks, A. Nekrutenko, J. Taylor *et al.*, "Galaxy: A comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences," *Genome Biol*, vol. 11, no. 8, p. R86, 2010.

[14] T. Oztan, R. Sinkovitz, and T. Menezes, "Complex social science (CoSSci) gateway: Autocorrelation modeling, kinship network modeling, k- and pairwise cohesion in large networks & open opportunities for online education. http://socscicompute.ss.uci.edu."

[15] B. Liu, R. K. Madduri, B. Sotomayor, K. Chard, L. Lacinski, U. J. Dave, J. Li, C. Liu, and I. T. Foster, "Cloud-based bioinformatics workflow platform for large-scale next-generation sequencing analyses," *Journal of biomedical informatics*, vol. 49, pp. 119–133, 2014.

[16] C. Sloggett, N. Goonasekera, and E. Afgan, "BioBlend: automating pipeline analyses within galaxy and cloudman," *Bioinformatics*, vol. 29, no. 13, pp. 1685–1686, 2013.

[17] L. Pireddu, S. Leo, N. Soranzo, and G. Zanetti, "A Hadoop-Galaxy adapter for user-friendly and scalable data-intensive bioinformatics in Galaxy," in *Proceedings of the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics*. ACM, 2014, pp. 184–191.

[18] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining." in *SDM*, vol. 4. SIAM, 2004, pp. 442–446.

[19] F. Lin and W. W. Cohen, "Power iteration clustering," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 655–662.

[20] D. A. Spielmat and S.-H. Teng, "Spectral partitioning works: Planar graphs and finite element meshes," in *Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on*. IEEE, 1996, pp. 96–105.

[21] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner, "On modularity clustering," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 20, no. 2, pp. 172–188, 2008.

[22] J. Shi and J. Malik, "Normalized cuts and image segmentation," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 22, no. 8, pp. 888–905, 2000.

[23] J. Lages, A. Patt, and D. L. Shepelyansky, "Wikipedia ranking of world universities," *arXiv:1511.09021 [cs.SI]*, 2015.

[24] Times Higher Education, "World university rankings," https://www.timeshighereducation.com/world-university-rankings/2017, 2016.