Graph Mining Meets the Semantic Web

Sangkeun Lee, Sreenivas R. Sukumar and Seung-Hwan Lim

Computational Sciences and Engineering Division
Oak Ridge National Laboratory, TN, USA
Email: lees4@ornl.gov, sukumarsr@ornl.gov and lims1@ornl.gov

***Abstract*—The Resource Description Framework (RDF) and SPARQL Protocol and RDF Query Language (SPARQL) were introduced about a decade ago to enable flexible schema-free data interchange on the Semantic Web. Today, data scientists use the framework as a scalable graph representation for integrating, querying, exploring and analyzing data sets hosted at different sources. With increasing adoption, the need for graph mining capabilities for the Semantic Web has emerged. We address that need through implementation of three popular iterative *Graph Mining* algorithms (Triangle count, Connected component analysis, and PageRank). We implement these algorithms as *SPARQL* queries, wrapped within *Python* scripts. We evaluate the performance of our implementation on 6 real world data sets and show graph mining algorithms (that have a linear-algebra formulation) can indeed be unleashed on data represented as RDF graphs using the SPARQL query interface.**

I. Introduction

Graph Mining techniques have been applied to relationship-oriented Big Data problems in various domains. Some examples include analysis of terrorist networks in the homeland security, protein-protein interactions in the life sciences, threat identification in cyber security, and guilt-by-association studies for fraud detection in electronic marketplaces. These techniques reveal patterns or characteristics latent in the graphs, by computing graph-theoretic measures such as *triangle count* [1], *degree distribution*, *eccentricity* [2], *connected component analysis* and *PageRank* [3] / *Personalized PageRank*[4]. Unfortunately, leveraging these algorithms on datasets published using the Semantic Web has always been challenging. Our goal in this paper is to introduce the power of in-situ graph mining to the Semantic Web Community.

As data scientists witnessing the increasing trend in both successful application of graph mining and dissemination of datasets using Semantic web tools, we see that the marriage of graph mining and the Semantic Web can have tremendous potential for knowledge discovery from disparate data sources. We attribute this to the active development in the Semantic Web community. In particular, the progress made in the definitions of the Resource Description Framework (RDF) and SPARQL Protocol and RDF Query Language (SPARQL) - the two major components of the Semantic Web. The RDF is the data model originally proposed by the World Wide Web Consortium (W3C) for expressively representing data resources on the Web. SPARQL (SPARQL Protocol and RDF Query Language) is the query language for datasets in RDF formats. Combined together, RDF and SPARQL offer a flexible and expressive data model to represent data resources along with the ability to integrate, query, explore and analyze data - even if datasets reside in multiple warehouses. They allow representation of information at fine granularity and as W3C standards enable inter-operability of implementation to several databases (triplestores) that support RDF formats. In fact, triplestores such as Jena [5], Sesame [6], and RDFSuite[7]), distributed triplestores (e.g., SPARQLVerse [8]), and graph processing appliances (e.g, Urika [9]) can all stage RDF datasets and support SPARQL. By definition, a SPARQL query written according to standard, executed successfully on one triplestore, should be executable on any of the other standard-compliant triplestores.

However, implementing graph mining algorithms (e.g., PageRank, connected component analysis, node eccentricity, etc.) using SPARQL queries is not a trivial task. The complexity arises from the fact that most graph-theoretic algorithms have a linear-algebra formulation and assume adjacency-matrices as the default data structure. Matrix and array structures are not straightforward to realize using the SPARQL query algebra. One has to redesign algorithms that can handle the triple representation and algorithms have to be simplified for graph operations supported by the SPARQL-query algebra. Also, most graph mining techniques are iterative algorithms and SPARQL does not support iterative querying.

We address these challenges and bring graph mining techniques to the Semantic Web. We implement three graph mining algorithms as a sequence of SPARQL queries wrapped in *Python* scripts. We are successfully able to implement even iterative graph mining algorithms using our approach and claim the following main contributions of this paper:

- We present the implementation of three graph mining algorithms *triangle count*, *connected component analysis*, *PageRank* for RDF datasets accessible through SPARQL endpoints. These algorithms can be exploited to analyze plenty of readily available RDF data sets, such as Linking Open Data (LOD) Project Cloud [10], OpenMaps, Freebase, DBPedia, etc.
- We evaluate the performance of our implementations with Jena TDB - one of the widely used triple stores. The experimental results confirm that our implementation of graph mining algorithms using SPARQL can be extended to perform mining tasks for a wide range of real world graphs with performance comparable to linear-algebra based methods even on a laptop computer.

Our approach leverages concepts and programming mod-

els from graph mining systems described in [11] and [12]. We believe, our implementation has inherited the ability to efficiently process large-scale graphs borrowing distributed computing principles from Pegasus [11] and Pregel [13]. Furthermore, we expect that our SPARQL implementation will inspire extensions to other graph query languages (e.g., Cypher [14], Gremlin[15], etc.) on graph databases such as Neo4j [14], DEX [16], and Titan [17].

The rest of this paper is organized as follows. Section II reviews relevant background and related work. Section III introduces software design and implementation of graph mining algorithms. Section IV shows the performance evaluation on 6 public real world datasets, followed by conclusions and directions for future work in Section V.

II. Background

The Resource Description Framework (RDF) is a data model published by The World Wide Web Consortium (W3C) in 1999. An RDF collection is of a set of *triples* in the form of <*subject*> <*predicate*> <*object*>. We use the term triplestore to describe a database that stores RDF triples and allows retrieval of triples using the SPARQL query language. We note that a collection of RDF triples is the generic representation of a graph data structure - consisting of nodes as entities and edges as relationships/associations between entities. As an additional feature, triples are seamless representations of heterogeneous graphs (graphs with different types of entities and different types of interactions). We consider RDF, SPARQL, and triplestore as the graph data model, graph query language, and the graph database respectively.

Today, the only way to conduct graph-analysis on triplestores is using SPARQL - which is a standard RDF query language also endorsed by W3C. Given a RDF data graph, SPARQL is an excellent tool to retrieve occurrences of a basic user-specified graph pattern. SPARQL even supports querying conjunctions and disjunctions along with retrieval operators such as projection, distinct, order, limit, and aggregation functions. Thanks to the active development in the Semantic Web community, there are various triplestores, specifically optimized for the storage and retrieval of RDF triples using SPARQL queries. In the following Section, we describe how we can connect to SPARQL query-endpoints and iteratively retrieve triples to implement graph-theoretic algorithms.

III. Implementing Graph Mining Using SPARQL

A. Software Design

Fig. 1 shows the conceptual software design for implementing graph mining algorithms on an RDF triplestore. The input RDF graph of interest is first staged into a triplestore as the *default Graph*. We then break down the algorithm of interest into a sequence of SPARQL-friendly processing steps. As illustrated, our approach uses both Python and SPARQL to process the graph data. The SPARQL queries enable processing within the triplestore, while the Python scripts interact with SPARQL endpoints and control the logical flow of the algorithm expressed as a series of SPARQL queries.

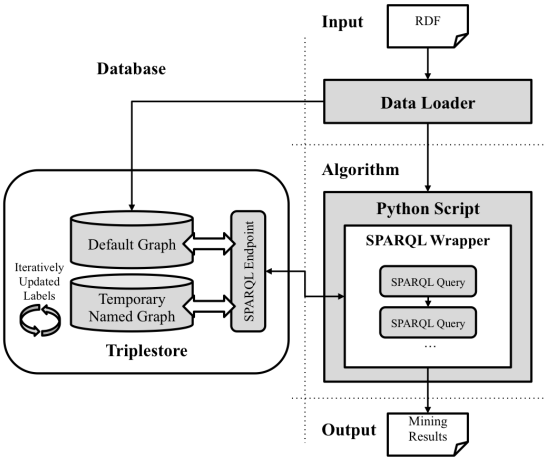


Fig. 1. The conceptual design of implementing graph mining algorithms using SPARQL queries wrapped in Python scripts

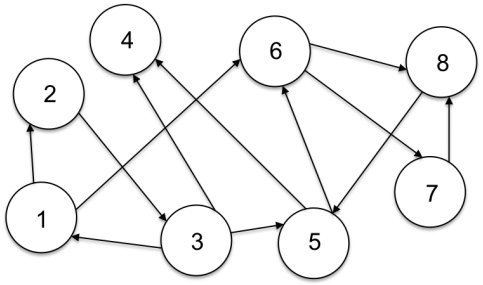


Fig. 2. An Example Graph

We used the SPARQLWrapper [18] library for interactions between the Python script and the triplestore.

We exploit the *named-graphs* feature supported by triplestores to maintain a copy of intermediate results as we process the input graph. A named-graph within a triplestore is a collection of RDF triples uniquely tagged using an identifier. A triplestore hosting a dataset can contain many named-graphs involving the same vertices and edges as the *default graph*. The named graphs feature is particularly important for iterative algorithms such as PageRank and connected component analysis. We will be explaining the details of each algorithm expressed as a sequence of SPARQL queries wrapped within Python scripts in the paragraphs below.

B. Software Implementation

In the next few paragraphs, we describe the implementation of three fundamental graph mining algorithms. Our choice was guided by the popularity of the triangle counting [19], connected component analysis [20] and PageRank [3] implementations in state-of-the-art graph mining systems such as GraphX [12], Pegasus [11], and Pregel [13].

1) ***Triangle Count****:* As the most basic subgraph in graphs, *triangles* play an important role in graph analysis [21]. For example, in social network graphs, counting triangles helped understand the *homophily* (people tend to be friends with other people that are similar to them) and the *transitivity* (people

tend to be friends with friends of friends). Our implementation counts distinct triangles composed of unique set of three nodes by using *FILTER* and *DISTINCT* keywords of SPARQL. Our approach is cognizant to the fact that the three edges of a triangle can have two different directions. Listing 1 shows the pseudo code for counting the number of distinct triangles in a graph.

Pseudo Python Code with SPARQL:

```
''' Counting triangles in a graph '''

rs = execute_query("""
SELECT (COUNT(*) AS ?numOfTriangle) WHERE {
    SELECT DISTINCT ?x ?y ?z
        WHERE {
            {?x ?p ?y} UNION {?y ?p ?x}.
            {?y ?p ?z} UNION {?z ?p ?y}.
            {?z ?p ?x} UNION {?x ?p ?z}.
            FILTER(STR(?x) < STR(?y)).
            FILTER(STR(?y) < STR(?z)).
        }
    }
""")

print_result(rs)
```

Example Result for the Graph in Fig.2:

```
-----------------
| numOfTriangle |
=================
| 4             |
-----------------
```

Listing 1. Counting triangles in a graph

*2) **Connected Component Analysis**:* The connected component of an undirected graph is a maximal set of nodes that can reach each other through paths in the graph. Finding all connected components in a graph is a fundamental operation that is used for various applications trying to understand community structure of a social network, predict trends in academia research and ranking of web pages [22]. Connected component analysis requires undirected graphs as input. Our implementation when faced with a directed graph, will ignore the direction of edges. Listing 2 shows the pseudo Python codes with SPARQL queries for finding all connected components in a graph. The output of the algorithm assigns a unique label to each node within a connected component.

Pseudo Python Code with SPARQL:

```
'''
    Initialize a named graph for storing labels
'''

execute_update("""
DROP GRAPH <ng:labels>; CREATE GRAPH <ng:labels>;

INSERT { GRAPH <ng:labels> {?s <label:comp_id> ?str_label} }
WHERE {
        SELECT ?s (str(?s) AS ?str_label)
        WHERE {
            {?s ?p ?o.}
            UNION
            {?q ?p ?s.}
        }
};""")


'''
   Iteratively update label of each node referring
   to the labels of the node's adjacent nodes
'''

converged = False
while not converged:
    execute_update("""
    DELETE
        { GRAPH <ng:labels>
            {?s <label:comp_id> ?original.}
        }
    INSERT
        { GRAPH <ng:labels>
            {?s <label:comp_id> ?update.}
        }
    WHERE {
        { GRAPH <ng:labels>
            {?s <label:comp_id> ?original.}
        }
        {
        SELECT ?s (MIN(?label) AS ?update) WHERE {
            {
            {?s ?p ?o.}
                { GRAPH <ng:labels>
                    {?o <label:comp_id> ?label.}
                }
                UNION
                { GRAPH <ng:labels>
                    {?s <label:comp_id> ?label.}
                }
            }
            UNION
            {
            {?o ?p ?s.}
                { GRAPH <ng:labels>
                    {?o <label:comp_id> ?label.}
                }
                UNION
                { GRAPH <ng:labels>
                    {?s <label:comp_id> ?label.}
                }
            }
        }
        GROUP BY ?s
        }
    };""")
    if there is no more update in <ng:labels>:
        converged = True

'''
    Retrieve the results
'''

rs = execute_query("""
SELECT (?s as ?node) (?o as ?comp_id)
WHERE { GRAPH <ng:labels> {?s <label:comp_id> ?o} };""")
print_result(rs)
```

Example Result for the Graph in Fig.2:

```
------------------------
| node      | comp_id  |
========================
| <node:4> | "node:1" |
| <node:2> | "node:1" |
| <node:6> | "node:1" |
| <node:1> | "node:1" |
| <node:8> | "node:1" |
| <node:3> | "node:1" |
| <node:5> | "node:1" |
| <node:7> | "node:1" |
------------------------
```

Listing 2. Finding connected components in a graph

Initially, an unique integer label l_{n_i} is assigned to each node n_i. We add triples such as (n_i, <label:min>, l_{n_i}) in a named graph graph <ng:labels>. The named graph <ng:labels> stores the intermediate and final results. Then, we iteratively update the temporary named graph as follows; for each node in the G, the associated label in the temporary graph is updated as the minimum of its neighbor's labels. The iteration continues until there is no more label update between iterations. Then,

we retrieve <label:comp_id> label's value of each node. The value of *comp_id* identifies the membership of each node to its connected components. All nodes in the same connected component will have the same label, and once converged, the number of distinct labels will reveal the number of connected components in the graph.

3) ***PageRank****:* PageRank is one of the most widely used link analysis algorithms, that is designed to measure the importance of nodes in a graph. The algorithm is originally designed to rank web pages that are linked to each other, but has been leveraged in other graph analysis applications. *PageRank* is computed iteratively using the formula, $\vec{r} = \alpha P^T \vec{r} + (1-\alpha)\frac{1}{N}\vec{e}$, where N is the number of vertices in the graph, P is the transition matrix for the graph, r_i is the *PageRank* value for node v_i, $\vec{e}$=$(1, 1, \ldots,)^T$, and α is a damping factor, usually 0.85. PageRank can be computed using SPARQL similar to how we conducted the connected component analysis. Listing 3 shows the pseudo code of our implementation.

Pseudo Python Code with SPARQL:

```
'''Parameters'''

numOfNodes = get_node_num()
dampingFactor = 0.85
addTerm = (1-dampingFactor)/numOfNodes
top_k = 10
convergenceThreshold = 1e-6

''' Initialize labels '''

execute_update("""
DROP SILENT GRAPH <ng:labels>;

INSERT {
    GRAPH <ng:labels>
        { ?s <label:outdegree> ?outdegree. }
    }
WHERE
{
    SELECT ?s (COUNT(*) AS ?outdegree)
    {
        ?s ?p ?o.
    }
    GROUP BY ?s
};""")

'''Set initial PageRank scores (1/numOfNodes)'''
execute_update("""
INSERT {
    GRAPH <ng:labels>
        { ?s <label:prevPR> """+str(1/numOfNodes)+""" }
    }
WHERE
{
    SELECT DISTINCT ?s
    WHERE
    {
        {?s ?p ?o.} UNION {?o ?p ?s.}
    }
};""")

''' Power iteration computation '''

while not converged:
    execute_update("""
    DELETE { GRAPH <ng:labels> { ?s <label:newPR> ?o. } }
    WHERE
    {
        GRAPH <ng:labels> {?s <label:newPR> ?o.} }
    ;
    INSERT {
        GRAPH  <ng:labels>
        { ?s <label:newPR> ?Contribution. }
        }
    WHERE
    {
        SELECT ?s
        ((SUM(?val2/?val1)*"""
        +str(dampingFactor)+""")+"""+str(addTerm)+"""
        AS ?Contribution) WHERE
        {
            {
                {?x ?p ?s.}
                { GRAPH <ng:labels>
                    {
                        ?x <label:outdegree> ?val1 .
                        ?x <label:prevPR> ?val2
                    }
                }
            }
        }
        GROUP BY ?s
    };""")

    ''' Checking if converged '''
    rs = execute_query("""
    SELECT (MAX(?diff) AS ?maxDiff)
    WHERE
    {
    GRAPH <ng:labels>
        {
            ?s <label:prevPR> ?o1 .
            ?s <label:newPR> ?o2 .
            BIND(ABS(?o1 - ?o2) AS ?diff)
        }
    }""")

    maxDiff = rs["diff"][0]
    if maxDiff <= convergenceThreshold:
        converged = True

''' Retrive the results '''

rs = execute_query("""
SELECT (?s AS ?node) (?newPR AS ?PageRank)
WHERE { GRAPH <ng:labels>
    {
     ?s <label:newPR> ?newPR .
    }
}
ORDER BY DESC(?newPR)
LIMIT """+str(top_k)+"""
""")
print_result(rs)
```

Example Result for the Graph in Fig.2:

node	PageRank
<node:5>	0.11578173283998455463965470
<node:8>	0.09875817744292230242966625
<node:6>	0.08148819567226215851691275
<node:4>	0.08104432629332689372493405
<node:7>	0.05338267626209973384203380
<node:3>	0.04618852554559382720807230
<node:2>	0.03228061828513480428418840
<node:1>	0.03183674890619953949220970

Listing 3. Computing PageRank for a directed graph

The implementation can be explained as follows. Outgoing edge degree for each node is stored in the named graph <ng:labels> using <label:outdegree> labels. Then, initial PageRank score $1/N$, where N is the number of nodes in the graph, is assigned to every node. Next, new PageRank score for each node is computed by using the previous PageRank score of its immediate neighbors. The new PageRank scores are added to the <ng:labels> named graph using the predicate <label:newPR>. Differences between previous and new PageRank scores for all nodes are then computed. If the max-

TABLE I
SUMMARY OF DATA SETS.

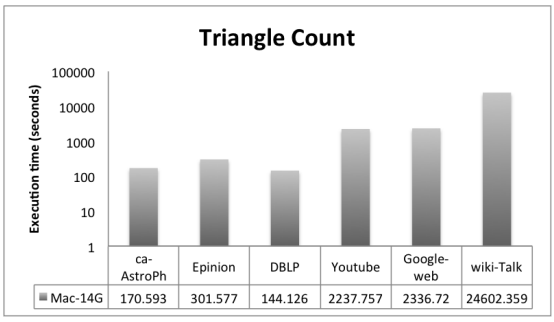
Name	Number of edges	Number of vertices	Number of triangles	Comments
ca-Astro	396,160	18,772	1,351,441	Collaboration network of Arxiv Astro Physics, undirected
Epinion	508,837	75,879	1,624,481	Who-trusts-whom network of Epinions.com, directed
DBLP	1,049,866	317,080	2,224,385	DBLP collaboration network, undirected
Youtube	2,987,624	1,134,890	3,056,386	Youtube online social network, undirected
Google-Web	5,105,039	875,713	13,391,903	Web graph from Google, directed
Wiki-talk	5,021,410	2,394,385	9,203,519	Wikipedia talk (communication) network, directed

imum value among all differences is smaller or equal to the convergence threshold value, the iteration stops. The default convergence threshold in our implementation is $\frac{1}{\#ofnodes \times 10}$. Finally, the top-k highest PageRank scores are printed by retrieving the values of <label:newPR> in <ng:labels>. We note that our implementation considers a directed graph as input to the algorithm.

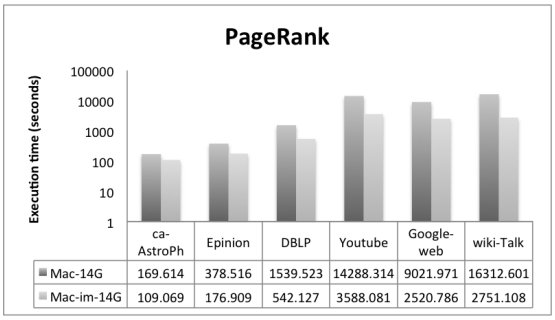
IV. PERFORMANCE EVALUATION

In order to test the performance of our implementation, we executed each graph mining algorithm on 6 public real world data sets released by Stanford Network Analysis Project (SNAP) [23]. We converted the edge-list datasets to RDF triples. We conducted our experiments on a laptop computer running OS X version 10.10 with 2.3 GHz Intel Core i7 CPU and 16 GB 1600 MHZ DDR3 memory. We used Jena Fuseki SPARQL server 1.0.2, which comes with Jena TDB triplestore. Java runtime version 1.8.0 was used to run Jena Fuseki, and Java heap space to was set to 14 GB. Apache Jena allowed in-memory processing and persistent-processing options. The in-memory option caches intermediate results in memory while the persistent option works by saving result sets to disk. We evaluated the performance of graph mining algorithms for both options.

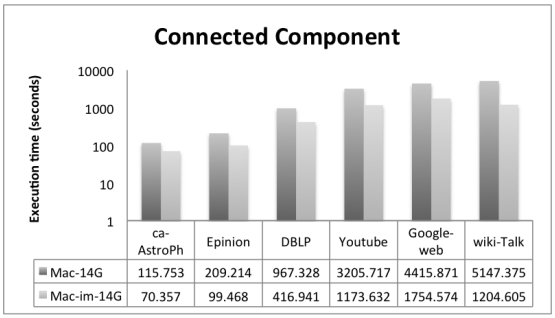
Table 1 shows the summary of data sets used in our experiments. Fig. 3 shows the execution time for each algorithm on our test datasets. The execution time does not include the time required to load RDF triples into the triplestore. We assume that the data to be analyzed is already loaded. However, we include the time for executing the algorithm and writing the result to an output file. As expected, we find that processing time increases according to the graph size. For PageRank and connected component analysis algorithms, the in-memory option (*Mac-im-14G*) showed better performance than persistent option (*Mac-14G*). With the exception of the triangle counting algorithm on the *wiki-Talk* dataset, the rest of the results are in the order of a few minutes. This is very comparable to results from linear-algebra based implementations of the same algorithms. A comparison of our implementation with different linear-algebra based methods and map-reduce based methods for graph processing is presented in [24]. The results in [24] along with the results in this paper clearly confirm that SPARQL-based implementation of graph mining algorithms can be unleashed on graph-data represented and hosted using Semantic Web standards and technologies.



(a) Execution time for Triangle Count



(b) Execution time for PageRank



(c) Execution time for Connected Component Analysis

Fig. 3. Performance Evaluation Results

V. CONCLUSION AND FUTURE WORK

This paper demonstrated implementation of three fundamental iterative graph algorithms using SPARQL queries wrapped within Python scripts. The experimental results are within reasonable latency requirements for interactive exploratory analysis and discovery. We will be adding more graph-theoretic algorithms (e.g. Shortest Path, Node Eccentricity, Community Clustering, etc.) to our software suite. In addition, we have identified the following items for future

work:

Better use of in-memory and persistent options in triple stores: Experimental results showed that storing temporary data using in-memory databases reduces latency while improving performance. In our current implementation, we had to choose between in-memory or persistent options while starting the SPARQL server. While each option serves a different purpose, we believe that we can simultaneously exploit both options to improve performance of graph mining algorithms on datasets that do not fit completely into memory. We will conduct experiments to intelligently cache temporary and intermediate data specific to the graph mining algorithm. The ability to optimally cache intermediate results is critical for large datasets that are stored in a distributed computing environment.

Developing building-blocks of graph mining algorithms: While implementing algorithms using SPARQL queries, we identified several common aspects of graph mining algorithms such as *node labeling*, *node-label propagation*, *and neighborhood-based node aggregation*. We will identify the necessary yet fundamental building blocks for graph mining algorithms and develop a set of application programming interfaces (APIs) to assist user-derived custom graph mining algorithms. These APIs will enable programmers to implement graph mining algorithms on triplestores with minimal understanding of SPARQL-algebra - analogous to how business intelligence tools abstract Structured Query Language (SQL) algebra in traditional databases for the end user.

Experiments with Big Data on graph-processing appliances: Cray's *Urika* is a massively parallel, multi-threaded, shared-memory computing environment optimized for SPARQL queries over large-scale RDF data sets. We will test our implementation on the Urika instance at Oak Ridge National Laboratory to investigate the scalability of graph mining in triplestores.

ACKNOWLEDGMENT

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

The authors acknowledge Tyler C. Brown who contributed significantly to the software development efforts during his internship at the Oak Ridge National Laboratory.

REFERENCES

[1] C. E. Tsourakakis, Fast counting of triangles in large real networks without counting: Algorithms and laws, in: Proceedings of the 2008 IEEE International Conference on Data Mining, 2008.

[2] P. Hage, F. Harary, Eccentricity and centrality in networks, Social networks 17 (1) (1995) 57–63.

[3] L. Page, S. Brin, R. Motwani, T. Winograd, The pagerank citation ranking: Bringing order to the web.

[4] H. Tong, C. Faloutsos, J.-Y. Pan, Fast random walk with restart and its applications, in: Proceedings of the 2006 IEEE International Conference on Data Mining, IEEE, 2006.

[5] M. Schmidt, T. Hornung, N. Küchlin, G. Lausen, C. Pinkel, An experimental comparison of RDF data management approaches in a SPARQL benchmark scenario, Springer, 2008.

[6] J. Broekstra, A. Kampman, F. Van Harmelen, Sesame: A generic architecture for storing and querying rdf and rdf schema, in: The Semantic WebISWC 2002, Springer, 2002, pp. 54–68.

[7] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, K. Tolle, The ics-forth rdfsuite: Managing voluminous rdf description bases., in: SemWeb, 2001.

[8] Z. Liu, A. Le Calvé, F. Cretton, N. Glassey, Using semantic web technologies in heterogeneous distributed database system: A case study for managing energy data on mobile devices, International Journal of New Computer Architectures and their Applications (IJNCAA) 4 (2) (2014) 56–69.

[9] S. R. Sukumar, N. Bond, Mining large heterogeneous graphs using crays urika, 2013 ORNL Computational Data Analytics Workshop.

[10] C. Bizer, T. Heath, T. Berners-Lee, Linked data-the story so far, International journal on semantic web and information systems 5 (3) (2009) 1–22.

[11] U. Kang, C. E. Tsourakakis, C. Faloutsos, Pegasus: A peta-scale graph mining system implementation and observations, in: Proceedings of the 2009 IEEE International Conference on Data Mining, 2009.

[12] R. S. Xin, J. E. Gonzalez, M. J. Franklin, I. Stoica, Graphx: A resilient distributed graph system on spark, in: First International Workshop on Graph Data Management Experiences and Systems, 2013, p. 2.

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

[14] I. Robinson, J. Webber, E. Eifrem, Graph databases, " O'Reilly Media, Inc.", 2013.

[15] F. Holzschuher, R. Peinl, Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j, in: Proceedings of the Joint EDBT/ICDT 2013 Workshops, 2013.

[16] N. Martínez-Bazan, V. Muntés-Mulero, S. Gómez-Villamor, J. Nin, M.-A. Sánchez-Martínez, J.-L. Larriba-Pey, Dex: high-performance exploration on large graphs for information retrieval, in: Proceedings of the sixteenth ACM conference on Conference on information and knowledge management, 2007.

[17] Titan graph database, http://thinkaurelius.github.io/titan/.

[18] SPARQLWrapper:SPARQL Endpoint interface to Python, http://rdflib.github.io/sparqlwrapper/.

[19] Z. Bar-Yossef, R. Kumar, D. Sivakumar, Reductions in streaming algorithms, with an application to counting triangles in graphs, in: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, 2002, pp. 623–632.

[20] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, X. Lin, Scalable big graph processing in mapreduce, in: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, 2014.

[21] C. E. Tsourakakis, U. Kang, G. L. Miller, C. Faloutsos, Doulion: counting triangles in massive graphs with a coin, in: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining, 2009, pp. 837–846.

[22] A. Varamesh, M. K. Akbari, Fast detection of connected components in large scale graphs using mapreduce, IOSR Journal of Engineering 4 (2014) 35–42.

[23] J. Leskovec, R. Sosič, SNAP: A general purpose network analysis and graph mining library in C++, `http://snap.stanford.edu/snap` (Jun. 2014).

[24] S.-H. Lim, S. Lee, S. R. Sukumar, G. Ganesh, T. C. Brown, Graph processing platforms at scale: Practices and experiences, in: Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software, 2015.