Using Storm to Perform Dynamic Egocentric Network Motif Analysis

Martin Harrigan, Pádraig Cunningham and Lorcan Coyle
Clique Research Cluster
University College Dublin, Dublin 4, Ireland
Email: {martin.harrigan, padraig.cunningham, lorcan.coyle}@ucd.ie

***Abstract*—In network analysis the ability to characterize nodes based on their attributes and surrounding network structure is a fundamental problem. For example, in financial transaction networks, it allows us to identify typical and anomalous behaviour – important for uncovering fraudulent behaviour. Egocentric network motif analysis is a counting algorithm that tackles this problem – although it is a computationally expensive algorithm. Fortunately, it is inherently parallelizable – each node in the network can be characterized independently of all others. In this paper, we use the distributed stream-processing system *Storm* to perform node characterization in large dynamic networks. We report on the resources required within the Amazon Web Services (AWS) cloud computing platform in order to support this type of analysis on two real-world datasets from the financial domain. This approach allows us to analyze networks that are several orders of magnitude larger than could be tackled with alternative, non-distributed approaches. Our approach also enables live analysis, by treating datasets as streams (as opposed to depending on an offline, batched analysis).**

I. Introduction

Social network analysis is an important field of research, useful for uncovering the relationships and interactions between members of social systems (*e.g.* friendship networks, communication patterns in telecommunication networks, or financial transaction networks). The field has grown in recent years due in part to the availability of large scale computing resources, which is important because the analysis of large-scale social networks tends to be computationally expensive.

A social network comprises a graph containing nodes, which are interconnected by edges. An egocentric network comprises an ego, or individual node in a network, the nodes connected directly to it by edges, and potentially further nodes connected to them. Typically there is a one or two hop bound on the distance from the ego. The egocentric network represents the local view of an ego in a network. It also provides characteristic information about the ego itself. For example, consider a bank account in a financial transaction network. If the account is typical, its egocentric network is structurally similar to a large class of other egocentric networks from the same transaction network. On the other hand, the account may be involved in questionable or undesirable behaviour from a regulatory persecutive – *e.g.* the practice of *smurfing*, where large transactions as split into multiple smaller transactions, each of which is below a limit above which financial institutions must report. Assuming the incidence of smurfing is relatively low, the account's egocentric network should be exceptional.

We can analyze egocentric networks independently, although there are some challenges in distributing the workload efficiently. The problem becomes more complicated when we consider dynamic networks, where the network structure is constantly evolving, *i.e.* nodes are entering and leaving the network and edges are appearing and disappearing. In this case we need to constantly update our characterizations of the egocentric networks over time.

There are practical limitations regarding the size of dynamic network we can handle. However, two recent advances have enabled us to tackle large dynamic networks with millions of edges. The first is the availability of cloud computing platforms, for example, Amazon Web Services (AWS)[1]. Amazon provide access to pools of computing resources and services on an ad-hoc basis. The second advancement is the development of distributed stream-processing systems, for example, Storm[2]. These systems act as a bridging model between a cluster of physical or virtual machines and an abstraction, available to us as programmers, to distribute the processing of streams of data.

This paper has two key contributions. Firstly, we present a system that is deployed on Amazon's AWS platform that uses Storm to perform large-scale dynamic network analysis (specifically, *network motif analysis* [11], [18]). Secondly, we evaluate this system in terms of its performance relative to the level of resources provisioned. We analyze two real-world networks from the financial domain with approximately 10 million edges each. The size of these networks far outstripped our processing ability as described in previous work [5]. However, distributed stream-processing systems allow us to utilize clusters of commodity machines and distribute the workload efficiently.

This paper is organized as follows. In Sect. II we review the state of the art as it relates to network motif analysis and distributed stream-processing systems. In Sect. III we introduce egocentric network motif analysis as it applies to dynamic networks. Sect. IV describes our implementation using the Storm platform. In particular, we describe the components and how they interact to maintain the counts of the network motifs

[1]Amazon's AWS website is here: http://aws.amazon.com/
[2]Storm's website is here: https://github.com/nathanmarz/storm/

over time. We analyze two real-world datasets in Sect. V and report on our results in Sect. VI. Finally, we conclude in Sect. VII and discuss some future work.

II. Related Work

This paper builds on earlier work by the authors that uses a combination of *static* network motif analysis at the egocentric level and dimensionality reduction techniques to produce a visual map of the egos in a network [5]. This map identifies clusters of egos with structurally-similar egocentric networks. This technique was applied in several application areas, for example, in assessing article quality in Wikipedia [19] and in identifying spam campaigns in YouTube comments [14]. However, the size of the networks was a limiting factor – the major computational bottleneck in each case was the network motif counting process. The work described in this paper addresses that problem and describes a scaleable solution for *dynamic* networks.

In this section, we review network motif analysis (Sect. II-A), the actor model (Sect. II-B), and distributed stream-processing systems (Sect. II-C). The actor model is a precursor to the model used in many distributed stream-processing systems. It is a useful starting point from which to introduce system-specific terminology. We summarize some of the concepts and terminology relating Storm at the end of Sect. II-C.

A. Network Motif Analysis

Network motif analysis considers a network as being comprised of simple substructures, or network motifs. In this work we consider all possible substructures involving triplets of nodes. There are sixteen distinct combinations of connections between three nodes, assuming connections can occur in either or both directions between nodes. *Network motif profiling* [11], [18] involves counting all occurrences of each subnetwork in a network. Milo et al. [10] use *network ratio profiles* to compare the counts of network motifs in networks of varying sizes. Using a correlation coefficient matrix, they identify several families of networks that have similar network ratio profiles, such as biological information-processing networks, WWW networks, social networks and autonomous systems networks. They show that networks from similar domains have similar network ratio profiles.

Koschützki et al. [9] formulate a number of network motif-based centrality measures. They rank the nodes of the E. Coli transcriptional network using each centrality measure. They claim that network motif-based centrality measures identify genes that are important regulators that are overlooked by local measures (*e.g.* based on out-degree) and global measures (*e.g.* using betweenness or centrality measures). Network motif analysis is generally concerned with entire networks and global counts. However, in the work presented here we apply an egocentric variation in which we consider local counts only – see Sect. III for more details. The results of Koschützki et al. [9] lend support to this approach by using local counts of network motifs to assess the importance of nodes.

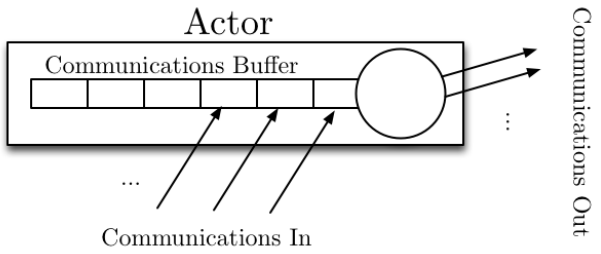

Fig. 1. In the actor model, each actor buffers received communications until it can process them.

Graphlet analysis [15], [16] is similar to network motif analysis except that the network motifs, or 'graphlets' in this case, are induced, the graphlets are locally counted with respect to each node, and the orbits or roles of the nodes with respect to the graphlets are also considered. Although their definition differs from that of networks motifs, they can be used in a similar way to characterize networks.

B. The Actor Model

In order to solve any problem using distributed computation, for example, dynamic egocentric network motif analysis, we need to answer two main questions. Firstly, how do we partition the problem into independently executable tasks? Secondly, how do we handle data dependencies between the tasks? The actor model [1], [6], [7] is a useful abstraction that helps answer these questions. It is a computational model in which computing elements known as actors are the universal primitives of concurrent computation. Each actor can send communications to other actors and decide how to behave in response to the communications it receives. All communications are asynchronous and unordered. Each actor buffers received communications until it can process them (see Fig. 1). The control structure emerges from the asynchronous communication pattern.

The actor model has been implemented in numerous languages and environments, *e.g.* ACT++ [8] and Scala [4]. There has been a resurgence in interest in this model with the advent of systems such as Yahoo!'s S4 [13], Nokia's Distributed Elastic Message Processing System (Dempsy[3]) and Nathan Marz' Storm, whose models closely resemble that of the actor model.

C. Distributed Stream-Processing Systems

Distributed stream-processing systems are typified by systems that process input streams in the form of an unbounded sequence of tuples, transform the streams in a distributed manner using a system of computing elements and produce output streams. S4 and Storm are examples. These systems are particularly geared towards clusters of commodity hardware. However, we note that there are some differences between the systems in terms of terminology and implementation. For example, Storm guarantees that all tuples will be processed

[3]Dempsy's website is here: http://dempsy.github.com/Dempsy

in the face of failures at the expense of additional capacity whereas S4 favors graceful degradation by accepting lossy failover. Since we rely on exact analysis we chose Storm for this paper.

In one sense, Storm is similar to the well-known Apache Hadoop[4] system in that it provides programmers with a general framework for performing streaming computations, much like Hadoop provides programmers with a general framework for performing batch operations. Storm introduces some novel concepts and terminology. *Topologies* define how a problem should be processed – how data is entered and processed through the system by means of data *streams*. *Spouts* are entities that handle the insertion of data tuples into the topology and *bolts* are entities that perform computation. The spouts and bolts are connected by streams to form a directed graph where spouts send streams of tuples to bolts, which perform some computation, and in turn pass further streams onto other bolts. Using a topology in this way it is possible to break a large and complex computation into simpler elements. Parts of the computation can be parallelised by setting a *parallelism number* for each spout and bolt in the topology specification. Bolts typically perform simple processing activities such as filtering, aggregations, and joins. They can also send tuples to external systems, *e.g.* distributed databases or notification services. The topology also specifies how streams are connected to bolts of a particular type. There may be many instances of each bolt as set by the parallelism numbers. For example, if a bolt has a parallelism number of five and receives a stream from a spout, then we must specify which of the five instances receives each tuple. For example, we may decide to distribute the tuples randomly between the instances, send the tuples to all of the instances, or choose an instance depending on the data contained within each tuple. A topology is similar to a job in Hadoop. However, Hadoop jobs eventually terminate whereas a topology may run forever (or until you kill it). This is important in the context of any dynamic network, where there is a need for continuous computation. Figure 2 illustrates the topology used for our experiments. It consists of two types of spout and three types of bolt. We will describe this topology in detail in Sect. IV.

Storm handles the coordination of computational resources to undertake the work specified by a topology. Storm also guarantees that all computations will be completed, even in the event of hardware failure.

III. Dynamic Egocentric Network Motif Analysis

The *egocentric network*, or k-*neighborhood subnetwork*, of an ego or node u is defined as the subnetwork induced by the set of nodes that have shortest-path distance at most k from u. In this paper we set $k = 2$ thereby limiting the size of the egocentric networks to one step out. We also assume that all networks are directed. We consider an ordered list of all connected networks with at most l nodes up to isomorphism. In this paper we set $l = 3$, and therefore the ordered list has

[4]Apache Hadoop's website is here: http://hadoop.apache.org/

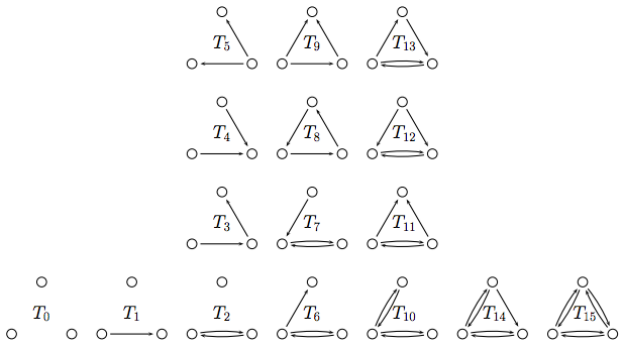

Fig. 3. The 16 possible directed graphs on three vertices, excluding isomorphisms (taken from [2]) are the basis for our network motif profiling.

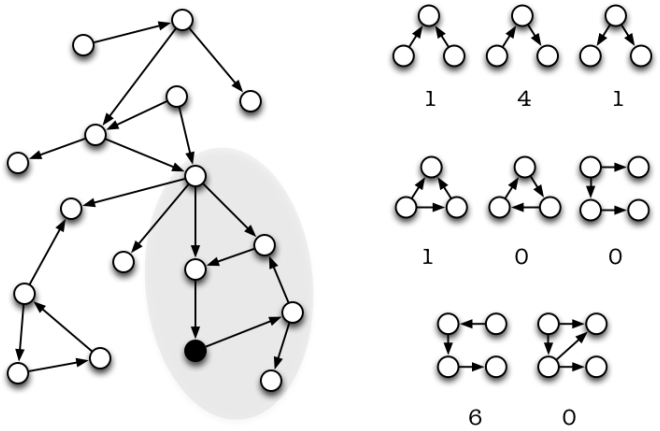

Fig. 4. The egocentric network of the filled node, for $k = 2$, includes all of the nodes and edges in the shaded area. The corresponding counts for a subset of the network motifs are shown on the right.

16 networks or network motifs. Both k and l can be increased to characterize the egos and their egocentric networks in more detail but at the expense of additional computational overhead. For each egocentric network we compute a *network motif profile*: a 16-element vector where each entry is the number of instances of the corresponding network motif in the ordered list in the egocentric network (see fig 3).

To see how this works in practice, consider the network in Fig. 4. The egocentric network of the filled node, for $k = 2$, includes all of the nodes and edges in the shaded area. A subset of the network motifs are shown on the right along with the number of times they occur in the egocentric network.

Furthermore, we are interested in a dynamic variant of this problem; the network is changing over time. We have implemented a dynamic algorithm by Eppstein *et al.* [2], [3] to maintain the egocentric network motif profiles as the network changes. This algorithm was originally developed for maintaining network motif profiles of entire networks. However, it is trivial to adapt this algorithm to the egocentric context. The algorithm maintains the network motif profiles for each egocentric network in $\mathcal{O}(h)$ time per update where h is the h-index of the corresponding egocentric network.

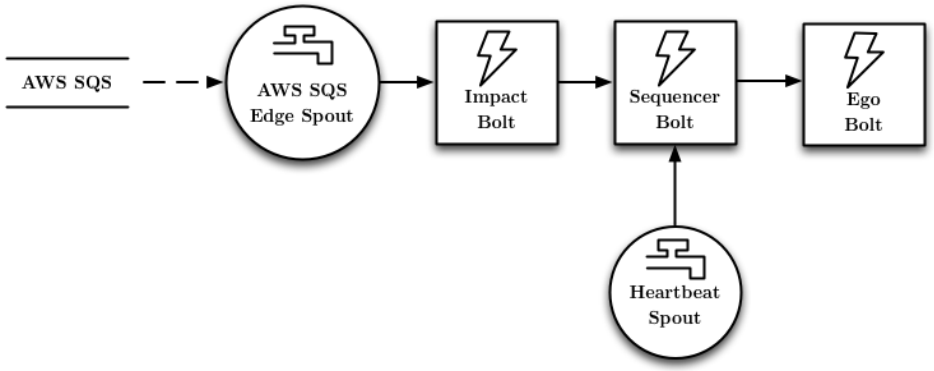

Fig. 2. A topology comprises spouts (circles) and bolts (squares); solid arrows are streams within the topology; dashed arrows connect to external systems.

In previous work [5], we found it useful to compute a *network ratio profile* [10] for each egocentric network: a 16-element vector where each entry is the *normalized ratio* of the corresponding entry in the network motif profile. The ratio profile rp of an egocentric network is computed using

$$rp_i = \frac{nmp_i - \overline{nmp_i}}{nmp_i + \overline{nmp_i} + \epsilon}$$

where nmp_i is the i^{th} entry of the network motif profile, $\overline{nmp_i}$ is the average of the i^{th} entry of all of the network motif profiles, and ϵ is a small integer that ensures that the ratio is not misleadingly large when the network motif appears very few times in all of the egocentric networks. To adjust for scaling the normalized ratio profile nrp of an egocentric network is computed using

$$nrp_i = \frac{rp_i}{\sqrt{\sum rp_j^2}}.$$

A normalized ratio measures the abundance of a network motif in each individual egocentric network relative to all the egocentric networks; it is similar to a z-score. It does not require the construction of an ensemble of random networks as found in related approaches [10]. The implementation described in the next section maintains network motif profiles only. However, it is possible to extend this implementation to produce network ratio profiles if required. We will discuss this in Sect. VII.

IV. Implementation Using Storm

Having implemented a dynamic algorithm to maintain egocentric network motif profiles over time, our next step is to use the Storm distributed stream-processing system to partition the workload. We need to specify a topology that defines the spouts and bolts and the streams that connect them. Figure 2 illustrates our topology. It has two types of spout and three types of bolt. In the following subsections we briefly describe the responsibility and implementation details of these components and how they interact to maintain the egocentric network motif profiles of a dynamic network over time. Any non-trivial topology must communicate with external systems for input and output, *e.g.* our topology's main spout takes network information (in the form of node connection information) from a external distributed queue, provided by the AWS platform. We have also implemented spouts and bolts that interface with other third-party systems, *e.g.* we use bolts tailored for Apache Cassandra[5] to persist the results of our analyses. However, we omit their details from this paper.

A. The Edge Spout

All information about changes in network structure is provided to the topology via a single spout type - the *edge spout*. This retrieves edges from a distributed queue deployed using Amazon's Simple Queue Service (AWS SQS[6]). It performs a simple data conversion and emits *edge* tuples. Each edge tuple contains a unique identifier, a timestamp, and unique identifiers for the source and target endpoints, *i.e.* the identifiers of the nodes connected by the edge. These four pieces of information are included in all edge tuples. Further information can be augmented to the edge, *e.g.* if edges represented financial transactions we could append the transaction value.

B. The Impact Bolt

The edge tuples emitted by the edge spout are received by *impact bolts*. The task of the impact bolt is to assess which egos are affected by each edge that is received. Obviously, the edge will affect the egos that it connects, but depending on the path distance set for the motif analysis (k, described in Sect. III) it may affect many more. For example, if k was set to 1 then it will at least affect all neighbours of both the egos directly affected by this edge. The impact bolt must maintain enough state to deduce which egos are impacted by each edge, *i.e.* each ego's egocentric network. Once an impact bolt has determined the set of egos that are affected by each edge they prepare a new tuple for each ego, which contains the edge, and a reference to the unique identifier of that ego.

[5]Apache Cassandra is an open source distributed database management system (see http://cassandra.apache.org/)

[6]Amazon SQS's website is: http://aws.amazon.com/sqs/

C. The Sequencer Bolt and Heartbeat Spout

The distributed and parallel nature of the input queue, as well as that of the processing performed by the edge spouts and the impact bolts introduces some perturbations in the ordering of the tuples emitted by the impact bolts. In order for the topology to perform a correct analysis it is crucial to smooth out these perturbations and order the tuples correctly. It is the role of the *sequencer bolt* to correct these perturbations.

The sequencer bolt receives ego tuples emitted by the impact bolt and checks the time sequences of each and reorders them as necessary using a bounded priority queue. The stream connecting the impact bolt with the sequencer bolt uses a fields grouping to ensure that all ego tuples relating to the same ego are handled by the same task. This allows the sequencer bolt to be easily parallelized and distributes the state required to correct perturbations.

The sequencer bolt requires a periodic 'heartbeat' to prompt it to check the timestamps of the edges contained in the tuples in its priority queue and emit those at the queue's head, thus correcting the perturbations in the input stream. The topology's second spout, the *heartbeat spout*, provides this stimulus in the form of a regular heartbeat tuple that is received by the sequencer bolt.

D. The Ego Bolt

The *ego bolt* is the workhorse of the topology; it encapsulates the dynamic algorithm that maintains the egocentric network motif profiles over time. It receives tuples emitted by the sequencer bolt that contain an edge and the identifier of the ego that is impacted by it. The stream of tuples coming from the sequencer bolt uses a fields grouping to ensure that all tuples relating to the same ego are handled by the same ego bolt. The ego bolt maintains the state that is required for the maintenance of the egocentric network motif profiles.

Ideally, due to the amount of work performed by this type of bolt we should set the parallelism number for the ego bolt to a significantly higher level than the other bolt types. We are using Storm primarily to distribute the work of this bolt type. However, we are ultimately bound by the size of the cluster to which we submit our topology. In Sect. V we investigate the size of a cluster and the appropriate number of ego bolts required to process a given network.

V. Datasets and Experimental Setup

Up to this point we have shown that the dynamic egocentric network motif analysis problem can be tackled using a distributed stream-processing system. We implemented a dynamic algorithm [2], [3] to maintain egocentric network motif profiles over time and we described a topology that can distribute the workload over a cluster of machines. Now we report on the resources required within the AWS cloud computing platform in order to support this type of analysis on two *real-world* datasets from the financial domain: a Bitcoin transaction network and a peer-to-peer lending network from the Prosper.com community. We also assess the scaleability of the analysis by investigating the speed-up provided by increasing the level of resources.

The networks derived from the Bitcoin transactions and Prosper.com Marketplace are proxies for any large dynamic network. That said, the results in Sect. VI will vary depending on the structure of the network being analyzed.

The experiments presented here can be considered a dynamic egocentric network motif analysis benchmark, where we are testing our implementation of the algorithm and topology described earlier using Storm and AWS's infrastructure. For the experiments presented here we used Storm version 0.6.2[7]. We specified the topology using a domain-specific language (DSL) for Storm in the Ruby programming language[8].

A. The Bitcoin Dataset

The first dataset is based on financial transactions in Bitcoin [12]. Bitcoin is an electronic currency with no central authority or issuer. The Bitcoin network came into existence in January 2009 and as of August 2012, one Bitcoin has a market value of approximately $12. Bitcoins are generated at a predictable rate such that the eventual total quantity of Bitcoins will be 21 million.

There is no requirement for a trusted third-party when making Bitcoin transactions. The participants of the Bitcoin peer-to-peer network form a collective consensus regarding the validity of each transaction by appending it to a public history of previously agreed-upon transactions (the *block-chain*). From the block-chain we can derive a network: the *transaction network* represents the flow of Bitcoins between transactions over time [17].

Each node in the transaction network represents a financial transaction and each directed edge from a source to a target represents an output of the transaction corresponding to the source that is an input to the transaction corresponding to the target, *i.e.* a flow of Bitcoins from the source to the target. Each directed edge also has the timestamp of the transaction corresponding to the target. The Bitcoin dataset used in the experiments presented here was generated based on all data up to the 9$^{\text{th}}$ July 2012. This network has 4.8 million nodes (or egos) and 9.3 million edges.

In ongoing work we are applying dynamic egocentric network motif analysis to this network in order to characterize the different types of transactions (*e.g.* consolidations and dispersals) and assess their prevalence over time. However, for the purposes of this experiment the focus is on assessing the level of resources and typical running-times needed to perform the analysis rather than the outcomes of the analysis itself.

B. The Prosper.com Dataset

The second dataset is based on the Prosper.com peer-to-peer lending network[9]. As of August 2012, Prosper.com has more than 1.4 million members and over $375 million in funded

[7]Storm is available on GitHub: https://github.com/nathanmarz/storm

[8]We used Colin Surprenant's RedStorm version 0.4.0, which is available on GitHub: http://github.com/colinsurprenant/redstorm

[9]Prosper.com's website is http://www.prosper.com

TABLE I
WE DEPLOYED OUR TOPOLOGY TO THREE CLUSTERS WITH VARYING NUMBERS OF SUPERVISORS: *small*, *medium* AND *large*.

Cluster Name	# Supervisors	# Supervisor ECUs	Ego Bolt Parallelism Number
Small	15	300	82
Medium	20	400	122
Large	25	500	162

loans. Borrowers participate in the Prosper.com marketplace by asking for money in the form of listings. Lenders bid on listings specifying repayment terms including interest rates. If enough lenders fund a listing, the listing becomes a loan. Prosper.com rates prospective borrowers according to their creditworthiness. It also maintains borrower and lender groups, endorsements, past listings, bids and loans. The social structure of the service is evident from the data: a node represents a borrower or lender and a directed edge represents a fraction of a listing or bid offered by a lender to a borrower. We note that borrowers can also be lenders and therefore the network is not necessarily bipartite.

In previous work, we applied egocentric network motif analysis to this network in order to characterize the behaviour of the borrowers and lenders [5]. However, due to the computational expense, we were restricted in the size of network we could handle. We could only apply static egocentric network motif analysis to subsets of the network, *e.g.* only those edges that applied to listings that eventually made it to loan status within a given time-slice, usually a month. Furthermore, we were unable to handle the temporal dimension and treated each month's network as a static network. We required a distributed stream-processing system to overcome this limitations.

The dataset generated for this work was generated based on all data up to the 19st July 2012. This network comprises 0.5 million nodes (or egos) and 10.5 million edges. We exclude isolated nodes (members who have not participated in any bid or listing) from this network.

C. The AWS Cloud Computing Platform

Amazon's AWS cloud computing platform provides access to a pool of computing resources and services. For our experiments we used clusters of virtual machines (AWS EC2), distributed queues (AWS SQS) and monitoring services (AWS CloudWatch). We briefly document the specifications of these resources in order to facility reproducibility and to put the results in Sect. VI in context.

We used two types of EC2 instance: the `m1.large` and `c1.xlarge` instances. An `m1.large` instance has 4 *EC2 Compute Units (ECU*[10]*)*, 7.5 GB memory and 850 GB storage. A `c1.xlarge` instance has 20 ECU, 7 GB of memory and 1 690 GB of instance storage. We used 64-bit Ubuntu 10.10 (Maverick Meerkat) on all instances.

We deployed three clusters, *small*, *medium* and *large*, to the U.S. East (Northern Virginia) region. Both clusters had one nimbus (`m1.large`) and varying numbers of supervisors (`c1.xlarge`) – see Table I. They also included a single-node Apache ZooKeeper cluster (`m1.large`). We populated six AWS SQS queues, three with the 9.3 million edge network derived from the Bitcoin dataset and three with the 10.5 million edge network derived from the Prosper.com dataset. We submitted the same topology, described in Sect. IV-A, to the three clusters.

VI. EXPERIMENTAL RESULTS

Our topology and cluster configurations have a large number of parameters and implementation details that impact performance. However, using monitoring tools and services such as Ganglia[11] and AWS CloudWatch we were able to tune these in order to extract the best performance. Storm provides a monitoring service that reports on the progress of each spout and bolt in a running topology. We polled this service every 5 minutes to produce the plots in figures 5 to 8. All times are rounded up to the nearest 5-minute mark.

For the Bitcoin network, our topology on the *large* cluster completed in 290 minutes. Fig. 5 illustrates the results. The 9.7 million edge tuples were emitted by the AWS SQS edge spout and acknowledged (*ack'd*[12]) by the impact bolt in 260 minutes. These edge tuples resulted in the impact bolt emitting 46 million ego tuples. These ego tuples were ack'd by the ego bolt, at which stage the appropriate egocentric network motif profiles had been updated, in 290 minutes. During the execution of the topology, the load on the cluster was in the 80–90% range. This indicates that our parallelism numbers were set appropriately.

We also ran the same topology on the *medium* and *small* clusters. The reduction in processing power is reflected in Fig. 6. The *medium* and *small* clusters required an extra 35 and 75 minutes respectively to process the same network. This ability of Storm to distribute a workload across the available supervisors is what makes it particularly suited to the dynamic egocentric network motif analysis problem. Although updating the egocentric network motif profiles is a computationally expensive process, it is easily parallelized and Storm takes care of distributing this workload across the cluster of machines.

We performed similar experiments with the Prosper.com dataset. As noted in Sect. V-A and V-B, both the Bitcoin and Prosper.com networks have a similar number of edges but very different numbers of nodes. The Prosper.com network is

[10]An ECU is an abstraction of CPU resources. Amazon states that an ECU has the equivalent CPU capacity of a 1.0–1.2 GHz Intel Xeon processor (2007) or AMD Opteron processor (2007).

[11]Ganglia is a scalable distributed monitoring system for high-performance computing systems (see http://ganglia.info)

[12]Storm uses acknowledgements (*acks*) to indicate that a tuple has been processed successfully.

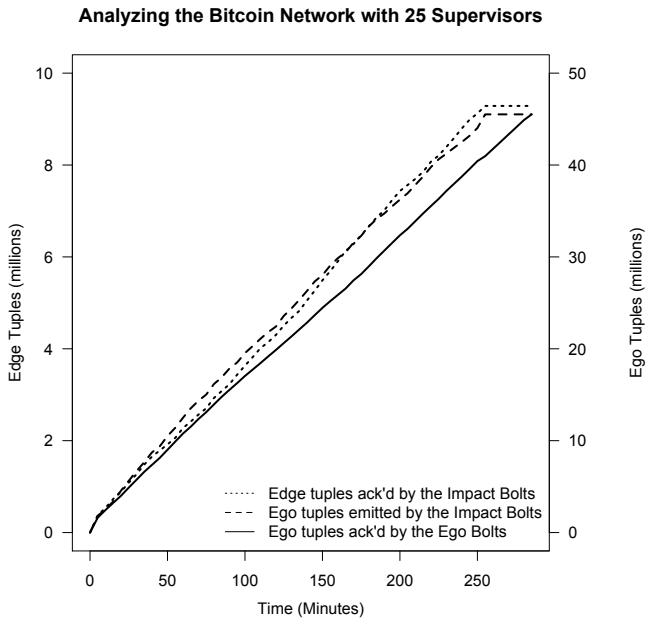

Fig. 5. The performance of our topology on the Bitcoin-derived network with 162 ego bolt tasks on the *large* cluster with 25 supervisors.

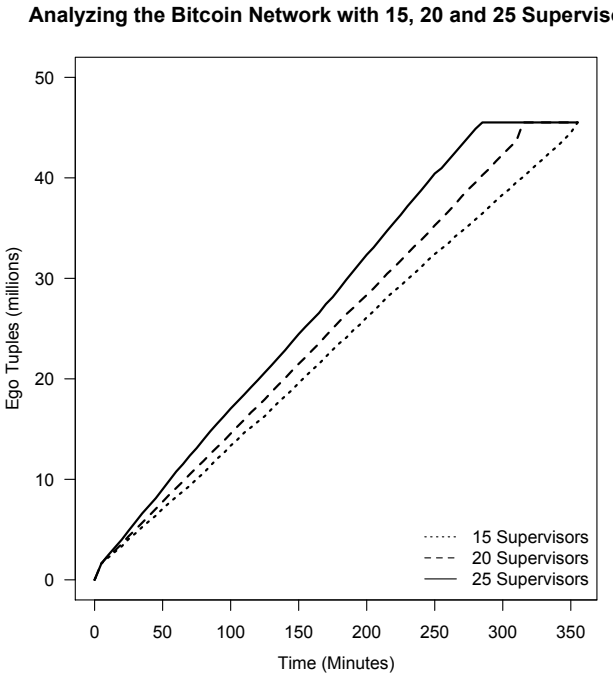

Fig. 6. A comparison of the performance of our topology on the Bitcoin-derived network on the *small*, *medium* and *large* clusters.

much denser. This means that the addition of a single edge can impact many more egos than was the case with the Bitcoin network. This is borne out by the results as the 10.5 million edge tuples create 243 million ego tuples (see Fig. 7). Our topology on the *large* cluster completed in 715 minutes. The 10.5 million edge tuples were emitted by the AWS SQS edge spout and acknowledged by the impact bolt in 345 minutes.

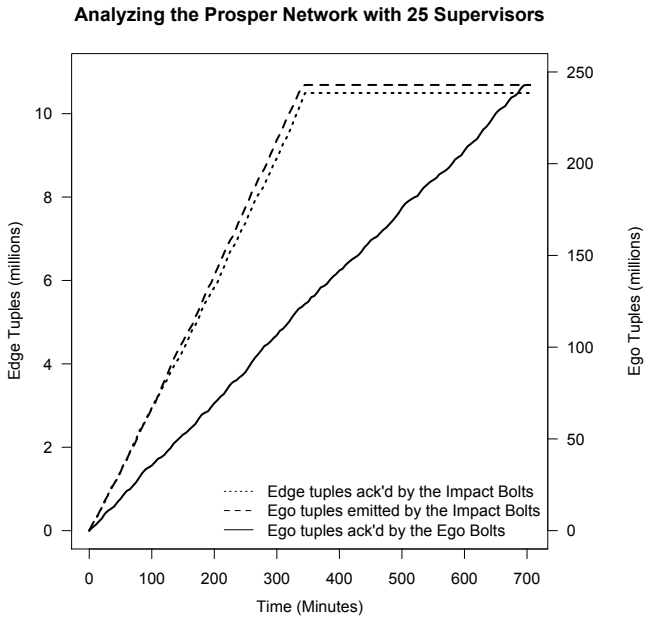

Fig. 7. The performance of our topology on the Prosper.com-derived network with 162 ego bolt tasks on the *large* cluster with 25 supervisors.

The resulting ego tuples were acknowledged by the ego bolt in 715 minutes. We also ran the same topology on the *medium* and *small* clusters (see Fig. 8). They required an extra 70 and 160 minutes respectively to process the same network.

Our results demonstrate the horizontal scaleability of Storm. By increasing the number of supervisors we were able to increase the slope of the lines indicating the number of ego tuples emitted by the Impact Bolts in both Fig. 6 and Fig. 8. Storm was able to distribute the workload using the additional supervisors and improve the overall running-times.

VII. Conclusions and Lessons Learned

We formulated a solution to the dynamic egocentric network motif analysis problem using the distributed stream-processing system Storm. This enables significant speedup when used in combination with a cluster of machines, for example, one provided by a cloud computing platform such as AWS. We reported on the resources required to perform such an analysis on two real-world networks from the financial domain with approximately 10 million edges in each. We demonstrated the horizontal scaleability of the system. This approach vastly increases the size of networks that are amenable to dynamic egocentric network motif analysis.

The main focus for future work revolves around extensions to our topology. For instance, in Sect. III we introduced both network motif profiles and network ratio profiles. The present topology maintains network motif profiles only. However, it is possible to augment the existing topology so that it also maintains network ratio profiles. This will allow for more meaningful comparisons between egos. We can also extend the topology to perform online clustering, thereby automatically identifying groups of egos with structurally-similar egocentric networks.

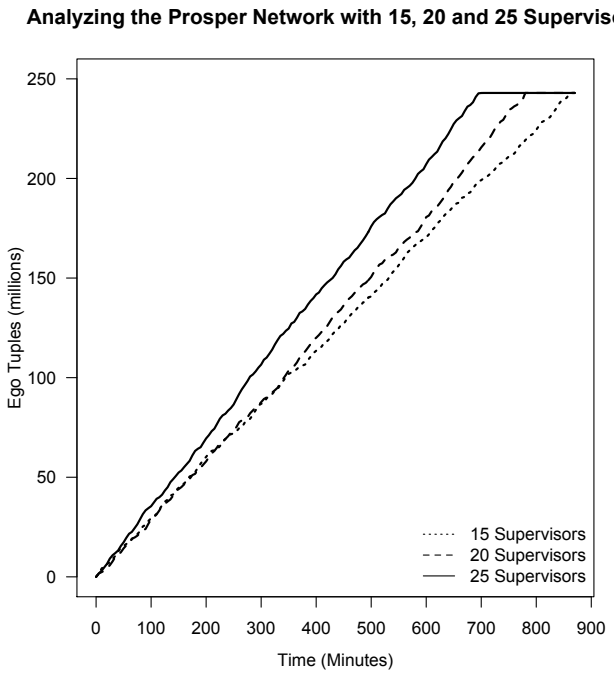

Fig. 8. A comparison of the performance of our topology on the Prosper.com-derived network on the *small*, *medium* and *large* clusters.

A. Lessons Learned

There were many engineering challenges associated with developing and deploying our system in the cloud. Some of these related to our choice of distributed stream-processing system, Storm, whereas others related to AWS. During our experiments, we used Storm version 0.6.2; many of the problems below are being addressed in recent and future versions.

Storm is not elastic; we cannot re-provision existing clusters and topologies at runtime. We cannot add or remove supervisors or dynamically change the parallelism numbers for spouts and bolts. This forces the developer to configure a cluster and topology through trial-and-error, which is very time-consuming, and potentially expensive.

It is difficult to develop and test the individual bolts and spouts that make up a topology in isolation. Ideally, we would like to mock a simple topology and test each component independently against functional and non-functional requirements. By looking at performance requirements this might help to identify bottlenecks within individual components before assembling them into real topologies.

Storm's monitoring service is currently rudimentary. It does not provide fine-grained, time-series metrics nor can it persist these metrics to an external datastore, for example, AWS CloudWatch. We had to poll this service every 5 minutes to produce the plots in this paper, which was not ideal.

Storm provides a tool for provisioning Storm clusters using AWS. However, the time required to provision a cluster, install and configure the necessary software and then submit a topology can slow down development (there is a certain amount of waiting around involved). It may be possible to significantly reduce this time using, for example, custom Amazon Machine Images (AMIs).

It should be stressed however that the Storm community is vibrant and growing. We believe that majority of these problems will be addressed through improved developer tooling.

ACKNOWLEDGMENT

This research was supported by Science Foundation Ireland (SFI) Grant No. 08/SRC/I1407 and an AWS in Education research grant award.

REFERENCES

[1] G. Agha. *ACTORS – A Model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.

[2] D. Eppstein, M. Goodrich, D. Strash, and L. Trott. Extended h-Index Parameterized Data Structures for Computing Dynamic Subgraph Statistics. In W. Wu and O. Daescu, editors, *Proceedings of the 4th International Conference on Combinatorial Optimization and Applications (COCOA'10)*, pages 128–141. Springer, 2010.

[3] D. Eppstein and E. Spiro. The h-Index of a Graph and its Application to Dynamic Subgraph Statistics. In F. Dehne, M. Gavrilova, J. Sack, and C. Tóth, editors, *Procs. of the 11th Intl. Symposium on Algorithms and Data Structures (WADS'09)*, pages 278–289. Springer, 2009.

[4] P. Haller and M. Odersky. Scala Actors: Unifying Thread-Based and Event-Based Programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.

[5] M. Harrigan, D. Archambault, P. Cunningham, and N. Hurley. EgoNav: Exploring Networks through Egocentric Spatializations. In *Proceedings of the International Conference on Advanced Visual Interfaces (AVI'12)*, pages 563–570. ACM, 2012.

[6] C. Hewitt. Actor Model of Computation: Scalable Robust Information Systems. 2011.

[7] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular Actor Formalism for Artificial Intelligence. In N. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI'73)*, pages 235–245, 1973.

[8] D. Kafura. ACT++: Building a Concurrent C++ with Actors. *Journal of Object-Oriented Programming (JOOP)*, 3(1):25–37, 1990.

[9] D. Koschützki, H. Schwöbbermeyer, and F. Schreiber. Ranking of Network Elements Based on Functional Substructures. *Journal of Theoretical Biology*, 248(3):471–479, 2007.

[10] R. Milo, S. Itzkovitz, N. Kashtan, R. Levitt, S. Shen-Orr, I. Ayzenshtat, M. Sheffer, and U. Alon. Superfamilies of Evolved and Designed Networks. *Science*, 303(5663):1538–1542, 2004.

[11] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network Motifs: Simple Building Blocks of Complex Networks. *Science*, 298(5594):824–827, 2002.

[12] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. Available from http://bitcoin.org/bitcoin.pdf.

[13] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *Procs. of the Intl. Workshop on Knowledge Discovery Using Cloud and Distributed Computing Platforms (KDCloud'10) at ICDM'10*, pages 170–177. IEEE, 2010.

[14] D. O'Callaghan, M. Harrigan, J. Carthy, and P. Cunningham. Network Analysis of Recurring YouTube Spam Campaigns. In J. Breslin, N. Ellison, J. Shanahan, and Z. Tufekci, editors, *Procs. of the 6th Intl. AAAI Conference on Weblogs and Social Media (ICWSM'12)*, 2012.

[15] N. Pržulj. Biological Network Comparison using Graphlet Degree Distribution. *Bioinformatics*, 26(6):853–854, 2010.

[16] N. Pržulj, D. Corneil, and I. Jurisica. Modeling Interactome: Scale-Free or Geometric? *Bioinformatics*, 20(18):3508–3515, 2004.

[17] F. Reid and M. Harrigan. An Analysis of Anonymity in the Bitcoin System. In *Procs. of the 1st Workshop on Security and Privacy in Social Networks (SPSN'11) at SocialCom'11*, pages 1318–1326. IEEE, 2011.

[18] S. Shen-Orr, R. Milo, S. Mangan, and U. Alon. Network Motifs in the Transcriptional Regulation Network of Escherichia Coli. *Nature Genetics*, 31:64–68, 2002.

[19] G. Wu, M. Harrigan, and P. Cunningham. Characterizing Wikipedia Pages Using Edit Network Motif Profiles. In *Procs. of the 3rd Intl. Workshop on Search and Mining User-Generated Contents (SMUC'11) at CIKM'11*, pages 45–52. ACM, 2011.