DiCeS: Detecting Communities in Network Streams Over the Cloud

Panagiotis Liakos
University of Athens
GR15703, Athens, Greece
p.liakos@di.uoa.gr

Katia Papakonstantinopoulou
Athens Univ. of Economics & Business
GR10434, Athens, Greece
katia@aueb.gr

Alexandros Ntoulas
LinkedIn
Mountain View, CA 94043
ntoulas@gmail.com

Alex Delis
University of Athens
GR15703, Athens, Greece
ad@di.uoa.gr

***Abstract*—We consider the problem of uncovering communities in complex real-world networks whose nodes and their respective associations originate in streams of data. Although community detection has received much attention in centralized settings, the prevalence of online social networks has resulted in unprecedented volumes of data whose handling calls for novel *streaming* approaches. Moreover, bursty production of network interactions necessitates *cloud-enabled techniques* that can both deal with diverse data rates and deploy more computing resources on the fly for improved performance yields. We propose a *distributed* streaming community detection approach termed DiCeS, and implement it as a *cloud application*. While seeking communities, the novelty of our approach is at balancing the incoming load to a cluster of computing nodes and adjusting the cluster processing capacity in an elastic manner. We also provide fault tolerance by ensuring that temporarily suspended or failed nodes are restored and all edges of the network stream ultimately received their due processing. Lastly, DiCeS is *interactive* regarding i) updating the target communities, and ii) obtaining results on demand. Our experimental results demonstrate that DiCeS does handle the edges of real-world network streams at impressive rates, allows for *near-linear* scaling, and outperforms previous non-distributed approaches. While using ground-truth communities for a wide range of large real-word networks, we also show that DiCeS attains improved accuracy if compared to earlier centralized algorithms.**

***Keywords*-Graph streams, community detection, cloud.**

I. Introduction

Network communities are groups of nodes that exhibit high cohesion among themselves and remain loosely connected to all other nodes in the network. In most cases, nodes of a community exhibit similarities. For instance, social network communities often group individuals that share common interests and WWW communities comprise websites that are similar as far as their content is concerned. Uncovering the community structure of real-world networks is a challenging problem that has received considerable attention [1], [2], [3], [4], [5]. Pertinent applications appear in emerging computational environments including social computing, web analysis, IoT and computational biology [6], [7], [8], [9], [10]. Understanding network communities does lead to invaluable insights on the functioning of many systems that are an integral part of everyday life. Networks portraying such systems often reach massive volumes and may evolve rapidly [11]. The sheer size of such networks makes most community detection approaches *prohibitive* and has motivated a) efforts focusing on expanding sets of 2 or 3 *seed nodes* to communities [12], [13], [14], [15], and b) streaming approaches that detect communities *as they are formed* [16], [17], [18].

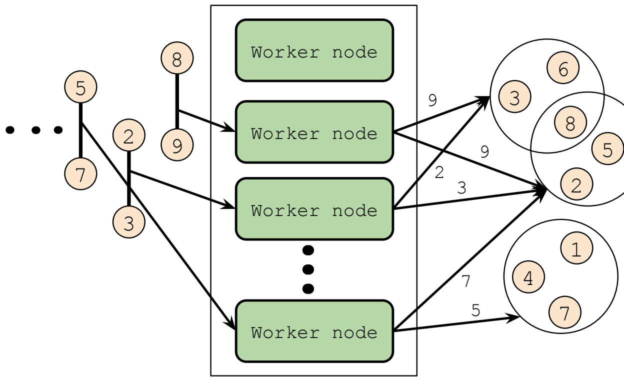

Figure 1: Context of DiCeS

Detecting communities in rapidly-evolving networks is of paramount importance in many contexts. In network streams portraying public on-line discussions, we can focus on individuals sharing a common interest, for example in sports, fashion or politics, and come with an extended group of individuals who *currently* share this specific concern. The latter allows for launching very accurate and successful targeted advertising campaigns. We can also significantly improve social network feeds by discovering in *real-time* different groups a user belongs to. Moreover, in the case of the World Wide Web, we can detect communities of websites focusing on topics defined by a few web pages and study these communities to gain insights on the evolution of the Web.

Existing streaming algorithms [16], [17], [18] are by design centralized and thus, bound by the processing power and memory limitations of a single machine. To address these limitations and manage varying or even extreme data arrival rates, we develop an elastic, streaming, interactive, and fault tolerant *cloud application* for community detection we call DiCeS. Figure 1 depicts the overall functionality of our approach. A network is made available as a stream of edges (i.e., $(8, 9), (2, 3), (5, 7)$), that arrive at *no particular* *order* and must be processed *as they arrive*. Furthermore, we consider a set of communities, each one defined by a few of seed nodes, that get expanded as we process the stream. Each arriving edge is processed by one of the available computing workers, and its adjacent nodes (e.g., 2 and 3 for the $(2, 3)$ edge) are appropriately added to all communities they belong to; this is initially accomplished using the adopted seed nodes.

The principle design choice of DICES is *elasticity* as we intend on handling volume surges in the stream and we occasionally have to augment the number of sought communities. In this context, every network edge is emitted to a single computing worker. Clearly, the number of such workers can be adjusted *at will* as we enhance or trim our processing capability. Our design also provides for the *seamless recovery* from worker crashes or faults, and allows users to i) *dynamically* update their sought communities, and ii) retrieve results *on demand*.

Our experimental results show that we can process on the average every edge of a streaming network in as little as $74 \mu s$ while using just 8 workers. In this respect, we can handle almost 50 million edges per hour which is more than twice the amount of tweets posted per hour in `Twitter`.[1] More importantly, we can vastly increase the processing capacity by adding nodes, as we achieve horizontal scalability that is close to *linear*. Last but not least, using numerous real-world networks that are accompanied with ground-truth communities, we show that DICES offers significant improvements with regard to detecting accuracy. In summary, we make the following contributions:

- We propose DICES, a novel distributed community detection algorithm for network streams. To the best of our knowledge this is the first streaming community detection algorithm that distributes execution in an elastic manner.
- We implement DICES as a cloud application and show that we handle streams of real-world networks at impressive rates. Adding processing nodes results in near-linear scaling and allows for greatly outperforming earlier approaches that do not scale out.
- We ascertain the accuracy of our algorithm and demonstrate significant improvements over earlier efforts.

Our paper is organized as follows: we first introduce the frameworks our approach builds on in Section II. Then, we formulate our problem and discuss our approach for graph stream community detection in Section III. In Section IV, we extensively evaluate our approach and its variations with regard to accuracy, execution time, and space requirements. Section V reviews related work and finally, Section VI offers our concluding remarks.

II. PRELIMINARIES

We build our approach atop the `Apache Storm`, a popular streaming processing framework and `Redis`, a main-memory key-value store. In this section, we briefly outline both these frameworks.

A. Apache Storm

A number of distributed stream processing platforms have become available such as `Apache Storm`, `Samza`, and `Apache Flink`. In the context of this work, we employ `Storm` due to its broad use in production environments.[2] `Storm` offers a set of building blocks that help realize distributed platforms to process large volumes of streaming data in a highly scalable manner [19]. `Storm`'s fundamental *data unit* is called *tuple*. Tuples are dynamically typed and comprise a list of fields. An unbound sequence of tuples forms a *stream* which is the main abstraction of the framework. A `Storm`-based application creates a *topology* that processes such a stream, using *spouts* and *bolts*:

• *Spouts*: serve as the source of tuples in a topology. Spouts listen to data from external sources and *emit* them into streams. When the processing of a tuple successfully completes, the spout receives an *Ack*. In light of an error, the spout re-emits the tuple again and ensures that all tuples have been processed at least once.

• *Bolts*: are responsible for transforming the stream into the desired result. Bolts are often assigned with a simple task and the coordination of many such bolts allows for the building of complex transformations. Bolts may also generate new streams by emitting a new tuple after processing the one just received. Bolts can also persist information by dispatching it to a database. To acquire tuples, bolts subscribe to streams produced by spouts or other bolts. When subscribing to a stream, bolts may define the *grouping* that determines how the tuples are exchanged; the following are some of the groupings available in `Storm`:

◦ *Shuffle* grouping determines that each tuple is randomly sent to only one of the bolts that have subscribed to the stream.
◦ *All* grouping broadcasts a copy of each tuple to all bolts that listen to the stream.
◦ *Fields* grouping uses a particular field of the tuple to guarantee that a given set of values is always directed to the same bolt.
◦ *Custom* grouping may also be defined and employed so that we can freely determine the bolts that will receive each tuple by combining the above techniques.

Storm users may combine spouts and bolts to create complex flows (topologies) in which every node transforms information and forwards it to another node. Topologies may locally run on a machine, mostly for developing and

[1]http://www.internetlivestats.com/twitter-statistics/

[2]http://storm.apache.org/Powered-By.html

debugging purposes, or they can be submitted into a running Storm cluster, for flexible use of cloud computing resources [20].

B. Redis

Redis is an effective in-memory key-value data store used in production settings. It provides ultra-fast read/write operations as it keeps by default all its data in memory [21]. Subsequently, Redis is suitable for applications calling for near-real-time access while processing fast-moving data streams. Redis has become popular due to the variety of complex data types it offers, including *Lists, Hashes, Bitmaps*, and *HyperLogLogs* [22]. In this work, we are predominantly interested in the following data types:

- *Strings*: are the most versatile data type as it offers numerous Redis commands and serves multiple purposes. For instance, we can use a String to realize a *counter* variable in our application issuing a Redis INCR command.
- *Sets*: are unordered collections of distinct Redis Strings implemented through a hash-table. Sets offer constant time operations for *insertion, removal*, and *lookup* and are valuable in scenarios that require maintenance of *membership* information.
- *Sorted Sets*: are collections of distinct Strings sorted according to the score each one is associated with. They are implemented using a skip list and a ziplist, and they are more expensive than Sets, as adding, removing, or updating an item run in logarithmic time. However, they come handy when we need to differentiate between items when it comes to their ranking.

Redis uses only one CPU-core as it is single-threaded. However, multiple Redis-servers may form a *cluster* that automatically shards data across different Redis instances. A cluster requires at least 3 master servers. Each such server owns a portion of a total of 2^{14} *hash slots*. Every key corresponds to a hash slot through a MOD operation on the integer that results after applying the CRC-16 hash function to the key [23]. By distributing keys to multiple Redis instances as described above, a cluster can scale horizontally.

III. Detecting Communities in Network Streams Over the Cloud

A. Problem formulation

The salient element of a network stream is an edge, defined as an unordered pair of nodes, i.e., edge $e = (u, v)$. A streaming sequence of unordered edges $S = \langle e_1, e_2, \ldots \rangle$ naturally defines an undirected, unweighted graph $G = (V, E)$, where $V = \{v_1, v_2, \ldots\}$ is a set of nodes and $E = \{e_1, e_2, \ldots \} \subseteq V \times V$ is a set of undirected edges. Our goal is to discover communities whose initial members are user-defined as sets of 2 or 3 seed-nodes. At the same time, we may seek to uncover multiple communities: every *seed-set* $K_i \subset V$ helps initially designate the ith community C_i we seek to uncover. Provided a stream S and a set of seed-sets $K = \{K_1, K_2, \ldots, K_s\}$, we are to generate the respective communities $C = \{C_1, C_2, \ldots, C_s\}$. To attain this, we *expand each seed-set* by adding nodes adjacent to the members of the set, while aiming to produce groups of nodes tightly connected to each other. For each such group, we simultaneously attempt to maintain limited ties with the rest of the network's nodes. To this end, we build on the remarkably effective community detection techniques of our recent *centralized* CoEuS algorithm [18].

Due to the lack of a universal definition of what a network community is, we base our evaluation on publicly available networks with ground-truth communities that enable us to compare our approach with earlier efforts.

B. DiCeS' *Design Principles*

DiCeS is designed to address several challenges arising in the context of streaming community detection over a cloud infrastructure. First and foremost, our approach needs to be *scalable* with regards to the rate with which edges arrive through the network stream as well as the total number of communities we seek. Moreover, we are targeting long running cloud applications that need to provide *fault tolerance*. Finally, we wish DiCeS to be *interactive*.

1) Scalability: To address the first challenge, we focus on isolating the processing that has to be carried out for every edge at hand. We employ a distributed key-value store such as Redis–cluster to hold our data (Figure 2), and provide a scalable solution by distributing the execution that takes place for each edge of the network stream across multiple nodes. As Figure 2 depicts, a spout receiving the network stream from an external data-source communicates with several processing bolts. Each edge is sent to a *single* bolt that is selected at random. This allows for near-uniform load distribution across the processing nodes as every worker node (bolt) is expected to receive and process approximately an equal number of edges. When the edge arrival rate increases considerably, we can scale DiCeS horizontally by adding more processing bolts. If we are to also expand the number of communities under detection, the processing time of each edge is respectively expected to increase. Again, by offering more bolts, DiCeS allows for horizontal scaling.

2) Fault tolerance: Cloud applications occasionally have to deal with events of failure. In this context, we need to make sure that: i) all edges get ultimately processed, and ii) failing nodes are restored. To address the first issue, we assign a message *ID* to every edge that is emitted by the spout. This *ID* allows for specific edge tracking and we use it to ascertain that all edges are ultimately processed, *regardless* of failures that might occur in the cloud workers. As far as the second issue above, we design DiCeS so that all data reside in a *high–availability* distributed key-value store and are accessible by *all functional elements* of our framework (Figure 2). We also configure our processing

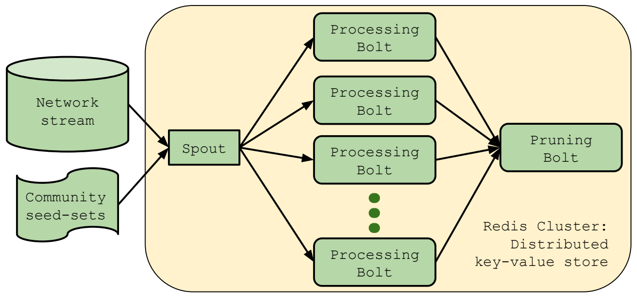

Figure 2: DICES's architecture: all spouts and bolts have access to a `Redis` cluster.

bolts so that, upon initialization, they only need to acquire the communities *currently* sought. Here, neither communication with sibling bolts, nor any synchronization of the bolts with a "master" node is required. If a processing bolt "dies", DICES attempts to restart the failed bolt on the worker node it was running on. If the worker has died, then DICES commences the bolt on one of the already available workers. Hence, there is no data loss, and the overall processing capacity is immediately restored.

3) Interactivity: The insights we gain while uncovering communities in a continually evolving network, often bring about the requirement to update the set of communities that we seek *on the fly*. DICES stores all pertinent information related to the communities in the `Redis`-cluster of Figure 2. Subsequently, the members of all communities as well as a representative score depicting their "involvement" in their own community are available *on demand*. Moreover, the pruning bolt of Figure 2 produces logs of the state of communities periodically. In this manner, DICES records information that allows for tracking the *evolution* of each community while the stream undergoes processing.

If we want to update the set of communities sought through either introducing new seeds all together or modify existing ones, we use the spout of Figure 2. The spout allows for ingesting input data from various external sources, such as an `HDFS` directory or a `Kafka` broker. Thus, we can feed DICES at anytime with a renewed set of communities that in turn gets broadcasted to all processing bolts.

C. DICES' *Cloud Components*

In this section we outline the operations performed by all DICES cloud components and how they co-operate to address the task of distributed community detection over streams. We report where appropriate all techniques adopted from the centralized COEUS algorithm, along with all the improvements and adjustments made in the context of our work.

1) Spout: This component is responsible for ingesting the application input, i.e., the network stream and the community seed-sets. The spout also initializes the communities in our distributed key-value store.

• *Community Initialization:* The spout keeps the community *ID*s and the seeds of each community in the `Redis`-cluster using `Sets`. This allows processing bolts to access the communities during their initialization. Additionally, for each community received, the spout creates a `Sorted Set`. The latter is initially populated with the *seeds* of the community associated with the maximum score of 1.0; this signifies that seeds remain in this community forever. Moreover, for every seed-node the spout creates a `Set` including the communities this node *is part of.* This `Set` serves as an inverted index for rapid access to all communities a node belongs to. This is a critical feature as it helps notably reduce the overall computations required. Finally, upon receipt a renewed set of communities, the spout broadcasts this set to all processing bolts. We use `Storm`'s `all grouping` to ensure that the bolts are up-to-date.

• *Stream Ingestion:* The spout is also responsible for dispersing the network stream to the processing bolts. We configure the spout to *randomly* dispatch every edge of the stream to a *single* bolt. We associate every edge with an *ID* and await for the corresponding acknowledgment which essentially ascertains the successful DICES processing of the edge. If such acknowledgment is not received before a time-out period elapses, the edge is emitted again.

2) Processing bolt: In our cloud topology, processing bolts do most of the work. Here we outline the actions taken when processing bolts enter their initialization and running phases.

• *Initialization phase:* during this stage, a processing bolt retrieves from the distributed key-value store all the communities that are currently under detection. Every community is labeled with a natural number that serves as its identifier. Thus, initializing the sought communities is simply a task of fetching the current set of community identifiers. Having acquired this set, a processing bolt can handle any incoming edge(s). This initialization phase is particularly useful when new processing bolts join an existing topology. In the case of a newly deployed topology, processing bolts do receive the seeding communities directly from the spout.

• *Running phase:* at this time, a processing bolt can receive edges as they become available from the spout. Upon receiving an edge, a processing bolt evaluates whether its adjacent nodes should be included in any communities and updates the data of the application accordingly. Additionally, a processing bolt may also handle messages that feature a renewed set of sought communities. Algorithm 1 depicts the actions taken when a processing bolt receives a tuple. The bolt first checks whether the input message features exactly one field (Line 2). If so, the tuple signifies a renewed community set and the bolt uses this set to update its local structure holding the communities (Line 4).

If the number of fields is greater than one (Line 5), the input message comprises the adjacent nodes of an edge (Lines 7-8). We commence processing an edge by

```
Algorithm 1: Processing bolt: execute($tuple$)
input : A tuple emitted from the spout.
1  begin
2      if $tuple$.length == 1 then
3          // renewed set of communities
4          $communities \leftarrow tuple[0]$;
5      else
6          // handling of an edge
7          $u \leftarrow tuple[0]$;
8          $v \leftarrow tuple[1]$;
9          $degrees[u] \leftarrow degrees[u]$+1;
10         $degrees[v] \leftarrow degrees[v]$+1;
11         foreach $C \in \{nc[u] \cup nc[v]\}$ do
12             if $u \in C$ then
13                 $cDegrees[C][v]$+=$\frac{cDegrees[C][u]}{degrees[u]}$;
14                 $communities[C]$.put($v$, $\frac{cDegrees[C][v]}{degrees[v]}$);
15                 $nc[v]$.add($C$);
16             if $v \in C$ then
17                 $cDegrees[C][u]$+=$\frac{cDegrees[C][v]}{degrees[v]}$;
18                 $communities[C]$.put($u$, $\frac{cDegrees[C][u]}{degrees[u]}$);
19                 $nc[u]$.add($C$);
20         emit(1);
```

```
Algorithm 2: Pruning bolt: execute($tuple$)
input : A tuple emitted from the processing bolts.
1  begin
2      $processedElements \leftarrow processedElements$+1;
3      if $processedElements \mod W$ == 0 then
4          foreach $C \in communities$ do
5              $C \leftarrow$ prune($C$);
6              log_to_file(sort($C$));
```

incrementing the degrees of its adjacent nodes (Lines 9-10). We use the label of the node as the *key* of a `Redis String` and issue an `INCR` operation. Then, the bolt retrieves from the key-value store the union of communities that the two nodes belong into. In [18], memory limitations enforce going through the entire set of communities sought for every edge. In contrast, in DICES we elect to use a `Redis` cluster that allows for horizontal scaling and so, we can handle the growth of the inverted index maintained for quickly accessing the communities of each node. In this manner, we significantly reduce the overall number of computations made in [18], as most nodes are associated with little or no communities under investigation.

For every community in this union (Line 11), the bolt increments the *community degrees* of the nodes appropriately (Lines 13, 17). The community degree is a metric we have adopted from [18] and serves as an estimation of the probability that a node belongs to a particular community. We employ the most effective variation examined in [18] that increments the community degree of node v by $\frac{cDegrees[C][u]}{degrees[u]}$, where u is a neighbor of v. Again, we employ `Redis Strings` and use a concatenation of the community *ID* and the node label –delimited with a special character– as the key. Note here that, due to its distributed design, DICES does maintain the actual values for both the degrees and community degrees of each node, whereas COEUS used approximations. This allows for increased effectiveness as we show in our experimentation.

Next, the bolt updates the community participation memberships: the node associated with a score is added to the community (Lines 14 & 18 - `Sorted Set`) as well as the community identifier is added to the communities of the node (Lines 15 & 19 - `Set`). In this, we use the score: $\frac{cDegrees[C][v]}{degrees[v]}$. Finally, the bolt emits a tuple with a value of 1 to signal that an edge has been processed appropriately (Line 20).

3) Pruning bolt: The pruning bolt eases the task of computation by removing *irrelevant* nodes from communities under formulation. Every time a window of W encountered edges elapses, we keep only the *top*-100 nodes exhibiting the highest score in each community. We use this threshold as related studies have shown that quality communities do not surpass 100 nodes [24]. Moreover, we set the window W of DICES to 10,000 edges; we derived this value through extensive exploratory testing and found that it consistently works well as far as both efficiency and accuracy are concerned.

Algorithm 2 outlines the operation of a pruning bolt: every time an edge is successfully processed by a processing bolt, the pruning bolt is signaled with a tuple. When the window W elapses (Line 3), we purge all *nodes ranked below* the *top*-100 (Line 5) for every sought community (Line 4). To accomplish this, we first retrieve all these nodes by issuing a `Redis ZREVRANGE`[3] operation. For each of the retrieved nodes, we remove the community from their set of communities. Then, we issue a `Redis ZREMRANGEBYRANK`[4] operation to remove the irrelevant nodes from the community. After purging the nodes ranked below the *top*-100, the community memberships are recorded to a file-log (Line 6). This allows for tracking the evolution of the communities as the stream undergoes processing. The actual size of a community is usually smaller than 100 nodes. Thus, to determine the size of each community, we adopt the `dropTail` technique of [18].

[3]https://redis.io/commands/zrevrange
[4]https://redis.io/commands/zremrangebyrank

Table I: Real-world networks of our dataset reaching up to 1.8 billion edges.

Network	Type	Nodes	Edges	Average Degree
Amazon	Co-purchasing	334,863	925,872	2.76
DBLP	Co-authorship	317,080	1,049,866	3.31
Youtube	Social	1,134,890	2,987,624	2.63
LiveJournal	Social	3,997,962	34,681,189	8.67
Orkut	Social	3,072,441	117,185,083	38.14
Friendster	Social	65,608,366	1,806,067,135	27.53

D. Scalability Analysis

Proposition 1. *The number of messages exchanged in* DiCeS *for edge processing depends solely on* $|E|$.

Proof: For every edge $e \in E$ the DiCeS spout emits one message to a processing bolt and the latter emits one message to the pruning bolt. Both messages are acknowledged. No other communication occurs in DiCeS; thus, the number of total messages is a function of $|E|$ only. ■

For a graph $G = (V, E)$, the total execution time of DiCeS is formulated as $\frac{t_p}{p} + t_s + t_m$, where p denotes the number of workers, t_p is the execution time for the parallel workload, t_s designates the execution time for the serial workload, and t_m outlines the communication cost incurred by the exchange of messages. The DiCeS serial workload comprises the actions of the pruning bolt and is constant for a given graph G. From Proposition 1, t_m is also constant for G. The parallel workload t_p grows with p due to the initialization cost of each worker. However, since the total edge processing cost is independent of p, we can reduce $\frac{t_p}{p}$ by adding workers, as long as $p < |E|$.

IV. Experimental Evaluation

We proceed by evaluating the performance of DiCeS on a range of networks from various domains. Our experiments measure the impact of the novel techniques of our algorithm and feature comparisons against state-of-the-art community detection approaches that use the entire graph. We first discuss the specifications of our experimental setting. Then, we evaluate DiCeS by answering the following questions:

- how fast can DiCeS process the edges of a network stream?
- how does DiCeS scale?
- how do the network characteristics affect DiCeS processing time?
- how does the accuracy of DiCeS compare to the state-of-the-art?

A. Experimental settings

Our dataset comprises the six publicly available networks with ground-truth communities listed in Table I,[5] reaching up to 1.8 *billion edges*. We have adopted the experimental setting of [18] and use the *top-5000* ground-truth communities of each network with regards to their quality [25], after enforcing a community size between 20 and 100.

DiCeS is easily submitted as a topology to any existing `Storm` cluster. Our implementation as well as reproducible execution tests are publicly available.[6] We performed our experiments using docker containers that were deployed on a *Dell PowerEdge R630* server with two Intel®Xeon® E5-2630 v3, 2.40 GHz with 8 cores (16 in total) and 256GB of RAM. We setup a Storm cluster using the official dockerfile[7] to create a `Zookeeper` container, a `Nimbus` container, 8 `Supervisor` containers, and a `UI` container. Each container is provided with a total of 8GB of RAM. Unless otherwise specified, we set the maximum allowed spout pending messages to be 15,000, and the total number of `Storm` workers to be equal to the number of processing bolts plus 2 – one for the spout and one for the pruning bolt. A single `Redis` instance causes an I/O bottleneck when using multiple processing bolts. Therefore, we setup a `Redis` cluster with a total of 3 master nodes and use this cluster as our distributed key-value store in all our experiments.

Our evaluation assumes that 3 *random nodes* of each ground-truth community are provided to each algorithm as an input seed-set. To measure the accuracy of each algorithm we use the average *F1-score* achieved for the communities of each network. All results reported are averages of multiple (10) executions (for various random seed-sets and permutations of the order of edges) and are accompanied with their respective 95% confidence intervals.

B. Performance, Scaling, and Fault Tolerance

We begin with an investigation on the execution time of finite network streams derived from the networks of our dataset. Figure 3 shows the processing time required for each edge per network for settings with 2, 4, and 8 processing bolts. The results shown are averages of multiple (10) executions. We can clearly see that we can reduce the processing time required for each edge by adding bolts to our topology. More specifically, for the *amazon* and *dblp* networks DiCeS scales *almost linearly* from 2 to 8 worker nodes, as the processing time of the latter setting is about 31% of the processing time of the former setting. For the other 4 networks in the dataset, we observe that our processing capacity is *more than quadrupled* when going from to 2 to 8 worker nodes, i.e., the processing time with 8

[5]https://snap.stanford.edu/data/#communities
[6]https://github.com/panagiotisl/DiCeS/
[7]https://hub.docker.com/_/storm/

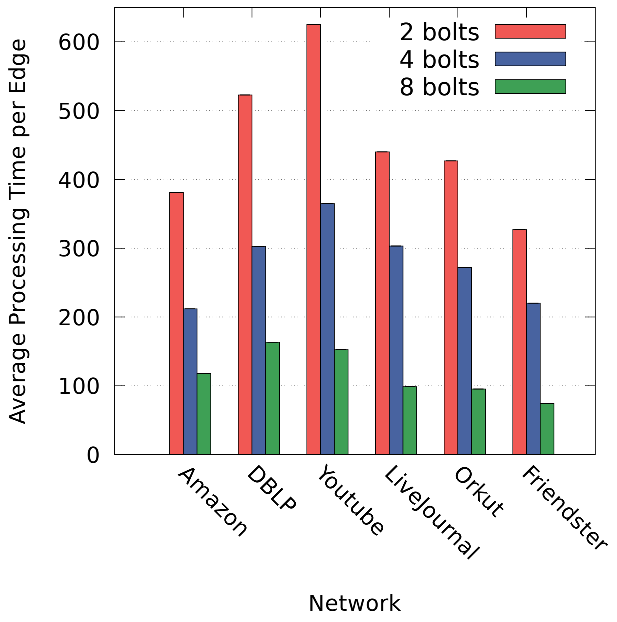

Figure 3: Processing time of an edge (in μs) for the networks of our dataset in settings of 2, 4, and 8 processing bolts. DICES scales horizontally, i.e., the processing time is reduced as we add worker nodes.

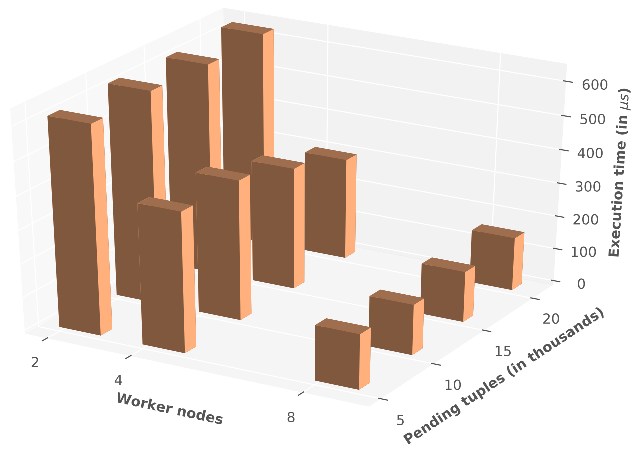

Figure 4: Processing time required per edge for the network of *youtube* for various settings of total worker nodes, and values of maximum allowed pending tuples.

nodes is below 25% of the processing time with 2 nodes. Our architectural design does not justify sublinear scaling, thus, this result is due to other settings involved in the execution environment. Additionally, we observe that when using 4 processing bolts, the execution time ranges between 56-69% of that of 2 bolts. That is, scaling seems to be worse than what we observe when going from 2 to 8 bolts. This is also attributed to execution environment settings rather than our algorithm; below, we examine this very issue.

One key execution setting causing the aforementioned issues is the number of maximum allowed pending tuples. Times of Figure 3 are produced under a setting in which no more than 15,000 tuples remain simultaneously pending. However, the optimal value of this setting varies, depending on the network and the total number of bolts utilized. Figure 4 shows the processing time required–per–edge for the network of *youtube* when the number of maximum allowed *pending* tuples are: 5,000, 10,000, 15,000, and 20,000. Results shown are averages of multiple executions. We observe that there is indeed an impact on the processing time for all 3 settings of bolts, i.e., 2, 4, and 8. When using 2 bolts the processing time ranges between $620 - 635\mu s$. The difference is rather evident when employing 4 bolts, as the processing time then ranges between $303 - 418\mu s$. Finally, when using eight processing bolts, we observe that time ranges between $151 - 165\mu s$. The results of Figure 4 explain why DICES scaling is not perfectly balanced. Nonetheless, we clearly observe in both Figures 3 and 4 that DICES offers *near-linear scaling*.

In the event of a worker failure, DICES initializes a new worker and re-emits the edges that were not acknowledged. Figure 5 shows the processing time required per 10,000 edges for the *youtube* network using 4 processing bolts when failures occur. In particular, we manually "kill" a worker node when DICES has processed 300,000, 600,000, and 900,000 edges. The resulting spikes in the performance of Figure 5 occur due to a total of 46,361 failed edges that need to be re-emitted, as well as the fact that only 3 bolts are processing edges until a new bolt replaces the one we "killed". We clearly see that DICES always recovers its processing speed in just a few seconds.

C. *Impact of average degree and number of communities*

Our next experiment investigates how the network's average degree and the number of communities we wish to uncover impact the average processing time per edge of DICES. We create 2 synthetic networks of 1 million nodes each, using the Lancichinetti-Fortunato-Radicchi (LFR) benchmark [26].[8] The latter produces networks as well as their community structure. The 2 networks are generated to exhibit an average degree of 10 and 20, respectively. For each synthetic network we randomly select 4,000 of the generated communities to use in this experiment.

Figure 6 shows that for a given average degree of the network, increasing the number of communities we seek from 2,000 to 4,000, results in increased processing time–per–edge. However, we observe that the increase of DICES is not as significant as it is with COEUS. Thank to its

[8]We use a mixing parameter of 0.1, a maximum degree of 100, and communities with $20 - 100$ member nodes.

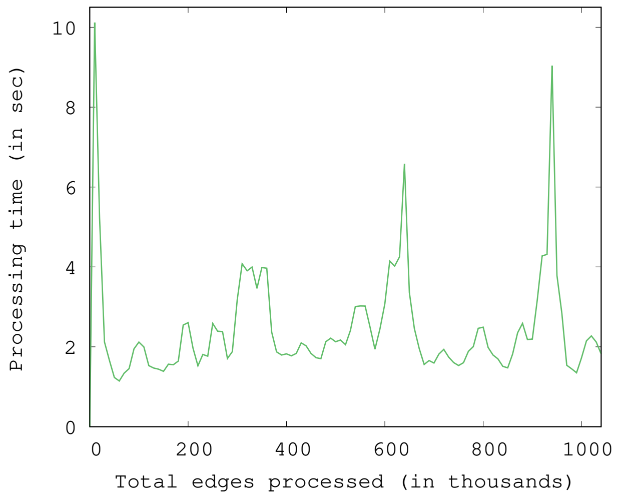

Figure 5: Impact of failed workers on processing time. We observe spikes in the performance each time a worker fails, i.e., when DICES has processed 300,000, 600,000, and 900,000 edges. However, DICES always recovers its processing speed in just a few seconds.

inverted index, DICES does not waste time examining communities that are irrelevant to a particular edge, as CoEuS does. Therefore, the increase is proportional to the nodes' involvement in the communities we seek, instead of the total number of communities. Moreover, DICES is able to scale out and adjust to a possible increased demand in the number of communities we wish to uncover. By using 8 bolts, the average DICES processing time is significantly lower than that of COEUS.

Figure 6 also shows that for a given number of communities we wish to uncover, the processing time is slightly reduced for DICES and is relatively stable with COEUS when the average degree increases from 10 to 20. This highlights the impact of our pruning step to the processing time–per–edge. In particular, each community we process is periodically cut down to 100 member nodes. Consequently, increasing the average degree results to more edges with adjacent nodes that are not involved in any community. Therefore, the average processing time per edge is reduced. The improved performance of DICES is again due to the use of our inverted index.

D. F1–score comparison of DICES *with the centralized streaming* COEUS *algorithm, and the non-streaming* LEMON *algorithm.*

We have demonstrated the merits of our architecture in terms of scalability. We now focus on the effectiveness of our algorithm in accurately uncovering communities. Figure 7 compares DICES against COEUS [18]. We initialize both approaches with 3 random seeds of each ground-truth community of our dataset and calculate the average *F1–score*

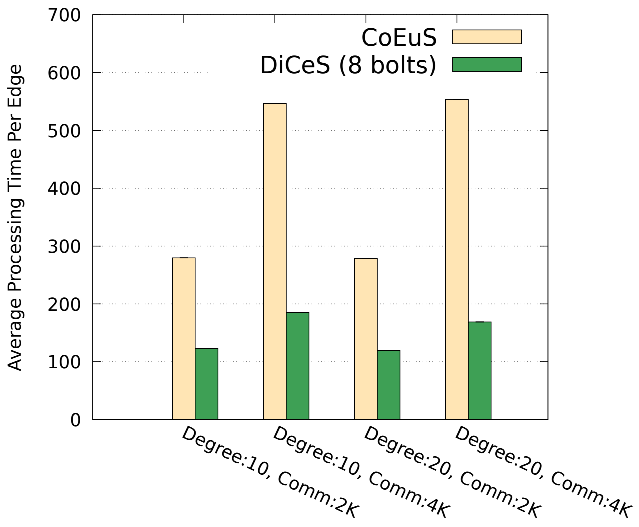

Figure 6: Impact of average degree and number of communities on the average processing time per edge (in μs).

achieved for all ground-truth communities of each network. The reported results are averages of multiple executions for random seed selections and random edgelist orderings. We remind the reader that COEUS exhibits impressive accuracy that is equivalent or better than state-of-the-art *non-streaming* seed-set expansion approaches [13], [12], [27], [9] that exploit the whole network and consequently cannot handle large-scale networks. We include results for LEMON [13] to make this more evident.

Figure 7 shows that the performance of the two approaches is equivalent for the 3 smaller networks of our dataset, namely *amazon*, *dblp*, and *youtube*. However, DICES outperforms COEUS for the 3 larger ones, i.e., *livejournal*, *orkut*, and *friendster*, offering significantly improved *F1–scores*. The improvement is mostly attributed to the different accuracy the two approaches have with regards to maintaining the nodes' degrees and community degrees. DICES uses a `Redis` cluster and is aware of the exact values. In contrast, COEUS employs COUNT-MIN sketches due to memory constraints. The sketches are configured to provide 99% confidence that $\epsilon < 10^{-5}$. The estimation error is proportional to ϵ as well as the total aggregate number of edges seen. By definition $cDegrees[C][u] \leq degrees[u]$; hence, the inaccuracy of the sketches progressively produces smaller values for $\frac{cDegrees[C][u]}{degrees[u]}$.

For the *livejournal* network, using the exact values with DICES results in improved performance with regards to *F1–score*. For the *orkut* and *friendster* networks, we initially found out that the approximations produced by the COUNT-MIN sketches of COEUS actually help achieving greater *F1–scores*. These networks exhibit large average degrees and the communities discovered quickly surpass 100 nodes. The decaying values produced by the sketches favor the nodes

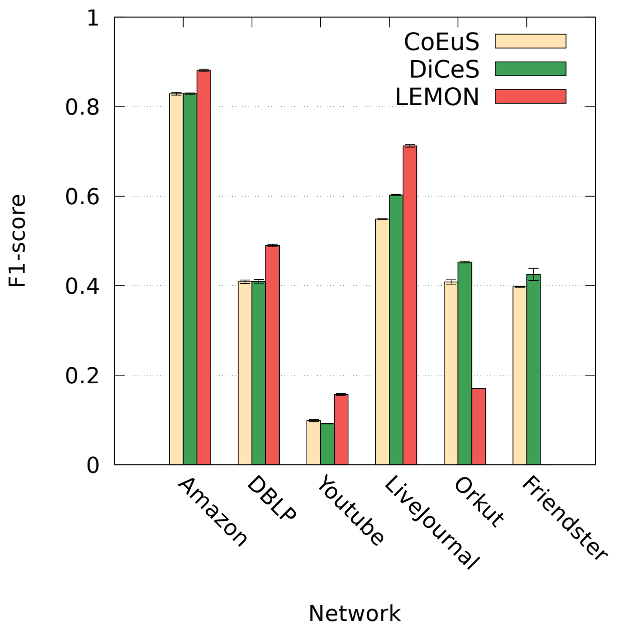

Figure 7: F1–score comparison.

that were added early on, as they are *close* to the seeds, and result in pruning more irrelevant nodes. We noticed that improving the accuracy of the sketches actually results in worse *F1–scores* for the two largest networks of our dataset. This effect can of course be easily reproduced with DICES, by adding a similar decay factor that is proportional to the total aggregate number of edges seen. In fact with DICES, we can achieve a more balanced decay than with COEUS as there is no ϵ. This allows improved control over this parameter and helps us achieve larger *F1–scores* for both networks with DICES.[9] We plan to investigate this effect further in the future, as this work is focused on the execution in the cloud.

V. Related Work

COEUS [18] processes a network stream in a centralized machine and expands seed-sets to communities by adding nodes adjacent to community members. Two novel techniques are introduced for i) estimating the involvement of a node in each community, and ii) determining the size of each community. Experimental results show that COEUS achieves accuracy comparable to state-of-the-art non-streaming approaches while being impressively efficient. Our work builds on techniques of COEUS to provide the first cloud application for detecting communities in network streams and to achieve further improvements with regard to accuracy.

Yun et al. [16] consider settings in which the size of the network is so large that maintaining the respective graph is prohibitive. They study the problem of clustering the nodes of a graph to communities in a streaming setting where rows of the adjacency matrix of the graph are revealed sequentially. They propose an online algorithm with space complexity that grows sub-linearly with the size of the network. Our setting does not assume that complete rows of the adjacency matrix are revealed to us. Instead, we consider that edges involving any node of the network may arrive at any moment. Moreover, our approach is unaware of the size of the graph, which grows with time. An edge streaming setting is considered in [17]. All nodes of a graph are assigned to non-overlapping communities using only two integers per node that hold: i) the node's degree, and ii) the current community index assigned to the node. The work is based on the observation that if we pick uniformly at random an edge of the graph, this edge is more likely to link nodes of the same community, than nodes from distinct communities. This is expected to be true as nodes tend to be more connected within a community than across communities, thus, if we process edges in a random order we expect many intra-community edges to arrive before the inter-community edges. However, this requires that we already hold the graph in its entirety and we can select its edges one by one uniformly at random. We operate on the more practical assumption that the edges of the graph arrive at no particular order.

VI. Conclusions

We propose and implement DICES, a streaming community detection virtual infrastructure for large-scale networks that evolve rapidly. DICES addresses various limitations imposed by the execution setting of our earlier COEUS algorithm [18]. DICES distributes the load to worker nodes in the cloud and easily adapts to workload changes by adjusting the number nodes. In the event of a failure, DICES resubmits network edges that were not processed and restarts nodes that "died". Equally critical is the fact that we provide interactivity in terms of dynamically updating the communities under investigation and requesting the current state of the communities on demand. We investigate the performance of our framework using both real-world and synthetic networks. Our findings show that we can process almost 50 million *edges per hour* using only 8 worker nodes. We can easily increase our processing capacity as DICES is shown to scale *almost linearly*. In this regard, we handily outperform prior approaches that do not scale out while trying to address the problem of community detection. Finally, the increased memory resources of our distributed setting realize further improvements with regard to accuracy in detecting the communities existing in 6 real-world networks.

Acknowledgments

Partial support for this work has been provided by the European Union Horizon 2020 Programme, project "Galena",

[9]For both networks we add a decay factor of $\frac{total\ aggregate\ value}{150{,}000}$.

under grant agreement № 641515. Alexandros Ntoulas contributed to this work prior joining LinkedIn.

References

[1] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, 2008.

[2] A. Clauset, M. E. Newman, and C. Moore, "Finding community structure in very large networks," *Physical review E*, vol. 70, no. 6, p. 066111, 2004.

[3] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. E*, vol. 69, no. 2, p. 026113, Feb. 2004.

[4] P. Pons and M. Latapy, "Computing communities in large networks using random walks," in *Computer and Information Sciences-ISCIS 2005*, 2005, pp. 284–293.

[5] T. Evans and R. Lambiotte, "Line graphs, link partitions, and overlapping communities," *Physical Review E*, vol. 80, p. 016105, 2009.

[6] Y.-Y. Ahn, J. P. Bagrow, and S. Lehmann, "Link communities reveal multiscale complexity in networks," *Nature*, vol. 466, no. 7307, pp. 761–764, 2010.

[7] D. F. Gleich and C. Seshadhri, "Vertex neighborhoods, low conductance cuts, and good seeds for local community methods," in *Proc. of the 18th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, 2012, pp. 597–605.

[8] J. Yang and J. Leskovec, "Community-affiliation graph model for overlapping network community detection," in *Proc. of the 12th IEEE Int. Conf. on Data Mining*, 2012, pp. 1170–1175.

[9] J. J. Whang, D. F. Gleich, and I. S. Dhillon, "Overlapping community detection using seed set expansion," in *Proc. of the 22nd ACM Int. Conf. on Information & Knowledge Management*, 2013, pp. 2099–2108.

[10] J. Yang and J. Leskovec, "Overlapping community detection at scale: a nonnegative matrix factorization approach," in *Proc. of the 6th ACM Int. Conf. on Web Search and Data Mining*, 2013, pp. 587–596.

[11] P. Liakos, K. Papakonstantinopoulou, and A. Delis, "Realizing memory-optimized distributed graph processing," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 4, 2018.

[12] K. Kloster and D. F. Gleich, "Heat kernel based community detection," in *The 20th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, New York, NY, USA*, 2014, pp. 1386–1395.

[13] Y. Li, K. He, D. Bindel, and J. E. Hopcroft, "Uncovering the small community structure in large networks: A local spectral approach," in *Proc. of the 24th Int. Conf. on World Wide Web*, 2015, pp. 658–668.

[14] K. He, Y. Sun, D. Bindel, J. E. Hopcroft, and Y. Li, "Detecting overlapping communities from local spectral subspaces," in *IEEE Int. Conf. on Data Mining, Atlantic City, NJ, USA*, 2015, pp. 769–774.

[15] P. Liakos, A. Ntoulas, and A. Delis, "Scalable link community detection: A local dispersion-aware approach," in *2016 IEEE Int. Conf. on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*, pp. 716–725.

[16] S. Yun, M. Lelarge, and A. Proutière, "Streaming, memory limited algorithms for community detection," in *Annual Conf. on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, 2014, pp. 3167–3175.

[17] A. Hollocou, J. Maudet, T. Bonald, and M. Lelarge, "A linear streaming algorithm for community detection in very large networks," *ArXiv e-prints*, Mar. 2017.

[18] P. Liakos, A. Ntoulas, and A. Delis, "CoEuS: Community detection via seed-set expansion on graph streams," in *2017 IEEE Int. Conf. on Big Data, BigData 2017, Boston, MA*, 2017, pp. 676–685.

[19] S. Chintapalli, D. Dagit, R. Evans, R. Farivar, Z. Liu, K. Nusbaum, K. Patil, and B. Peng, "Pacemaker: When zookeeper arteries get clogged in storm clusters," in *9th IEEE Int Conf. on Cloud Computing, CLOUD 2016, San Francisco, CA*, 2016, pp. 448–455.

[20] K. Tsakalozos, H. Kllapi, E. Sitaridi, M. Roussopoulos, D. Paparas, and A. Delis, "Flexible use of cloud resources through profit maximization and price discrimination," in *Proc. of the 27th Int. Conf. on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, 2011, pp. 75–86.

[21] M. Meredith and B. Urgaonkar, "Towards performance modeling as a service by exploiting resource diversity in the public cloud," in *9th IEEE Int. Conf. on Cloud Computing, CLOUD 2016, San Francisco, CA*, 2016, pp. 204–211.

[22] K. Ouaknine, O. Agra, and Z. Guz, "Optimization of rocksdb for redis on flash," in *Proc. of the Int. Conf. on Compute and Data Analysis*. ACM, 2017, pp. 155–161.

[23] M. D. Da Silva and H. L. Tavares, *Redis Essentials*. Packt Publishing, 2015.

[24] J. Yang and J. Leskovec, "Structure and overlaps of ground-truth communities in networks," *ACM Transactions on Intelligent Systems and Technology*, vol. 5, no. 2, p. 26, 2014.

[25] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Statistical properties of community structure in large social and information networks," in *Proc. of the 17th Int. Conf. on World Wide Web*, ser. WWW '08, 2008, pp. 695–704.

[26] A. Lancichinetti, S. Fortunato, and F. Radicchi, "Benchmark graphs for testing community detection algorithms," *Physical review E*, vol. 78, no. 4, p. 046110, 2008.

[27] I. M. Kloumann and J. M. Kleinberg, "Community membership identification from small seed sets," in *Proc. of the 20th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pp. 1366–1375.