Graph Processing Platforms at Scale: Practices and Experiences

Seung-Hwan Lim, Sangkeun Lee, Gautam Ganesh, Tyler C. Brown, and Sreenivas R. Sukumar

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

***Abstract*—Graph analysis has revealed patterns and relationships hidden in data from a variety of domains such as transportation networks, social networks, clinical pathways, and collaboration networks. As these networks grow in size, variety and complexity, it is a challenge to find the right combination of tools and implementation of algorithms to discover new insights from the data. Addressing this challenge, our study presents an extensive empirical evaluation of three representative graph processing platforms: Pegasus, GraphX, and Urika. Each system represents a combination of options in data model, processing paradigm, and infrastructure. We benchmark each platform using three popular graph mining operations, degree distribution, connected components, and PageRank over real-world graphs. Our experiments show that each graph processing platform owns a particular strength for different types of graph operations. While Urika performs the best in non-iterative graph operations like degree distribution, GraphX outperforms iterative operations like connected components and PageRank. We conclude this paper by discussing options to optimize the performance of a graph-theoretic operation on each platform for large-scale real world graphs.**

I. Introduction

Graph analysis unveils insightful patterns hidden in data sets. This ability has brought graph mining to the forefront in knowledge discovery applications. The growing demand for graph analysis desires general-purpose graph processing platforms that can perform arbitrary graph operations over large data sets in a generic fashion, instead of customized platforms and services for each specific graph-operation. Responding to such a requirement, a few categories of general-purpose graph processing platforms have emerged. Some examples of such platforms are graph transactional systems [1] and graph processing systems [2]–[4], along with graph processing appliances that package graph processing software with infrastructure [5].

The primary benefit of general-purpose graph processing platforms is that it enables algorithm development over system administration tasks such as providing fault tolerance, optimizing data loading processes, and minimizing communication overheads. In order to support such features, a graph processing platform typically involves a stack of options (See Figure 1). The top layer is the data model layer to efficiently represent the graph for the desired analysis. The processing paradigm layer controls the efficiency of implementation of graph-operations. The bottom infrastructure layer controls the productivity of implemented graph-operations.

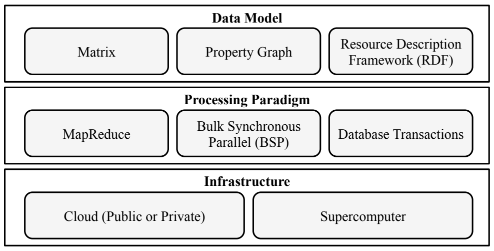


Figure 1: Graph processing systems involve multiple layers of stack in processing graph operations.

Since choices in each layer of the stack is independent of each other, a wide variety of options are available for data scientists to perform graph analysis. For instance, Resource Description Framework (RDF) can be processed by MapReduce-based engines [6] or databases [7]. As an example of processing paradigm, SPARQL query can be processed in query engines in either Cloud environments [8] or high-performance computing environments [5]. Due to such a wide variety in the possible combination of choices, evaluating general-purpose graph processing platforms requires deeper understanding of all the layers in the stack.

Therefore, this study presents an extensive empirical evaluation on representative platforms to perform graph analysis for large scale real-world problems. This study evaluates Pegasus, GraphX, and Urika for graph analysis. Pegasus [9] deals with graph-data represented as adjacency matrices, processes using the MapReduce paradigm, and runs in the Cloud environments. GraphX [4] prefers a property graph format, supports Bulk Synchronous Parallel (BSP) paradigm, and runs in the Cloud environments. Urika [5] is a graph processing appliance based upon Cray's XMT architecture, in which a graph is represented using RDF and graph analysis is conducted via transactional queries. For infrastructure, we experiment with (1) an instance of Urika at Oak Ridge National Laboratory, which has 2TB of shared memory in the system, connected with Cray's proprietary interconnect network, and (2) a 65 node cluster in a public Cloud environment. We believe the experimental setting in this study is a representative medium-to-large sized environment for analysis purposes.

This study benchmarks each graph processing platform for three popular graph mining algorithms: degree distribution, connected components, and PageRank. With Pegasus and GraphX, we leverage the source code provided by the authors. For Urika, we use our own implementation of graph algortihms expressed using SPARQL queries. Our approach is a first attempt at implementing iterative graph mining operations in SPARQL. Detailed description of our implementation is available in [10]. This study employs real data sets from [11] such as DBLP, Patents, and Friendster. DBLP has 0.3 million nodes and 1 million edges, Patents has 3.8 million nodes and 16.5 million edges, and Friendster has 65.6 million nodes and 1.81 billion edges.

Our benchmark results indicate that Urika is about 2 times faster than GraphX, and 7 times faster than Pegasus, for calculating degree distribution of Friendster. For connected components over Friendster, GraphX shows 90 times faster execution than Urika and 654 times faster than Pegasus. As for PageRank over Friendster, GraphX demonstrates 8 times faster execution than both Urika and Pegasus. However, achieving desired performance with GraphX requires a substantial effort such as controlling the level of parallelism by the number of concurrent tasks (# of task executors) and the number of total tasks (# of partitions), along with tuning communication overheads.

The remainder of this paper flows as follows: §II describes popular data model to represent graphs. §III provides a brief description of three popular processing paradigms for graph processing. §IV presents our empirical study results, followed by discussion in §V. §VI presents the state of the art in graph processing at large scale. §VII concludes the paper.

II. Representing graphs

Given a graph $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ represents vertices and $\mathcal{E}$ represents edges, three different models are used in practice to represent graph structures; matrix, property graph, and RDF. While the matrix vector model considered in Pegasus and HAMA [12] mainly represents the topology of given graph, property graph and RDF have capabilities to describe properties/attributes of entities (nodes/edges) in the graph. Although property graph and RDF share the same goal to represent graphs with properties, they differ in format and represenation. The rest of this section provides formal descriptions of matrix, property graph and RDF representations.

A. Matrix

A graph $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of nodes and $\mathcal{E}$ is a set of edges, can be characterized by a $|\mathcal{V}| \times |\mathcal{V}|$ adjacency matrix M, where it is defined by:

Definition 1.

$$M(i,j) = \begin{cases} 1 & , if\ \exists e(v_i, v_j) \in \mathcal{E} \\ 0 & , otherwise. \end{cases}$$

Note that $e(v_i, v_j)$ represents an edge connecting v_i and v_j in the definition. Matrix representation efficiently describes the topology of graphs, but provides limited capability to include properties/attributes for nodes and edges.

B. Property graph

A property graph is a directed multigraph, wherein two vertices may share multiple parallel edges. Property graph model is commonly used in graph databases such as *Neo4J* [13], *Titan* [14], *DEX* [15], as well as graph processing systems like GraphX [4]. Formal definition of property graph is given by,

Definition 2. *A property graph is a directed multigraph $G = (\mathcal{V}, \mathcal{E}, T_{\mathcal{V}}, T_{\mathcal{E}})$, where $\mathcal{V}$ is a finite set of nodes, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a finite multi-set of edges, $T_{\mathcal{V}}$ is a finite set of node types, and $T_{\mathcal{E}}$ is a finite set of edge types. Each node is mapped to a node type by a node type mapping function $\phi_{\mathcal{V}} : \mathcal{V} \to T_{\mathcal{V}}$, and each edge is mapped to an edge type by an edge type mapping function $\phi_{\mathcal{E}} : \mathcal{E} \to T_{\mathcal{E}}$. Each node $v_i \in \mathcal{V}$ or edge $e_k(v_i,v_j) \in \mathcal{E}$ has a set of <attribute, value> pairs that describe the properties of the node.*

C. Resource Description Framework (RDF)

The RDF and the SPARQL query languages are two standards recommended by the W3C for representing and querying linked data on the Web. RDF is widely adopted by Semantic Web communities. An RDF collection consists of a set of triples in the form of '$subject, predicate, object$', where *subject* denotes a globally unique resource, *object* denotes either a unique resource or a literal (i.e., a string or a number), and *predicate* denotes a relationship between the subject and object. Also, RDF data can represent a directed, labeled multi-graph, formally,

Definition 3. *An RDF data graph $G = (\mathcal{V}, \mathcal{E}, \mathcal{L}, \pi)$ is a directed, labeled multi-graph, where $\mathcal{V}$ is the set of nodes, $\mathcal{E}$ is the set of directed edges between the nodes in $\mathcal{V}$. $\mathcal{L}$ is the set of edges and associated node labels. π is a labeling function with $\pi : \mathcal{V} \cup \mathcal{E} \to \mathcal{L}$ such that $\forall v_i,\ v_j \in \mathcal{V}$, $v_i \neq v_j$. It holds that $\pi(v_i) \neq \pi(v_j)$.*

III. Processing paradigms for graph processing

This section describes three popular processing paradigms for general purpose graph processing: MapReduce, Bulk Synchronous Parallel, and database transactions. Figure 2 summarizes the differences between processing models of aforementioned processing platforms. Since many graph algorithms are iterative, Figure 2 illustrates the logical flow while processing iterations in each paradigm. Following subsections provide more detailed description for each processing paradigm, with the tools chosen for this study.

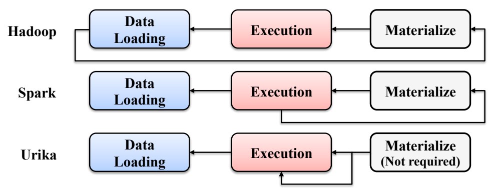


Figure 2: Processing models of each iteration in graph algorithms in Hadoop (Pegasus/MapReduce), Spark (GraphX/Pregel runtime), and Urika (SPARQL). In Hadoop MapReduce, each iteration repeatedly loads data from *disk* and materializes the results into *disk*. Spark, while initial graph is loaded from storage, each iteration is materialized on distributed *in-memory* data structure for the next iteration (Superstep in BSP). Urika supports online transactions. Computation results are available in the system memory after the execution of each iteration.

A. MapReduce-based Processing

The MapReduce programming paradigm, introduced by Google [16] is a data-parallel framework. It allows developers to implement highly scalable and fault-tolerant parallel applications to process large scale data in a distributed shared-nothing environment. Apache Hadoop is an open-source implementation of MapReduce. A MapReduce program involves three phases: map, shuffle, and reduce. In the map phase, each machine reads segments of input key-value pairs from the distributed file system and generates a new set of output key-value pairs (e.g. aggregates from each segment). The shuffle phase receives all the key-value pairs from map phase and sorts different output key-value pairs. In reduce phase, each machine groups the key-value pairs with the same key and stores grouped key-value pairs as a new set of key-value pairs into the secondary storage system.

Pegasus [9] is a notable graph analysis tool, adopting the MapReduce paradigm. Since graph-mining tasks are typically dictated by the adjacency relationships, they involve a large number of matrix manipulations. Pegasus implements distributed versions of matrix multiplication called generalized iterative matrix vector multiplication (GIM-V). The formulation and design of matrix multiplication operations exploit the block-structured nature of matrices to perform its computations and hence can provide scalability. In experiments, it was shown that GIM-V enabled computations on Hadoop with high linear scalability, in addition to handling over 2 billion edges in the graph.

B. Bulk Synchronous Parallel (BSP) Processing

In Pregel [2], Google proposed to use BSP paradigm [17] for graph analysis in order to overcome the limitations of MapReduce paradigm for graph processing: a large memory requirement due to poor locality of memory accesses; too many small jobs due to small computation per vertex; and varying and unpredictable level of parallelism during iterations.

BSP consists of the following steps: (1) dividing the algorithm into a series of super-steps; (2) computing a local update function on the vertex (in parallel across the graph); (3) reading messages from a previous super-step; (4) sending messages to a super-step; and (5) modify the states of other outgoing edges. Pregel implements fault-tolerance by allowing (at every super-step) messages to be checkpointed at a master node that aggregates the local results. When a failure occurs on any of the worker nodes, all the worker nodes proceed from the previous super-step. While Google's Pregel is proprietary, Apache Giraph [18] is an open-source implementation.

GraphX [4] is a graph processing platform that runs on Spark [19], which supports the Pregel-runtime. Spark is a open-source in-memory cluster computing system, which overcomes many deficits of Hadoop MapReduce for iterative algorithms. For instance, Spark maintains the dataset in a distributed in-memory data structure, which can be efficiently reused by iterative algorithms. In contrast, Hadoop only allows storing the data back into secondary storage in every iteration, which incurs inefficiency for iterative algorithms. Spark, because of its in-memory processing capabilities, enables low latency interactive data analysis. Exploiting those features of Spark, GraphX conveniently associates graph processing systems with Extract, Transform, and Load (ETL) process of a graph from secondary storage, though it is known to provide less efficient graph processing than special-purpose graph processing systems like GraphLab [20], and Giraph [18].

C. Database Transaction Processing

As part of the NoSQL movement [21], graph databases have emerged to overcome the limitations of using relational databases to process graph data structures. Graph databases are designed for optimally storing, retrieving and querying graph data, analogous to how tuples are represented in a relational database. Typically, graph databases can more efficiently handle graph queries (e.g., subgraph pattern matching) compared to relational databases. The major strength of transaction-oriented graph processing paradigm is that it allows processing online transactional queries over graphs using graph query languages. There are two options when it comes to choosing a graph database: (1) RDF triplestores like Jena SDB [22], Sesame [23], Virtuoso [24]. They are graph databases that store RDF triples, which can be queried using SPARQL; and (2) graph databases like Neo4J [13], Titan [14], DEX [15] that are well suited for property graphs. Graph databases also support query languages. (e.g., *Cypher*, *Gremlin*, etc.).

Database transaction systems prefer a uniform latency over a large chunk of shared memory over distributed memory slots [25]. Therefore, a reasonable choice of infrastructure is a large-scale shared memory system such as Urika. Urika from YarcData, a sister-company of Cray, is

Table I: Data sets in this study

Name	Edges	Vertices	Comments
DBLP	1 mil.	0.3 mil.	Computer science collaboration network
Patents	16.5 mil.	3.8 mil.	Citation network among US patents
Friendster	1.81 bil.	65.6 mil.	Friendster online social network

built around a parallel supercomputer architecture, the Cray XMT. Urika is a graph processing appliance that provides both hardware-based solutions for resolving poor locality in memory access, with the emphasis on communication overhead, and an optimized software engine to process SPARQL queries over RDF triples.

IV. Empirical evaluation of platforms

This section presents the empirical results on benchmarking three graph processing platforms, Pegasus, GraphX, and Urika. Table I shows data sets used in the experiments. They are publicly available data sets obtained from the Stanford Network Analysis Project (SNAP) website [11] and widely used by graph analysis community. Evaluation results can be found in §IV-A (degree distribution), §IV-B (connected components), and §IV-C (PageRank). As for experimental environment, Table II shows the configuration for Hadoop and Spark used in this study, as deployed on a cloud instance in Amazon Web Services. We employed a 65 node cluster for one master node and 64 worker nodes, with the *m3.2xlarge* instance type. A VM instance of *m3.2xlarge* type offers 8 CPUs and 30GB of RAM. Hadoop and Spark are configured according to general guidelines from [26], [27]. As mentioned earlier, Urika is a shared memory system. Urika uses 64 Threadstorm processors, each of which provides 128 hardware threads. Urika can support up to 512 TBs of shared memory with a single memory address for the processors. We used an instance that installed at ORNL, which has 2 TBs of shared memory. Although the system architectures are fundamentally different from each other, they all have the same amount of memory (2 TBs).

A. Degree distribution

The degree of a node in a graph is the count of the number of edges associated with that node.The degree distribution is the probability distribution of the node-degrees over the whole graph. More specifically, the *degree distribution* $P(k)$ of a graph is defined as the histogram of nodes in the graph with degree k. Degree distribution is an important statistical metric of a graph [28], [29]. However, it is a non-trivial question to calculate degree distribution in a timely and scalable manner, when the graph scale is large [30] and the structure varies over time [31].

With SPARQL, degree distribution can be computed by a single query, as standard SPARQL natively supports the COUNT aggregation function. The query for computing degree distribution of graphs is presented in Listing 1. It is a nested SPARQL query where the subquery computes degree of each node by counting the number of edges connected to each node considering both incoming and outgoing directions, then the main query counts the frequencies of each degree.

```
SELECT ?degree (COUNT(?degree) AS ?count)
WHERE
{ {     SELECT (COUNT(?s) AS ?degree)
        WHERE
        {
            { ?s <url:edge> ?outgoing. }
            UNION
            { ?incoming <url:edge> ?s. }
        }
        GROUP BY ?s } }
GROUP BY ?degree
ORDER BY ?degree
```

Listing 1: The SPARQL query for retrieving degree distribution of a graph

As for the MapReduce implementation of degree distribution, we used the implementation in Pegasus [9], which consists of two phases of MapReduce jobs (refer to Listing 2). In the first pass, it reads the input file as a list of edges, considering the source node as the key, and the destination node as the value. The output of the first pass is the degree of each node. In the second pass, it reads the output from the first pass to count the number of nodes with the same degree and then, group by degree. GraphX implements the same MapReduce algorithm on Spark.

```
//Pass 1 group by node id
INPUT: edge list
OUTPUT: <Key(node_id), Value(degree)>
Map (Key k, Value v) begin
    output (k, v)
    output (v, k)
end
Reduce (Key k, Value v[1...m]) begin
    for each k do
        degree = sum[v]
    end
    for each k do
        output (k, degree)
    end
end
//Pass 2 group by degree
INPUT: <Key(node_id), Value(degree)>
OUTPUT: <Key(degree), Value(count)>
Map (Key k, Value v) begin
    output (v, 1)
end
Reduce (Key k, Value v) begin
    for each k do
        count = length (v)
    end
    for each k do
        output (k, count)
    end
end
```

Listing 2: Pseudocode of degree distribution in MapReduce

Figure 3 shows evaluation results for degree distribution. As Urika is designed for on-line transactional processing, degree distribution is efficiently processed as on-line queries. This advantage is seen in the graph that shows Urika outperforming other considered processing paradigms. Degree distribution in Hadoop MapReduce-based Pegasus has two phases. The second phase reading the output from

Table II: Hadoop and spark configuration that may impact on the performance.

Parameter	Value	Comments
		Hadoop
Hadoop version	2.4.0	CDH 5.1.0 is used.
mapreduce.map.memory.mb	1536	
mapreduce.reduce.memory.mb	3072	
dfs.replication	3	HDFS replication factor
dfs.blocksize	268435456	256MB of HDFS block
yarn.nodemanager.resource.cpu-vcores	8	a total of 8 tasks or 8 spark executors per node
yarn.nodemanager.resource.memory-mb	25943	a total of 25GB of memory per node for yarn. 5GB of memory is saved for OS.
		Spark
Spark version	1.0.0	CDH 5.1.0
SPARK_EXECUTOR_INSTANCES	128, 192	Across cluster, a total of 128 (2×64) and 192 (3×64) executors to distribute computation
spark.executor.memory	6g, 4g	6g for 128 task executors, 4g for 192 task executors
spark.storage.memoryFraction	0.5	en executor reserves 50%of memory for distributed in-memory data structure
spark.shuffle.memoryFraction	0.3	an executor reserves 50% of memory for shuffle
spark.shuffle.consolidateFiles	True	We employed ext4 filesystem for the storage mounted for Spark.
spark.rdd.compress	True	we compressed RDD with LZFCompressionCodec
deploy mode	yarn-client	it deploys a spark job with YARN

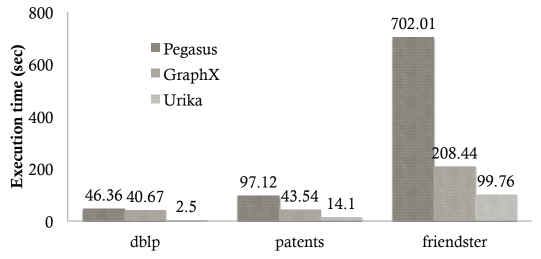


Figure 3: Degree distribution

the first phase - both of which happen at the file system level involving multiple attempts to read and write from disk. Therefore, Pegasus shows the slowest performance for this operation. GraphX uses distributed in-memory data structure, which prevents iterative jobs from the inefficiency of looping between loading and flushing intermediate data to file system. Thus, it is natural that GraphX performs faster than Pegasus. However, the tested Urika instance uses 64 Threadstrom processors, each of which supports 128 hardware threads. Thus, the level of parallelism for Urika was much higher than GraphX (64x2x8=1024 or 64x3x8 =1536 concurrent tasks) in our environments, which made Urika outperform GraphX for computing the degree distribution. This result indicates that shared-memory based massively parallel architecture may perform the best for basic-graph pattern retrieval or finding summary statistics of entities in a graph.

B. Connected components

Given an undirected graph $G(V, E)$ with n nodes and m edges, a connected component is a maximal set of nodes that can reach each other through paths in G. Computing all connected components of G is a fundamental graph problem and can be solved efficiently on a sequential machine using $O(n + m)$ time. However, it is nontrivial to design an efficient parallel algorithm for large number of nodes n and edges m on parallel machines. According to a recent study by Qin *et. al.* [32], challenges stem from (1) non-scalable communication overhead, $O(\log(n)(m+n))$ communication cost for each iteration, and (2) impractically large working set for a single machine, $O(n)$ memory for each machine to hold a whole connected component in memory.

```
FOR EACH NODE ?s
UPDATE {?s <labeled> ?prev.} WITH {?s <labeled> ?next}
IN TEMP GRAPH
WHERE
{ { //new label ?next is decided as follows
        SELECT ?s (MIN(labels(?s)) AS ?new)
        WHERE
        {
            labels(?s) ← {label of ?s} UNION
              {labels of ?s's adjacent nodes}
        }
        GROUP BY ?s
} }
```

Listing 3: The SPARQL query for retrieving degree distribution of a graph

Let us detail how to design a SPARQL query for finding connected components. Since SPARQL query for finding connected components is not publicly available, we developed our own sequence of queries, which is planned to be open to public in the near future. Let us assume that we are computing connected components for a graph G. Initially, an unique integer label l_{n_i} is assigned to each node n_i, which can be done by adding triples such as (n_i, <labeled>, l_{n_i}) in a newly created temporary graph, in which intermediate and final results of the computation are stored. Once the temporary graph is prepared, the algorithm iteratively updates the temporary graph using G as follows; for each node in the G, associated label in the temporary graph is updated as the minimum label among all of labels of its adjacent nodes in G, including its own label. For example, if a node's label is 6, and its adjacent nodes' labels are 4, 5, and 9, the node's label is updated to be 4, as 4 is the minimum among all four numbers. To enhance reader's understanding, we illustrate the pseudo-SPARQL query for updating the graph at each iteration in Listing 3. In this approach, it should be noted that even though the same task is performed at every node, the task for each node is independent of the tasks executed in parallel at the other nodes.This approach maximizes Urika's capability of simultaneously processing each node in parallel.

The iteration stops when there are no more label updates. As a result, all nodes in the same connected component will have the same label, and the number of distinct labels will be the number of graph components. The algorithm requires $O(n)$ of space complexity, where n is the number of nodes, as it requires assign labels to every node. The time complexity of the algorithm is $O(n \times d)$, where d is the number of iterations until it converges. In the worst case scenario, the maximum number of iterations d required is the diameter of the graph.

```
def run[VD: ClassTag, ED: ClassTag](graph: Graph[VD, ED]):
Graph[VertexId, ED] = {
    val ccGraph = graph.mapVertices {case (vid, _)=> vid }
    def sendMessage(edge: EdgeTriplet[VertexId, ED]) = {
        if (edge.srcAttr < edge.dstAttr) {
            Iterator((edge.dstId, edge.srcAttr))
        } else if (edge.srcAttr > edge.dstAttr) {
            Iterator((edge.srcId, edge.dstAttr))
        } else {
            Iterator.empty
        }
    }
    va initialMessage = Long.MaxValue
    Pregel(ccGraph, initialMessage, activeDirection =
EdgeDirection.Either)(
        vprog = (id, attr, msg) => math.min(attr, msg),
        sendMsg = sendMessage,
        mergeMsg = (a, b) => math.min(a,b))
} // end of connectedComponents
```

Listing 4: Connected components in Pregel

Listing 4 shows the code snippet of connected components within GraphX version of Pregel runtime. In the Pregel or Bulk Synchronous Parallel (BSP) paradigm, the vertex program is analogous to a map task in MapReduce paradigm. At the end of each superstep, vertices send messages as specified in *sendMessage* function. It distributes the message to all connected vertices and a message is calculated in vertex program, the smallest attribute amongst connected vertex, or component ID in this case. After sending the messages, each vertex combines the message according to *mergeMsg* function, which simply selects the minimum amongst all received messages. The primary benefit of this processing paradigm is to reduce communication overhead. All the vertices partitioned in one machine collectively exchange updated message to vertices in other machines, in each superstep. In our experiments, supersteps are less than 50 for both connected components and PageRank.

Although there are newer methods to find connected components using the MapReduce paradigm [33], we used the matrix-based implementation in Pegasus [9]. Verisimilar to the method used in the Pregel implementation, finding connected components in Pegasus consists of three steps: (1) each node sends its current component ID to neighbors; (2) for each node, find the minimum value amongst current component ID and all received component ID; and (3) update component ID according to the results from the step (2). The major differences between MapReduce and Pregel stem from the method to communicate messages among vertices. The implementation in Pegasus calculates for each partition stored in each machine and computation from each step is flushed into disk to be used in the next step. However, the Pregel paradigm does not require storing the output from each step into disk. Instead, it continues to remain in the system main memory. Excepting for this difference, the conceptual footprint of the MapReduce and Pregel implementation are similar to each other. This is why we have not presented the pseudo code for the MapReduce implementation. Readers can refer to [9] for details.

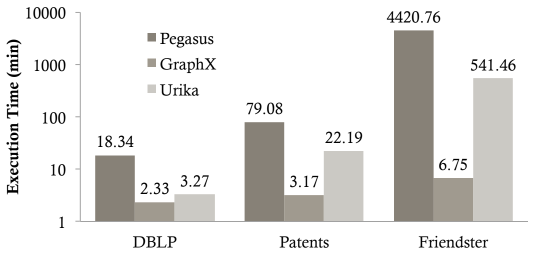


Figure 4: Connected components: note that the y-axis is in logarithmic scale.

Figure 4 shows the evaluation results for connected components. Finding connected components involves multiple iterations and the number of iterations is typically associated with the diameter of given graph. The number of iterations required for convergence increases with the size of graph. GraphX outperforms Hadoop-based approach by several orders of magnitude as size increases. Also, GraphX uses Pregel runtime, that is, BSP paradigm, for connected components.The BSP paradigm reduces the communication overheads in updating each vertex and, in each superstep, only vertices with updated results from prior superstep is involved in future computations. Also, as the values converge, the number of vertices participating in the computation decreases. In contrast, each iteration in Urika essentially performs the same amount of computation. Such a characteristic made GraphX extremely efficient in processing connected components for large graphs like Friendster data set. However, we believe that processing efficiency will vary according to the topology of a graph. Hence, reader may have to exercise caution in extrapolating our execution times for their own graphs in each processing paradigm.

C. *PageRank*

PageRank is a one of the most widely used graph algorithms, and it is designed to measure the importance of nodes in a graph. The algorithm is originally designed to compute rankings of web pages that are linked with hyper links, but it is now generally exploited in many other graph analysis tasks. The PageRank algorithm computes the significance of a node by looking at the number incoming links from other important nodes. By that definition, nodes connected to nodes with higher PageRank score will tend to have a higher PageRank value. PageRank of a node is recursively defined, and it is dependent on the number and PageRank scores of all nodes from its incoming links. PageRank outputs a probability distribution, which represents the likelihood of a random walker, will arrive at each node. *PageRank* is computed by, $\vec{r} = \alpha P^T \vec{r} + (1-\alpha)\frac{1}{N}\vec{e}$, 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 in a similar way we computed connected components, as shown in Listing 5. We again highlight that this is based upon our own query design. It assigns initial PageRank score $1/N$, where N is the number of nodes in the graph, to all nodes and creates a temporary graph to save the intermediate score assignments. Then, SPARQL query updates the temporary graph iteratively. The iterative update of each node's PageRank score is computed by querying its adjacent node's PageRank score. When PageRank score converges, the iteration stops.

```
//Prepare to compute the next PageRank scores
UPDATE ALL {?s <uri:nextPR> ?score}
WITH {?s <uri:prevPR> ?score}
IN TEMP GRAPH;

//Compute new PageRank scores
//and insert into the temporary named graph
FOR EACH NODE ?s
INSERT {?s <uri:nextPR> ?score} IN TEMP GRAPH
WHERE
{ { //df is typically set to 0.85,
    //N is number of nodes in graph
    SELECT ?s ((SUM(?val1/?val2) * df)+(1-df)/N AS ?score)
    WHERE
    {
        {?x <uri:edge> ?s.}
        { TEMP GRAPH {
            ?x <uri:totalDegree> ?val1.
            ?x <uri:prevPR> ?val2
          }
        }
        UNION
        {?s <uri:edge> ?x.}
        { TEMP GRAPH {
            ?x <uri:totalDegree> ?val1.
            ?x <uri:prevPR> ?val2
          }
        }
        //considering all edges as bidirectional edges
    } GROUP BY ?s
} }
```

Listing 5: The pseudo-SPARQL query for iteratively updating PageRank scores; Note that ***df*** should be replaced with actual a damping factor and ***N*** with the number of nodes

```
// Initialize the pagerankGraph with each edge attribute
// having weight 1/outDegree and
// each vertex with attribute 1.0.

//Define the three functions to implement PageRank
//in the GraphX version of Pregel

def vertexProgram(
    id: VertexId, attr: (Double, Double), msgSum: Double):
        (Double, Double) = {
            val (oldPR, lastDelta) = attr
            val newPR = oldPR + (1.0 -resetProb) * msgSum
            (newPR, newPR - oldPR)
        }
def sendMessage(
    edge: EdgeTriplet[(Double, Double), Double]) = {
    if (edge.srcAttr._2>tol){
    Iterator((edge.dstId, edge.srcAttr._2 * edge.attr))
    } else {
        Iterator.empty
    }
}
def messageCombiner(a: Double, b: Double): Double = a + b

//The initial message recieved by all vertices in PageRank
val initialMessage = resetProb / (1.0 -resetProb)

//Execute a dynamic version of Pregel
Pregel(pagerankGraph, initialMessage, activeDirection =
EdgeDirection.Out)(
    vertexProgram, sendMessage, messageCombiner)
    .mapVertices((vid, attr) => attr._1)
```

Listing 6: PageRank in Pregel

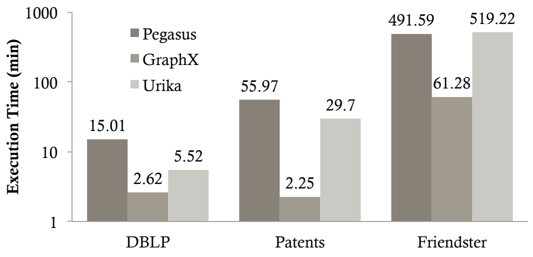
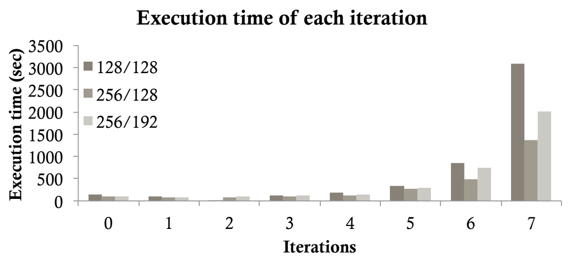


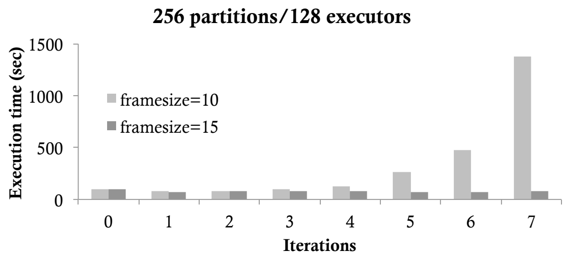
Figure 5: PageRank: y-axis values are in logarithmic scale. As for GraphX, we employed different number of graph partitions for each data set, 64 partitions for DBLP, 128 partitions for Patents, and 320 partitions for Friendster.

Listing 6 shows a simplified code snippet for PageRank algorithm in Pregel, where initialization part is omitted due to the space constraints. Similar to finding connected components, we have *vertexProgram*, *sendMessage*, and *messageCombiner*. In this implementation, we note that each vertex will send a message only if the difference between prior message and current message is larger than given tolerance value. Our tolerance value was 0.0001. For Pregel experiments, we used a dynamic PageRank, which runs until attributes assigned to each vertex converges. Also, reader may note that the version of PageRank implemented in GraphX is a non-normalized PageRank, which is different from conventional PageRank algorithm. Conventional PageRank algorithm initializes the attribute of each node with a normalized value of $1/N$, where N is the number of nodes, as with our SPARQL query and Pegasus. Otherwise, the Pregel implementation will behave similar to the SPARQL query approach. As for PageRank included in Pegasus, the high level description is the same as the Pregel implementation.

Figure 5 shows evaluation results for PageRank. We find that GraphX, which uses the Pregel paradigm, outperforms other two platforms. Compared with Pegasus, Urika shows faster execution time for small or medium data sets like DBLP and Patents. It should be noted that SPARQL is originally designed for finding patterns, instead of performing algorithmic operations. However, we find that Urika can



(a) For PageRank over friendster, execution time of each Pregel iteration on GraphX, a Pregel runtime on Spark. In the legend, X/Y represents X partitions over Y task executors and so on.



(b) For PageRank over Friendster, optimizing shuffle behavior of Pregel runtime: framesize is the size of buffer in MB to transfer computed results.

Figure 6: Optimally configuring GraphX depends on the input graph and desired analysis, together with infrastructure. (a) shows the number of partitions and the number of task executors should be carefully tuned, which determines the level of the parallelism. (b) shows that controlling communication overheads will be critical to the performance.

process algorithmic operations based on SPARQL queries, faster than or equal to MapReduce-based approach. We believe that the shared-memory hardware of the Urika system, allowed us to overcome the limitation of transaction-based processing paradigm. We attempted to create a SPARQL query that manipulates the graph as large as possible to maximally utilize the memory capability. For GraphX results, the execution time of Patents is faster than that of DBLP partly because of different number of partitions. We will discuss this behavior in detail in regard to the performance implication of the number of partitions for each data set in the following discussion section.

V. Discussion: challenges in using each platform

Each platform has its advantages and disadvantages and offers opportunities for optimization in different ways. For instance, Pegasus on Hadoop requires extensive study to find optimal configuration for workloads in different system infrastructures [34].Urika, on the other hand, is an appliance that bundles software and hardware. Users are not allowed to tune system configuration. The only available option is to design efficient queries that best utilizes a shared-memory architecture. While we reserved this query optimization task on Urika as future work, we discuss the challenges in tuning GraphX in this section.

Figure 6 shows that it is tricky to find the best configuration for Pregel runtime on Spark, for each algorithm on each data set. As for small data sets like DBLP and Patents, it is relatively less critical with our 65-nodes cluster in the Cloud environments. Although we have already noticed that PageRank over 64 partitions DBLP (0.3 million nodes) can perform slightly slower than PageRank over 128 partitions Patents (3.8 million nodes), for PageRank over Friendster data set, we suffered exponentially increasing execution time while iterating. For the Friendster data set, PageRank on GraphX had to go through 51 iterations (i.e., it did not converge). Exponentially increasing execution time delayed convergence. In order to perform analysis within reasonable time, we choose a set of good configuration parameters through trial-and-error.

Figure 6(a) shows that the level of parallelism can create macro-level effects in the execution time of each iteration. Initially, we configured two task executors per machine and the number of graph partitions matched the number of task executors. However, we faced exponential increase in the execution time per iteration. We were not able to finish PageRank over Friendster within a week. We conjectured that the number of partitions might be too small since the number of reduce tasks matches with the number of partitions in GraphX. Then, we increased the number of partitions while the number of task executors remained the same, which yielded a substantial performance gain. However, the execution time increased as the algorithm progressed. We tried to increase the number of task executors for the same number of partitions, but we were unable to see improvements in execution time. Consequently, we ended up optimizing communication overheads.

We speculated that communication overhead between map tasks and reduce tasks may be causing delays in convergence. Following [35], we started adjusting the buffer size. We increased the buffer size by 50% for the shuffle phase over the default buffer size. The shuffle buffer size is responsible for the communication between mappers and reducers. Figure 6(b) shows that, with the framesize of 15MB, we achieved a constant execution time for each iteration. After the buffer size increase, we managed to execute PageRank algorithm over Friendster data set in 61.28 minutes on average, as shown in Figure 5. We believe that a systematic and automatic approach to find optimal configuration is possible, but requires more attention.

VI. Related work

While knowledge discovery from graph data has been studied for a while [36], discovery from integrated information networks has seen a revitalization after the adoption of the internet, social-networks and proliferation of social-media. Recent surveys on managing and mining graph data [37] and graph algorithms [38] along with the study of laws and generating models [39] have helped design of tools such as SUBDUE [40], gSpan [41] , OddBall [42], Pegasus [9], NetworkX [43], and GraphLab [20].

Pegasus on Hadoop [9] and GraphLab, which can be instantiated on the Cloud environments, are restricted to mining homogenous graphs. On the other hand, graph-databases such as Neo4j [13], Titan [14], and Trinity [44] can house and retrieve massive heterogeneous graphs on commodity hardware, which can be useful to describe real world problems [37], [45]. However, graph databases have limited data mining capabilities, in comparison to tools such as Pegasus or GraphLab. This is primarily because of the difficulty in expressing graph-theoretic algorithms using query languages [46].

The scalability of graph processing tools is of interest in the data analysis community in order to handle large scale graphs. Infact, researchers at Microsoft performed a similar empirical comparison of Microsoft products that can process large graphs [47]. Other empirical evaluation studies on general purpose graph processing systems include [48], [49]. However, since GraphX is relatively new, none of them include GraphX. Also, none of the earlier studies compare graph databases or triple stores with general purpose graph processing systems, with respect to graph mining capabilities. Our study performed a benchmark study for open-source tools, which were not discussed in aforementioned Microsoft study. In addition to including GraphX, this study included a specialized supercomputer for graph analysis. To the best of our knowledge, graph appliances such as Cray's Urika had not been benchmarked with graph mining capabilities until now.

VII. Conclusion

This study benchmarked three popular graph analysis platforms, Pegasus, GraphX, and Urika, in order to compare efficacy of each platform. We employed two categories of graph analysis operations: finding statistical metrics of a graph (degree distribution), and algorithmic operations (connected components and PageRank). While each platform presented unique challenges for optimal performance, such as tuning configuration parameters or implementing algorithms to best utilize the platform, this study suggests that each category of platforms may show performance advantage in a certain category of graph analysis operation. Our benchmark results indicated that, as for finding statistical metrics of a graph, database transaction based platforms like Urika may perform better than MapReduce-based platforms like Pegasus or Pregel-based platforms like GraphX. However, Pregel-based platforms demonstrate a few orders of magnitude faster execution time for algorithmic operations than MapReduce-based platforms or database transaction based platforms. In the future, we will evaluate graph pattern matching capabilities of each graph processing platform. In this study, most of graph processing evaluation studies only benchmarked graph mining capabilities. We believe that this study will provide valuable insights for data scientists to decide the suitable graph processing systems for their desired graph analysis.

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).

References

[1] R. Angles and C. Gutierrez, "Survey of graph database models," *ACM Comput. Surv.*, vol. 40, no. 1, Feb. 2008.

[2] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010.

[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 2012.

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

[5] Cray, "YarcData Urika Big Data Graph Appliance," http://www.cray.com/Products/BigData/uRiKA.aspx.

[6] X. Zhang, L. Chen, Y. Tong, and M. Wang, "Eagre: Towards scalable i/o efficient sparql query evaluation on the cloud," in *Proceedings of the 2013 IEEE International Conference on Data Engineering*, 2013.

[7] Apache Jena, https://jena.apache.org.

[8] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald, "Triad: A distributed shared-nothing rdf engine based on asynchronous message passing," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014.

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

[10] S. Lee, S. R. Sukumar, and S.-H. Lim, "Graph mining meets the semantic web," in *6th International Workshop on Data Engineering meets the Semantic Web*, 2015.

[11] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the 2012 IEEE 12th International Conference on Data Mining*, 2012.

[12] Apache HAMA, https://hama.apache.org/.

[13] J. Webber, "A programmatic introduction to neo4j," in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, 2012.

[14] Aurelius, "Titan: distributed graph database," https://github.com/thinkaurelius/titan.

[15] N. Martínez-Bazan, V. Muntés-Mulero, S. Gómez-Villamor, J. Nin, M.-A. Sánchez-Martínez, and 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*. ACM, 2007.

[16] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, 2004.

[17] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, Aug. 1990.

[18] Apache Giraph, http://giraph.apache.org/.

[19] 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*, 2012.

[20] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new parallel framework for machine learning," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2010.

[21] J. Han, E. Haihong, G. Le, and J. Du, "Survey on nosql database," in *2011 6th international conference on Pervasive computing and applications (ICPCA)*. IEEE, 2011.

[22] B. McBride, "Jena: Implementing the rdf model and syntax specification." in *SemWeb*, 2001.

[23] J. Broekstra, A. Kampman, and 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.

[24] O. Erling and I. Mikhailov, *Virtuoso: RDF support in a native RDBMS*. Springer, 2010.

[25] N. Savage, "The power of memory," *Communications of the ACM*.

[26] Hortonworks, "How to plan and configure YARN and MapReduce2 in HDP 2.0," http://hortonworks.com/blog/how-to-plan-and-configure-yarn-in-hdp-2-0/.

[27] Apache Spark, "Tuning spark," https://spark.apache.org/docs/1.0.2/tuning.html.

[28] A. L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[29] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1999.

[30] M. Hay, C. Li, G. Miklau, and D. Jensen, "Accurate estimation of the degree distribution of private networks," in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, 2009.

[31] M. McGlohon, L. Akoglu, and C. Faloutsos, "Weighted graphs and disconnected components: Patterns and a generator," in *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2008.

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

[33] L. Chitnis, A. Das Sarma, A. Machanavajjhala, and V. Rastogi, "Finding connected components in map-reduce in logarithmic rounds," in *Proceedings of the 2013 IEEE International Conference on Data Engineering*, 2013.

[34] H. Herodotou, F. Dong, and S. Babu, "No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, 2011.

[35] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: Exploiting parallelism to scale software routers," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009.

[36] T. Washio and H. Motoda, "State of the art of graph-based data mining," *SIGKDD Explor. Newsl.*, vol. 5, no. 1, Jul. 2003.

[37] C. C. Aggarwal and H. Wang, *Managing and Mining Graph Data*. Springer Publishing Company, Incorporated, 2012.

[38] S. Even, *Graph Algorithms*, 2nd ed. New York, NY, USA: Cambridge University Press, 2011.

[39] D. Chakrabarti and C. Faloutsos, "Graph mining: Laws, generators, and algorithms," *ACM Computing Survey*, 2006.

[40] L. B. Holder, D. J. Cook, and S. Djoko, "Substructure discovery in the subdue system," in *In Proc. of the AAAI Workshop on Knowledge Discovery in Databases*, 1994, pp. 169–180.

[41] X. Yan and J. Han, "gspan: Graph-based substructure pattern mining," in *Proceedings of the 2002 IEEE International Conference on Data Mining*, 2002.

[42] L. Akoglu, M. McGlohon, and C. Faloutsos, "Oddball: Spotting anomalies in weighted graphs," in *Proceedings of the 14th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining - Volume Part II*, 2010.

[43] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference*, 2008.

[44] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013.

[45] Y. Sun and J. Han, *Mining Heterogeneous Information Networks: Principles and Methodologies*, 2012.

[46] J. Celko, *Joe Celko's SQL for Smarties: Advanced SQL Programming*, 4th ed. Morgan Kaufmann Publishers Inc., 2010.

[47] M. Najork, D. Fetterly, A. Halverson, K. Kenthapadi, and S. Gollapudi, "Of hammers and nails: An empirical comparison of three paradigms for processing large graphs," in *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining*, 2012.

[48] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14, 2014.

[49] Y. Lu, J. Cheng, D. Yan, and H. Wu, "Large-scale distributed graph computing systems: An experimental evaluation," *PVLDB*, vol. 8, no. 3, pp. 281–292, 2014.