NScale: Neighborhood-centric Analytics on Large Graphs

Abdul Quamar
University of Maryland
abdul@cs.umd.edu

Amol Deshpande
University of Maryland
amol@cs.umd.edu

Jimmy Lin
University of Maryland
jimmylin@umd.edu

ABSTRACT

There is an increasing interest in executing rich and complex analysis tasks over large-scale graphs, many of which require processing and reasoning about a large number of multi-hop neighborhoods or subgraphs in the graph. Examples of such tasks include *ego network* analysis, *motif counting* in biological networks, finding *social circles*, *personalized recommendations*, *link prediction*, *anomaly detection*, analyzing *influence cascades*, and so on. These tasks are not well served by existing *vertex-centric* graph processing frameworks whose computation and execution models limit the user program to directly access the state of a *single vertex*, resulting in high communication, scheduling, and memory overheads in executing such tasks. Further, most existing graph processing frameworks also typically ignore the challenges in extracting the relevant portions of the graph that an analysis task is interested in, and loading it onto distributed memory.

In this demonstration proposal, we describe NSCALE, a novel end-to-end graph processing framework that enables the distributed execution of complex neighborhood-centric analytics over large-scale graphs in the cloud. NSCALE enables users to write programs at the level of neighborhoods or subgraphs. NSCALE uses Apache YARN for efficient and fault-tolerant distribution of data and computation; it features GEL, a novel graph extraction and loading phase, that extracts the relevant portions of the graph and loads them into distributed memory using as few machines as possible. NSCALE utilizes novel techniques for the distributed execution of user computation that minimize memory consumption by exploiting overlap among the neighborhoods of interest. A comprehensive experimental evaluation shows orders-of-magnitude improvements in performance and total cost over vertex-centric approaches.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*

Keywords

Graph analytics; Cloud computing; Ego-centric analysis; Subgraph extraction; Graph Partitioning; Data Placement; Social networks

Proceedings of the VLDB Endowment, Vol. 7, No. 13

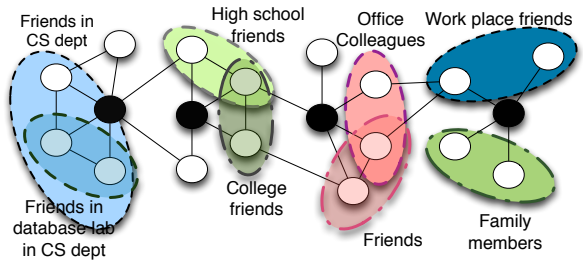


Figure 1: An example of neighborhood-centric analysis: identify users' *social circles* in a social network, i.e., cluster each user's friends into different (possibly overlapping) groups using the network structure and the node and edge contents. Figure shows a small subset of such circles that may be present.

1. INTRODUCTION

Large volumes of graph-structured data are being generated in a range of application domains including social media, Web, bioinformatics, communication, finance, transportation, and many others. There is a growing interest in executing complex analytics over such graph data to get valuable insights about the interconnection structures among the entities, leading to much work on developing distributed graph programming frameworks in recent years.

A large number of complex graph analysis tasks can be viewed as operations on multi-hop local neighborhoods (or subgraphs of local neighborhoods) of a large number of nodes in the graph. For example, there is much interest in analyzing *ego networks*, i.e., 1- or 2-hop neighborhoods, of the nodes in the graph. Examples of specific ego network analysis tasks include identifying *structural holes*, brokerage analysis, counting *motifs*, identifying *social circles* [4] (Figure 1), link prediction and recommendation using Personalized Page Rank, computing *local clustering coefficients*, and anomaly detection.[1] In other cases, there may be interest in analyzing connected/induced subgraphs satisfying certain properties. As an example, we may be interested in analyzing the induced subgraph on users who tweet a particular *hashtag* in the Twitter network. Similarly, we may be interested in analyzing groups of users who have exhibited significant communication activity in recent past. More complex subgraphs can be specified as *unions* or *intersections* of neighborhoods of pairs of nodes; this may be required for graph cleaning tasks like *link prediction* and *entity resolution*.

Prior Graph Processing Frameworks: Several vertex-centric distributed graph processing frameworks have been proposed in recent

[1] See the extended version of the paper [5] for references and more details.

years, including Pregel, GraphLab [3], Apache Giraph, to name a few. In these frameworks, the users write vertex-level programs, that are then executed by the framework in either bulk synchronous or asynchronous fashion. The computation and execution models in these frameworks fundamentally limit the user program's access to a single vertex's state (including its edges and in some cases neighbor IDs). Hence, most of the aforementioned tasks cannot be easily written in these frameworks. For example, to analyze a 2-hop neighborhood to find social circles, one would first need to gather all the information from 2-hop neighbors through message-passing (a huge communication overhead), and then reconstruct those neighborhoods locally. This would require multiple iterations and would not only duplicate the graph processing functionality, but will likely be infeasible because of high memory requirements arising from duplication of state. Customizing these existing frameworks for analytics that require traversing beyond 1-hop neighbors may not be practical or efficient. Giraph++, proposed in a recent work [6], opens up the partition structure to users to optimize execution, however the basic programming framework is not sufficiently different to support the aforementioned analysis tasks.

Secondly, most of these frameworks ignore the issues in extracting relevant portions of the underlying graph that an analysis task may be specifically interested in, and loading it onto distributed memory. In many cases, the user may only want to analyze a subgraph (or several subgraphs) of the overall graph, and may only need access to a subset of the node and edge attributes. Naively loading each disk partition of the graph onto a separate machine may lead to unnecessary distributed communication, especially for distributed graph analytics, where the number of messages exchanged typically increases superlinearly with the number of machines used.

Proposed Approach: In this demonstration proposal, we describe a general, expressive, intuitive, and novel distributed graph processing framework, called NSCALE, aimed at addressing these deficiencies of prior graph processing frameworks. NSCALE is an end-to-end graph processing framework that enables distributed execution of a wide range of querying and analysis tasks including complex neighborhood-centric analytics over large-scale graphs in the cloud. Unlike vertex-centric programming frameworks, NSCALE allows users to write programs at the level of a *subgraph* rather than a *vertex*. More specifically, in NSCALE, users specify: (a) a set of subgraphs or neighborhoods of interest, using a high level specification language, and (b) a *user-specified program* that should be executed on those subgraphs, potentially in an iterative fashion. The user program is written against a general graph API (specifically, BluePrints), and has access to the entire state of the subgraph against which it is being executed. NSCALE execution engine is in charge of ensuring that the user program only has access to that state and nothing more, and thus a program written to compute, say, connected components in a graph, can be used as is to compute the connected components in all the subgraphs of interest.

User programs corresponding to rich and complex analysis tasks may make arbitrary and random accesses to the graph they are operating upon. Hence, one of the key design decisions that we made was to try to ensure that *each of the subgraphs of interest was entirely in memory at one of the machines while it is being executed against*. NSCALE has a novel graph extraction and loading layer (GEL) that aims to achieve that while minimizing the number of machines needed; GEL extracts the relevant data from the underlying graph, employs a cost-based optimizer for data replication and placement, and also attempts to balance load across machines to guard against the *straggler effect*. NSCALE uses a distributed

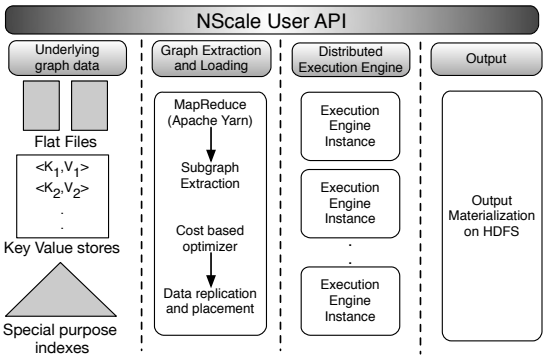


Figure 2: System Architecture

```
ArrayList<RVertex> n_arr = new ArrayList<RVertex>();
for(Edge e: this.getQueryVertex().getOutEdges)
     n_arr.add(e.getVertex(Direction.IN));

int maxlinks = n_arr.size()* (n_arr.size()-1)/2;

// compute #actual edges among the neighbors
for(int i=0; i < n_arr.size()-1; i++)
   for(int j=i+1; j < n_arr.size(); j++)
      if(edgeExists(n_arr.get(i), n_arr.get(j)))
          numEdges++;
double lcc = (double) numEdges/maxlinks;
```

Figure 3: Example user program to compute *local clustering coefficient* written using the BluePrints API. The *edgeExists()* call requires access to neighbors' states, and thus this program cannot be executed as is in a vertex-centric framework.

execution engine that executes user-specified computation on the subgraphs in distributed memory. The execution engine employs several optimizations that reduce the total memory footprint by exploiting overlap between subgraphs loaded on a machine, without compromising correctness.

2. NSCALE: SYSTEM DESIGN

Figure 2 shows the NSCALE system architecture. Users, analysts, applications or visualization tools use the NScale user API to specify the subgraphs of interest and the kernel computation that needs to be run on the subgraphs. NSCALE supports the storage of the underlying graph in a variety of different formats and storage platforms. The Graph Extraction and Loading (GEL) layer is based on MapReduce (MR); it extracts the relevant subgraphs, loads them onto distributed memory across a small number of machines, and instantiates the distributed execution engine. The execution engine instances execute the user-specified subgraph computation on each subgraph in parallel and output the data to HDFS. Next, we briefly elaborate on some of the key components of NSCALE; a more detailed discussion can be found in [5].

2.1 User API

Specification of subgraphs of interest. We envision NSCALE will support a wide range of subgraph extraction queries, including predefined parameterized queries, and declaratively specified queries using a Datalog-based language that we are currently developing. Our current NSCALE prototype, however, supports a limited subset of extraction queries. Specifically, the subgraphs of interest are specified using four parameters: (1) a predicate on vertex attributes that identifies a set of *query vertices*, (2) k – the radius of the subgraphs of interest, (3) edge and vertex predicates to select a subset

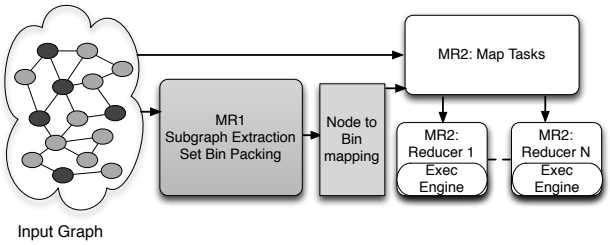


Figure 4: GEL: Graph Extraction and Loading

of nodes and edges from those k-hop neighborhoods, and (4) a list of edge and vertex attributes of interest. This captures a large number of neighborhood-centric graph analysis tasks, including all of the tasks discussed in the Section 1.

Specifying subgraph computation. The user programs to be executed against the subgraphs is specified as a Java program against the BluePrints API [1], a generic API that binds to a large number of graph database backends (e.g., Neo4j) and is used by many open-source graph processing frameworks. Implementing the BluePrints API thus enables the use of existing toolkits and programs over large graphs. Figure 3 shows a sample code snippet of how a user can write a simple local clustering coefficient computation using the BluePrints API. The subgraphs of interest here are the 1-hop neighborhoods of all vertices.

2.2 GEL: Graph Extraction and Loading

Unlike prior frameworks, the graph extraction layer forms a major component of the overall NSCALE framework. From a usability perspective, it is important to provide the ability to read the underlying graph from the persistent storage engines that are not naturally graph-oriented. However, more importantly, partitioning and replication of the graph data are more critical for graph analytics than for analytics on, say, relational or text data. Graph analytics tasks, by their very nature, tend to *traverse* the graph in arbitrary and unpredictable manner. If the graph is partitioned across a set of machines, then many of these traversals are made over the network resulting in a significant performance hit. Hence, in NSCALE, we have made a design decision to avoid distributed traversals altogether by replicating nodes and edges sufficiently so that every subgraph of interest is fully present in at least one partition.

We built the GEL module as a 2-stage MapReduce (MR) job over YARN. Figure 4 shows the overall GEL architecture. Figure 5 shows an example original graph with four query-vertices (orange nodes) that the user is interested in, the subgraph extraction parameters, and the four subgraphs of interest. The output of GEL is a set of partitions (also called *bins*) such that each subgraph of interest is fully contained within at least one partition. The partitions may overlap and further, vertices/edges that are not part of any subgraph of interest are not present in any of the partitions.

The first stage of the MR job (JOB1) reads the input graph as well as an auxiliary input to each mapper that specifies the subgraph extraction query. The mappers of JOB1 filter the underlying graph data based on the predicates, whereas the JOB1 reducer constructs the subgraphs of interest. A key challenge here is to minimize the number of partitions by exploiting the overlap between subgraphs of interest, while ensuring that the subgraphs are evenly distributed across the partitions. This problem is a generalization of the *set bin packing* problem, itself a generalization of the standard *bin packing* problem (both of which are NP-Hard). We have developed a collection of heuristics to solve this optimization problem; further details

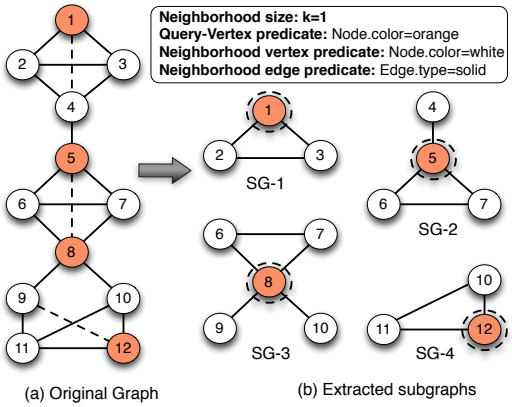


Figure 5: Subgraph Extraction

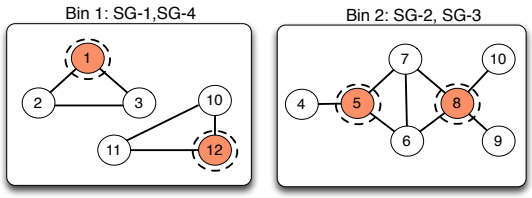


Figure 6: A *packing* of subgraphs into 2 bins

along with extensive experimental evaluation can be found in [5]. Figure 6 shows an example packing of subgraphs to bins for the subgraphs extracted in Figure 5 using a *shingle*-based heuristic that packs subgraphs with high overlap together to improve utilization of the bin capacity. The final output of JOB1 is a mapping of the relevant vertices and edges to partitions (bins).

The second stage MR job (JOB2) takes as input the original graph data on HDFS and the output of JOB1 as an auxiliary input for each mapper. The number of reducers in JOB2 is equal to the number of bins (M) computed by JOB1 required to hold the required subgraphs in memory. The mappers shuffle the required graph data using the mapping provided by JOB1 onto M reducers. The YARN framework also distributes the execution engine as a runtime library and the user specified computation to each reducer. The reducer instantiates the runtime and hands over the graph data to it in memory. The YARN framework thus provides transparent data and computation distribution, and fault tolerance for NSCALE.

2.3 Distributed Execution Engine

The distributed execution engine runs as a runtime library inside the reducers of MR2. At each machine, a thread pool is utilized to execute the user programs on the subgraphs of interest in parallel. If there are more subgraphs of interest than the size of the thread pool (T), then the execution proceeds in a batched fashion, with each batch executing the user programs on T or fewer subgraphs of interest. To ensure than a thread only sees the vertices and edges contained in the subgraph that is assigned to it, we associate *bitmaps* with the vertices and edges that indicate which subgraphs they belong to. These bitmaps are reset at the beginning of each batch accordingly. Figure 7 shows an example bitmap assignment for the partitioning shown in Figure 6.

3. EXPERIMENTAL RESULTS

We have experimentally evaluated NSCALE, deployed on a 16-node Apache YARN cluster, on a variety of real-world datasets ranging in size from 3M nodes/10M edges to 428M nodes/1.4B

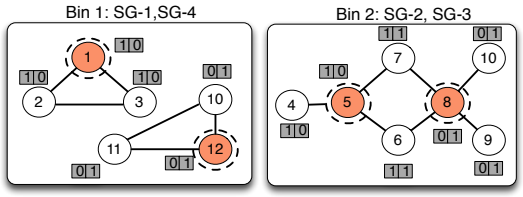
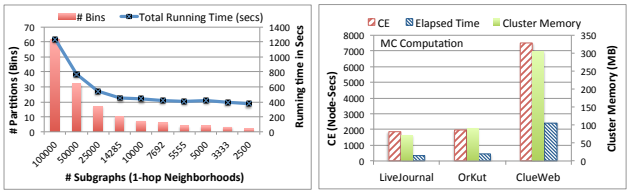


Figure 7: Bitmap based parallel execution



(a) (b)

Figure 8: (a) End-to-End running times and numbers of partitions required for different number of subgraphs; (b) Scalability: NSCALE performance on large graphs.

edges (Clue Web Graph). Table 1 compares the performance of Apache Giraph on YARN with NSCALE for *triangle counting on 1-hop neighborhood*, and *personalized page rank on 2-hop neighborhood* for different datasets. We see that for these two neighborhood-centric analysis tasks, Giraph does not scale to larger graphs – it quickly runs out of memory (OOM) and does not complete (DNC) the computation. Even for smaller graphs, depending on the type of application and the size of neighborhood, NSCALE performs 3X to 5X times better in terms of $\mathcal{CE}$, the computational effort in node-secs (the total sum of the time taken in seconds across the nodes in the cluster) and consumes much less (up to 6X) total cluster memory. Figures 8(a), and 8(b) show that NSCALE scales well both as the #subgraphs extracted increases, and for datasets of different sizes. A more comprehensive discussion can be found in [5].

4. DEMO PLAN

Demo Setup. We will deploy NSCALE on a private 16 node cluster that runs Apache YARN (MRv2) with 15 data nodes and 1 resource manager. Each data node has 2x 4-core Intel Xeon E5520 processors, 24GB RAM and 3x 2 TB disks. The resource manager has 2x 6-core Intel Xeon X5650 processors, 48GB RAM and 3x 2TB disks. We will have different datasets ranging from graphs with a few million nodes and edges up to several 100s of millions nodes and few billion edges in the form of edge lists stored on HDFS on the NSCALE cluster. For comparison with a pure vertex-centric approach we will run Apache Giraph using the same data from HDFS on the same cluster. In order to visualize the performance we will use a web console to connect to the cluster and show the progress of the analysis tasks submitted and the output generated on HDFS by the two systems.

Demo Conduct. We will connect to the cluster remotely and demonstrate NSCALE's end-to-end functionality using different applications scenarios such as local clustering coefficient computation, motif counting (triangles, feed-forward loops, etc.), determining *weak ties*, *link prediction*, *recommendation using personalized PageRank*, and several other neighborhood-centric graph applications. To compare performance with Apache Giraph, we will run several queries on different datasets, with different size neighborhoods, and number of query-vertices. We give an example walk through for a

	Triangle Counting (1-hop)				Personalized PageRank (2-hop)				
Dataset	NSCALE		Giraph			NSCALE		Giraph	
	$\mathcal{CE}$ (Node-Secs)	Mem Reqd (GB)	$\mathcal{CE}$ (Node-Secs)	Mem Reqd (GB)	#Query Vertices	$\mathcal{CE}$ (Node-Secs)	Mem Reqd (GB)	$\mathcal{CE}$ (Node-Secs)	Mem Reqd (GB)
EU	264	15	1012	27	3200	52	3	782	18
N'Dame	477	17	1518	31	3500	119	9	1058	32
Google	663	25	1978	36	8750	786	31	DNC	OOM
WikiTalk	822	21	DNC	OOM	12000	3450	79	DNC	OOM

Table 1: Baseline Comparisons

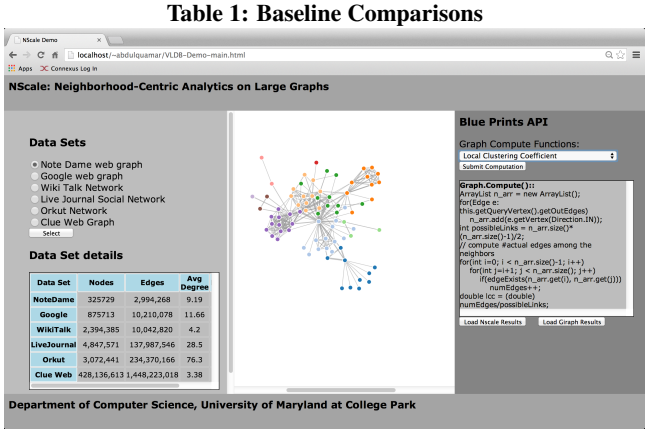


Figure 9: NSCALE: Web DashBoard

neighborhood-centric application on a graph data set.

Example walk through. Users will be able to choose a social network or a graph data set and and the neighborhood-centric application, e.g., personalized page rank computation. The job would then be submitted to both NSCALE and Giraph which would then compute the personalized page rank with respect to a set of query-vertices in their 2-hop neighborhood, and for each query-vertex, display the top k nodes that are closest in rank to the query vertices.

Web DashBoard. We have built a web based interface (Figure 9) for the demo, which will enable users to submit pre-specified computations on different graph datasets and see results as they become available. These choices will be passed to an underlying script that will submit the analysis tasks to both the systems on the cluster. The VLDB attendees will be able to play around with different graph analysis tasks, vary parameters using the web interface, and also be able to see different subgraph computations written using the BluePrints API and compare them with the vertex programs written for Apache Giraph for the same graph analysis tasks.

Acknowledgments: This work was supported by NSF under grant IIS-1319432.

5. REFERENCES

[1] Blueprints API: https://github.com/tinkerpop/blueprints/wiki.

[2] J. Huang, D. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 2011.

[3] Y. Low et al. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *PVLDB*, 2012.

[4] J. McAuley and J. Leskovec. Learning to Discover Social Circles in Ego Networks. In *NIPS*, 2012.

[5] A. Quamar, A. Deshpande, and J. Lin. NScale: neighborhood-centric large-scale graph analytics in the cloud. *CoRR*, abs/1405.1499, 2014.

[6] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "Think Like a Vertex" to "Think Like a Graph". *PVLDB*, 2013.