Scalable Dynamic Graph Summarization

Ioanna Tsalouchidou, Francesco Bonchi, Gianmarco De Francisci Morales, and Ricardo Baeza-Yates, *Fellow, IEEE*

Abstract—Large-scale dynamic interaction graphs can be challenging to process and store, due to their size and the continuous change of communication patterns between nodes. In this work, we address the problem of summarizing large-scale dynamic graphs, while maintaining the evolution of their structure and interactions. Our approach is based on grouping the nodes of the graph in supernodes according to their connectivity and communication patterns. The resulting summary graph preserves the information about the evolution of the graph within a time window. We propose two online algorithms for summarizing this type of graphs. Our baseline algorithm kC based on clustering is fast but rather memory expensive. The second method we propose, named μC, reduces the memory requirements by introducing an intermediate step that keeps statistics of the clustering of the previous rounds. Our algorithms are distributed by design, and we implement them over the Apache Spark framework, so as to address the problem of scalability for large-scale graphs and massive streams. We apply our methods to several dynamic graphs, and show that we can efficiently use the summary graphs to answer temporal and probabilistic graph queries.

Index Terms—Graph summarization, dynamic graphs, clustering

1 Introduction

In a variety of application domains such as social networks, molecular biology, and communication networks, the data of interest is routinely represented as a very large graph with millions of vertices and billions of edges. This abundance of data can potentially enable more accurate analysis of the phenomena under study. However, as the graphs under analysis grow, mining and visualizing them becomes computationally challenging. In fact, the running time of most graph algorithms grows with the size of the input (number of vertices and/or edges): executing them on huge graphs might be impractical, especially when the input is too large to fit in main memory. The picture gets even worse when considering the dyna'mic nature of most of the graphs of interest, such as social networks, communication networks, or the Web.

Graph summarization speeds up the analysis by creating a *lossy concise representation of the graph* that fits into memory. Answers to otherwise expensive queries can then be computed by using the summary without accessing the exact representation on disk. Query answers computed on the summary incur a minimal loss of accuracy. When multiple graph analysis tasks are performed on the same summary, the cost of building the summary is amortized across its life cycle. Summaries can also be used for privacy purposes [9], to create easily interpretable visualizations of the graph [13], or to store a compressed version of the graph [15].

- *I. Tsalouchidou and R. Baeza-Yates are with Pompeu Fabra University, Barcelona 08002, Spain. E-mail: ioanna.tsalouchidou@upf.edu, rbaeza@acm.org.*
- *F. Bonchi and G. De Francisci Morales are with the ISI Foundation, Turin 10126, Italy. E-mail: francesco.bonchi@isi.it, gdfm@acm.org.*

Manuscript received 28 Jan. 2018; revised 8 Nov. 2018; accepted 15 Nov. 2018. Date of publication 30 Nov. 2018; date of current version 8 Jan. 2020. (Corresponding author: Francesco Bonchi.) Recommended for acceptance by L. Chen. Digital Object Identifier no. 10.1109/TKDE.2018.2884471

In this paper we tackle the problem of *building high quality summaries for dynamic graphs*. In particular, we aim at creating summaries of a dynamic graph over a sliding window of a given size w. At every new timestamp, as the graph evolves, the time window of interest includes a new adjacency matrix and discards the oldest one that occurred w timestamps ago.

We consider a general setting where each entry of the adjacency matrix at every timestamp contains a number in $[0, 1]$. This can be used to model interaction networks, where the entry (i, j) of the adjacency matrix at time t can indicate the strength of the link or the amount of information exchange between i and j during the timestamp t. From the classic dynamic graph standpoint, once an edge (i, j), with weight 0 up to timestamp t, takes a weight > 0, it is considered to *appear* for the first time at t. Similarly an edge that starts having weight 0 after t, can be considered to *disappear* after time t.

In this paper we introduce a new version of the *dynamic graph summarization* problem, by generalizing the definition of LeFevre and Terzi [9] (discussed next) to the dynamic graph setting in a streaming context.

Our main contributions can be summarized as follows:

- We introduce the problem of *dynamic graph summarization* in a streaming context by generalizing the problem definition for static graphs of LeFevre and Terzi [9].
- We design two online, distributed, and tunable algorithms for summarizing dynamic large-scale graphs. The first one is inspired by Riondato et al. [15] and is based on clustering. The second one overcomes the memory requirements limitation of the first one by using the *micro-clusters* concept from Aggarwal et al. [3], adapted to our graph-stream setting.

- Our algorithms are distributed by design, and we implement them over the Apache Spark framework to address the problem of scalability for massive graph streams.
- We experiment on several real-world and synthetic dynamic graphs, showing that we can effectively and efficiently use our summaries to answer temporal and probabilistic queries on the dynamic graphs.

The rest of the paper is organized as follows. In Section 2, we provide a small survey of the related work. In Section 3, we give the preliminary definitions and the formal problem statement. In Section 4, we present the two algorithms in full details. In Section 5, we discuss the distributed implementation on Apache Spark. In Section 6, we present our empirical evaluation. Finally, we conclude our work in Section 7 and we discuss future work.

A preliminary version of this work was presented in [22]. This work is enriched with technical details and extensive description of our methods, and a presentation of the distributed version of our algorithms. Additionally, we extend the evaluation with a query section that demonstrates the usefulness of our results. Finally, we provide a description of the synthetic datasets used in the experimental evaluation, and the algorithm to generate them.

2 Background and Related Work

As we are the first to study dynamic graph summarization in a streaming context, there is no prior art on this exact problem. However, we extend existing definitions for static graph summarization and we adopt methods coming from data stream clustering literature. Therefore, in the following we cover these two areas of research.

Graph Summarization. LeFevre and Terzi [9] propose to use an enriched "supergraph" as a summary. The supergraph has an integer for each supernode (set of vertices) and for each superedge (an edge between two supernodes), which represent respectively the number of edges between vertices in the supernode in the original graph, and between the two sets of vertices connected by the superedge. From this lossy representation one can infer an *expected adjacency matrix*, where the expectation is taken over the set of *possible worlds* (i.e., graphs that are compatible with the summary). Thus, from the summary one can derive approximated answers for graph property queries. Their method follows a greedy heuristic resembling an agglomerative hierarchical clustering with no quality guarantee.

Riondato et al. [15] build on the work of LeFevre and Terzi [9] and, by exposing a connection between graph summarization and geometric clustering problems (i.e., k-means and k-median), they propose a clustering-based approach to produce lossy summaries of given size with quality guarantees. Their approach is based on minimizing the error while reconstructing the original graph from the summary.

Navlakha et al. [13] propose a summary consisting of two components: a graph of "supernodes" (sets of nodes) and "superedges" (sets of edges), and a table of "corrections" representing the edges that should be removed or added to obtain the exact graph. Liu et al. [10] follow the definition of [13] and present the first distributed algorithm for summarizing large-scale graphs. A different approach followed by Tian et al. [20] and Liu et al. [11], for graphs with labeled vertices, is to create "homogeneous" supernodes, i.e., to partition the graph so that vertices in the same set have, as much as possible, the same attribute values. In [20] the authors propose SNAP, which summarizes graphs by gathering groups of nodes that share the same categorical attributes. Furthermore, nodes inside groups are adjacent for all types of relationships with the nodes of the same group (attribute- and relationship-compatibility). k-SNAP further extends SNAP and allows the users to control the size of their summaries, by relaxing the homogeneity requirement for the relationships. Zhang et al. [23] build on k-SNAP and propose CANAL which automatically categorizes numerical attribute values by exploiting their similarities and the link structure of the nodes of the graph.

The work by Adler and Mitzenmacher [2], Suel and Yuan [17], and Boldi and Vigna [5] discuss techniques to compress the Web graph so as to reduce the bits used to encode the links without any loss in information. Shah et al. [16] approach the problem of graph summarization as a (lossless) compression problem, and further extend it to dynamic graphs. Adhikari et al. [1] propose a node-grouping technique with diffusion-equivalent representation on dynamic graphs. Other approaches by Tang et al. [19], Khan and Aggarwal [8] and Qu et al. [14], include graph sketches, which are general purpose synopsis that maintain structural and frequency properties of graph streams or summarizing dynamic networks by capturing only some of the most interesting nodes and edges over time. Sun et al. [18] propose an incremental algorithm for dynamic tensor analysis which aims at dimensionality reduction to produce compact summaries for high-order and high-dimensional data.

Toivonen et al. [21] propose an approach for graph summarization tailored to weighted graphs, which creates a summary that preserves the distances between vertices. Fan et al. [6] present two different summaries, one for reachability queries and one for graph patterns. Hernandez and Navarro [7] focus instead on neighbor and community queries, and Maserrat and Pei [12] just on neighbor queries. These proposals are highly query-specific, while our summaries are general-purpose and can be used to answer different types of queries.

Data Stream Clustering. Aggarwal et al. [3] study the problem of clustering evolving data streams over different time horizons. They use *micro-clusters* that provide spatial and temporal information of the *evolving* streams that are used for a horizon-specific offline clustering. Micro-clusters are a temporal extension of *cluster feature vectors* introduced by Zhang et al. [24] in their *BIRCH* method. Micro-clusters maintain statistical information about the data locality of the nodes. Their additivity property make them an adequate choice for clustering data streams. The snapshots in which the micro-clusters are stored, follow the *Pyramidal Time Frame*, which is a technique used to store data at different levels of granularity based on their arrival time.

As mentioned before, Shah et al. [16] deal with the problem of lossless dynamic-graph compression. Instead, we tackle the problem of lossy summarization of dynamic graphs. By contrast, our goal is to develop a summary that, while small enough to be stored in limited space (e.g., in main memory), can also be used to compute approximate

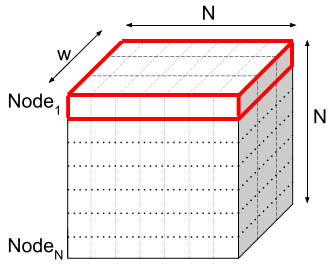


Fig. 1. Order 3 tensor of dimensions $N \times N \times w$ where N is the number of nodes and w is the length of the window W. One of the nodes of the tensor ($Node_1$) is highlighted.

but fast answers to queries about the original graph. The summaries we produce have a simpler topology than the input graph, and can be used as substitutes at the cost of introducing an error. Our algorithms are distributed by design with scalability as main goal. Differently from the work by Liu et al. [10], the task distribution of our algorithm does not create dependencies or requirements for hand-crafted message-passing.

3 Problem Formulation

In this section we first define the problem of static graph summarization. We then present the problem of dynamic graph summarization in tensors with sliding windows.

3.1 Static Graph Summarization

Given an undirected, simple, weighted graph $G(V, E, \eta)$ with $V = \{V_1, \ldots, V_N\}$, a *weight function* η: $E \rightarrow [0, 1]$, and $k \in \mathbb{N}$ ($k \leq N$); a k-summary of G is an undirected, complete, weighted graph $G'(S, S \times S, \sigma)$ uniquely identified by a k-partition of V, i.e., $S = \{S_1, \ldots, S_k\}$, with $\bigcup_{i \in [1,k]} S_i = V$ and $S_i \cap S_j = \varnothing$ if $i \neq j$. The function σ: $S \times S \rightarrow [0, 1]$ maintains the average edge weight among the nodes contained in two supernodes, and is given by

$$\sigma(S_i, S_j) = \frac{\sum_{u \in S_i, v \in S_j} \eta(u, v)}{|S_i||S_j|}, \text{ for } S_i \neq S_j,$$

and

$$\sigma(S_i, S_i) = \frac{\sum_{u, v \in S_i} \eta(u, v)}{|S_i||S_i - 1|}, \text{ for } |S_i| > 1.$$

In the case of $|S_i| = 1$ we have $\sigma(S_i, S_i) = 0$ by definition. For ease of presentation, in the rest of the paper we define the main concepts using the adjacency matrices of G and G', denoted as A_G and $A_{G'}$, respectively.

We can find as many k-summaries as the number of k-partitions of the nodes V. Following LeFevre and Terzi [9], the goal is to find the summary G' that minimizes the *reconstruction error*. That is, the error incurred by reconstructing our best guess of the base graph G from the summary G':

$$RE(A_G|A_{G'}) = \frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1}^{N} |A_G(V_i, V_j) - A_{G'}(\Phi(V_i), \Phi(V_j))|,$$

where Φ is the mapping function from nodes to the supernodes they belong to. For simplicity, in the formula above, we use the entire adjacency matrix of the graph. However, since the graphs we consider are undirected, we could also have used the half (triangular) matrix.

Riondato et al. [15] show that the problem of minimizing the reconstruction error with guaranteed quality can be approximately reduced to a traditional k-means clustering problem, where the elements to be clustered are the adjacency lists of each node. The resulting clusters are then used as the supernodes.

3.2 Tensor Summarization

Given a window $W = [t - (w - 1), t]$, where t is a timestamp of a potentially infinite temporal domain T, we consider next a time series of $w = |W|$ static graphs as described before. The time series of static graphs can be expressed as a time series of adjacency matrices $A_{G^t} \in [0, 1]^{NN}$, or as an order 3 tensor $\mathcal{A}_G^W \in [0, 1]^{NNw}$, as depicted in Fig. 1. Similarly to the static graph case, given $k \leq N$ we define as k-summary of the tensor $\mathcal{A}_G^W$ the adjacency matrix $A_{G'} \in [0, 1]^{kk}$ which is uniquely identified by a k-partition $S = \{S_1, \ldots, S_k\}$ of V:

$$A_{G'}(S_i, S_j) = \frac{\sum_{t=1}^{w} \sum_{u \in S_i, l \in S_j} \mathcal{A}_G^W(u, v, t)}{w|S_i||S_j|}, \text{ for } S_i \neq S_j, \quad (1)$$

and

$$A_{G'}(S_i, S_i) = \frac{\sum_{t=1}^{w} \sum_{u, v \in S_i} \mathcal{A}_G^W(u, v, t)}{w|S_i||S_i - 1|}, \text{ for } |S_i| > 1. \quad (2)$$

In the case of $|S_i| = 1$, we have $A_{G'}(S_i, S_i) = 0$ by definition. The reconstruction error for tensor summarization is defined as follows:

$$RE(\mathcal{A}_G^W|A_{G'}) = \frac{\sum_{t=1}^{w} \sum_{i=1}^{N} \sum_{j=1}^{N} |\mathcal{A}_G^W(V_i, V_j, t) - A_{G'}(\Phi(V_i), \Phi(V_j))|}{wN^2}. \quad (3)$$

3.3 Dynamic Summarization via Tensor Streaming

In the streaming setting we are given a streaming graph (an infinite sequence of static graphs) and a window length w. The goal is to produce a tensor summary for the latest w timestamps.

More formally, we are given a graph stream $\mathcal{G}^t(V, E, f)$, described by its set of nodes $V = \{V_1, \ldots, V_N\}$, edges $E \subseteq V \times V$ and a function f: $E \times T \rightarrow [0, 1]$. This can be represented as a time series of adjacency matrices where each adjacency matrix $A_{G^t} \in [0, 1]^{NN}$. At each time stamp $t \in T$ we have a new adjacency matrix as input, which represents the last instance of the dynamic graph. As time passes by, the information contained in old adjacency matrices can become obsolete and no longer interesting. Therefore, we define a window W_t of fixed length w, that limits our interest to the w more recent instances of the dynamic graph. We refer to this window as a *sliding tensor window*, which is updated at each timestamp with the latest adjacency matrix while the oldest adjacency matrix is removed. Fig. 2 shows the tensor window that indicates which timestamp are considered for the summarization, for three successive timestamps.

At each time stamp t, we summarize the adjacency matrices that are included in the tensor window, i.e., the tensor $\mathcal{A}_G^{W_t} \in [0, 1]^{NNw}$, where $W_t = [t - (w - 1), t]$. The tensor

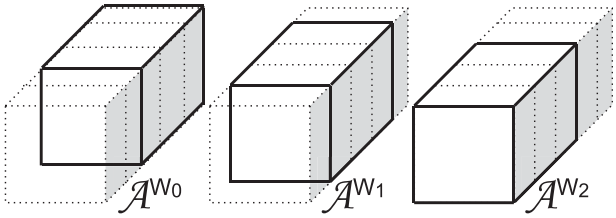


Fig. 2. Sliding tensor-window in three consecutive timestamps. At every timestamp, the window slides one position to include the newest snapshot of the graph and remove the oldest one.

summary is defined as in Section 3.2 by minimizing the reconstruction error of Eq. (3). Finally, the reconstructed tensor $A^{W_t}_{G'}$ is computed from Eqs. (1) and (2).

4 Algorithms

In this section we first describe our baseline clustering-based algorithm inspired by Riondato et al. [15], kC, which is effective but memory expensive. Then, we show the more memory efficient and scalable μC, which is based on the idea of micro-clusters by Aggarwal et al. [3].

4.1 Baseline Algorithm: kC

Following Riondato et al. [15], we apply the k-means algorithm to cluster the nodes of the graph and thus produce the summary of the tensor that is currently inside the sliding window of length w. Fig. 1 shows a tensor window of length w, and highlights one of the matrices ($Node_1$) that are the input for the clustering algorithm. We treat each matrix as a $w \times N$ vector for the purpose of clustering. After clustering these vectors, each cluster represents a supernode of the summary graph.

Algorithm 1 describes kC. For timestamp $t = 0$, we initialize the tensor window (lines 1 & 2) and continue with the computation of the summary (lines 4 to 9). The rest of the algorithm (lines 3 to 10) describes the streaming behavior of the algorithm for the following timestamps. A high-level overview of the process is shown in Fig. 3a.

Since the algorithm needs to work in a high-dimensional space, we prefer to use cosine distance rather than euclidean distance to measure the distance between two data points [4]. This variant of k-means is also known as *spherical* k-means. The input graph changes continuously, as a new adjacency matrix arrives at each timestamp (line 4). Additionally, at each timestamp the tensor window slides to include the newly arrived adjacency matrix (line 5), and exclude the oldest one, as shown in Fig. 2.

Algorithm 1. kC

input: Graph $G^t(V, E)$ as $A_{G^t} \in [0,1]^{NN}$, number of supernodes k, length w of window
output: Summary graph $G'(S, S \times S)$ as $A'_G \in [0,1]^{kk}$, function $\Phi : V \to S$

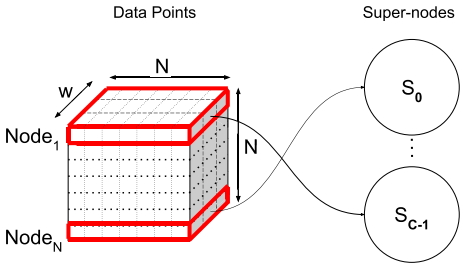
1 $t \leftarrow 0$
2 $\mathcal{A}^{W_0} \leftarrow$ Initialize the adjacency tensor window with zero
3 **while** true **do**
4 $\quad A \leftarrow$ Read input graph A_{G^t}
5 $\quad \mathcal{A}^{W_t} \leftarrow$ Slide window and update with A
6 $\quad C \leftarrow k$-means($\mathcal{A}^{W_t}$)
7 $\quad \Phi \leftarrow$ Calculate mapping function from nodes to supernodes
8 $\quad A^{W_t}_{G'} \leftarrow$ Calculate summary from C `// Equations (1) & (2)`
9 $\quad$ **report** $(A^{W_t}_{G'}, \Phi)$
10 $\quad t \leftarrow t + 1$

Computational Complexity and Limitations. Computing the cosine distance between two Nw-dimensional vectors requires $\mathcal{O}(Nw)$ time. The clustering algorithm computes the distance of each of the N vectors to the center of each of the k clusters. Let the number of iterations for the k-means be bounded by I. Thus, the computational complexity of the algorithm for a single tensor window is $\mathcal{O}(N^2wkI)$. The space requirement is $\mathcal{O}(N^2w + Nwk)$, where the first term accounts for the tensor window, and the second for the clusters' centroids.

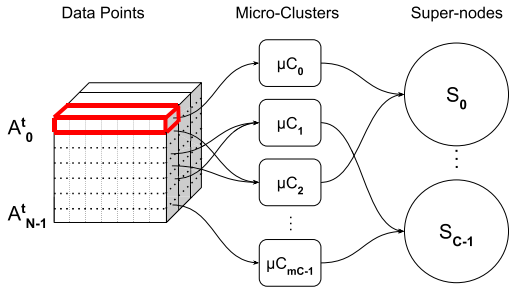
We repeat the same procedure at each new timestamp without taking into account that the tensor window is updated with N^2 new values, and drops N^2 old values, whereas $(w-2)N^2$ values of the window remain unchanged. Clearly, although it is desirable to leverage this fact, the baseline algorithm described so far fails to do so. Indeed, kC simply discards the previous computation, and re-executes the algorithm from scratch. In the next algorithm we show how to take advantage of this optimization.

4.2 Micro-Clustering Algorithm: μC

The key idea towards space-efficiency and scalability is to make use of the clustering obtained at the previous



(a) kC approach: summarizing a tensor window to supernodes.



(b) μC approach: Summarizing a tensor window by micro-clusters.

Fig. 3. Overview of the clustering process of (a) kC algorithm and (b) μC algorithm. In the kC approach, every node is clustered to the supernodes. In the μC approach at each timestamp, t all the A^t_i (highlighted with red) are clustered to the micro-clusters. The micro-clusters include statistical information from the previous timestamps. Finally, the micro-clusters are clustered to the supernodes.

timestamp, by updating it to match the new information arrived, instead of recomputing it from scratch at every new timestamp. To this end, we add an extra intermediate step in between the input step and the final clustering that creates the supernodes, consisting in summarizing the input data via *micro-clusters*. At any given time, the algorithm maintains a fixed amount of micro-clusters q that is set to be significantly larger than the number of clusters k, and significantly smaller than the number of input vectors N. Each micro-cluster (μC) is characterized by its centroid and some statistical information about the input vectors it contains in a concise representation (described further).

The centroid of the micro-cluster (μc) is an (Nw)-dimensional vector that is the mean value of the coordinates of the vectors it contains. The statistics of the micro-cluster include the standard deviation (SD) of the vectors from the centroid, and the frequencies F of the nodes that are included in the micro-cluster. In addition to the structure of the micro-cluster, μC also keeps the IDs of the nodes contained in the last tensor window. For each node we also keep the timestamps ($IDList$) in which the node is contained in the micro-cluster (within the period w of the current window).

Definition 1. *A micro-cluster μC_i is the tuple (F, μc, SD, $IDList$), where the entries are defined as follows:*

- *F is a vector of length w that gives the number of vectors that are included in the micro-cluster i at each timestamp in the current window.*
- *μc is the centroid of the micro-cluster, which is represented by a vector $\in [0,1]^{Nw}$. The centroid is the mean of the coordinates of all the vectors included in the micro-cluster.*
- *SD represents the standard deviation of the distances of all the vectors that are included in the micro-cluster from its centroid in the latest timestamp.*
- *$IDList$ is a list of tuples (NodeID, $BitMap_{ID}$) that stores the IDs of the nodes that are included in the micro-cluster, along with a bitmap of length w that represents the timestamps in which the node was included in the micro-cluster. The least significant bit represents the latest arrival. The sum of the bits of the bitmaps with the same ID in all existing micro-clusters is constant and equal to w.*

Algorithm 2 describes the different steps of μC for every timestamp (lines 3 to 10). The input of the algorithm is an adjacency matrix A_{G^t} that corresponds to the graph G^t of the current timestamp. Fig. 3b shows the tensor window of length w and highlights one of the N vectors A_0^t of the input for the clustering algorithm. The algorithm does not keep the input data that arrived in the previous $w-1$ time stamps, since it only uses statistical information that is stored in the micro-clusters. Micro-clusters are initialized by executing a modified k-means algorithm for the initial adjacency matrix A_{G^t}, similar to what is described before. At this point the seeds of the k-means algorithm are selected randomly from the input vectors. The same procedure is followed at every timestamp to reflect the changes in the sliding window (line 4). Once the micro-clusters have been established, they can be passed to the μC-maintenance phase (line 5) that is explained in detail further. After the maintenance phase, the micro-clusters can be clustered to the final clusters (line 6), we calculate the mapping function from input nodes to clusters (line 7) and the summary nodes (line 8). At the end of every timestamp the algorithm outputs the summary graph G' and the mapping function Φ (line 9).

Algorithm 2. μC

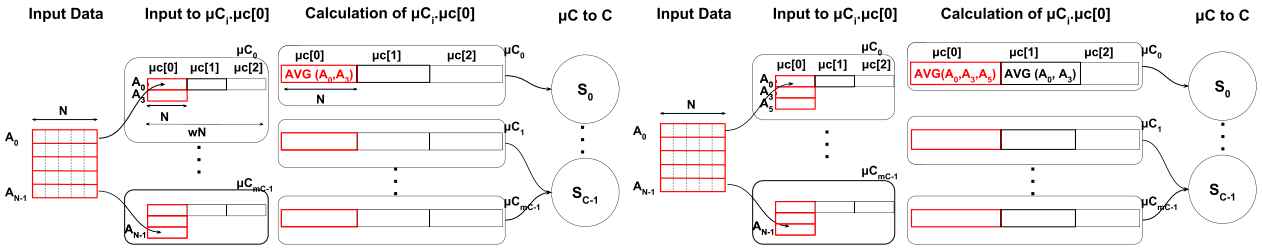
input: Graph $G^t(V,E)$ as $A_{G^t} \in [0,1]^{NN}$, q, k, w
output: Summary graph $G'(S, S \times S)$ as $A'_G \in [0,1]^{kk}$, function $\Phi : V \rightarrow S$

1 $t \leftarrow 0$
2 **while** true **do**
3 $A \leftarrow$ Read input graph A_{G^t}
4 $\mu C \leftarrow \mu C$-kmeans(A) `// Algorithm 3`
5 $\mu C \leftarrow \mu$C-maintenance(μC) `// Algorithm 4`
6 $C \leftarrow C$-kmeans(μC)
7 $\Phi \leftarrow$ Calculate mapping from nodes to supernodes
8 $A_{G'} \leftarrow$ Calculate summary from C `// Equations (1) & (2)`
9 **report** $(A_{G'}, \Phi)$
10 $t \leftarrow t+1$

From input to micro-clusters. At each timestamp, N new vectors arrive and get absorbed by the micro-clusters. Algorithm 3 describes how the input is added to the micro-clusters. First, μC finds the closest micro-cluster to the current input vector v^*, i.e., $\mu C^* = \min_i \text{dist}(\mu C_i, v^*)$, where dist is the cosine distance between two vectors, and μC_i is represented by its centroid (lines 3 to 3). The micro-cluster updates the values of the centroids and calculates the distance from their previous value ($shift$), selecting the maximum shift and updating the values of the seeds (lines 3 to 3). If the maximum shift exceeds a predefined threshold ($cutoff$), the process continues until either the centroids do not change more than this threshold or the number of the iterations exceeds a predefined value of $iterations$ (line 3). When the algorithm converges (line 3), the micro-cluster absorbs the vector and updates its statistics (lines 3 to 3). The statistics include the update of the $IDList$ and its bitmap array that represents the existence of a node in the micro-cluster. Additionally, updates the values of $F[0]$, the standard deviation of the absorbed points and calculates the centroid of the micro-cluster.

Algorithm 3 starts by selecting the seeds of the clusters and dropping the oldest statistics in order to keep the most recent ones. In the online phase of the algorithm, the seeds of k-means are selected to be the values of the centroids of the micro-clusters computed in the previous timestamp (line 2). This way, the algorithm can converge faster given that the edges between the nodes do not change significantly. Additionally, we shift all the bitmaps of the $IDList$ by one position so that the least significant bit (lsb) is free to be updated by the new arrivals. We also remove the least recent value of F, we set $SD = 0$, and we shift the centroid μc of the micro-cluster to liberate the position for the new centroid (lines 3 to 3). Fig. 4 shows the clustering process from the input data to the micro-clusters, and the computation of the centroids for two consecutive timestamps.

Micro-Cluster Maintenance Phase. If the newly absorbed vectors cause the micro-cluster to shift its centroid beyond a *maximum boundary*, then the micro-cluster is split. We define



(a) Timestamp t **(b)** Timestamp $t+1$

Fig. 4. Example of μC algorithm for two consecutive timestamps. In both figures, the input data that are clustered at each timestamp are those in red. In black are the data that have been clustered in previous timestamps. After the input data are clustered to the micro-clusters, the statistical information is updated and the micro-clusters pass on the maintenance phase. Finally, the micro-clusters are clustered to the supernodes.

the *maximum boundary* of a micro-cluster as the standard deviation of the distances of the vectors that belong to the micro-cluster from its centroid. Additionally, if a micro-cluster has absorbed fewer vectors than a threshold, then it is merged. If a micro-cluster needs to be absorbed, a new micro-cluster should be split, in order to keep the total number of micro-clusters q unaltered. Algorithm 4 describes the maintenance phase of the μC algorithm. The input of the maintenance algorithm are the micro-clusters μC, the input matrix A, the split threshold, and the merge threshold. The micro-clusters with $F[0]$ less than a threshold form the $ListMerge$ (line 4) whereas the ones with SD larger than a threshold form the $ListSplit$ list. The next step is to rank the $ListSplit$ (line 4) by non-increasing SD and select only the top $|ListMerge|$ elements to form the H list (line 4), which contains all the micro-clusters that needs to be split. In this way we ensure that we merge the same number of micro-clusters as we split, so that the total number of micro-clusters remains q. In the merge phase of the algorithm (lines 4 to 4), all micro-clusters that exist in the $ListMerge$ are merged with the micro-cluster with the closest centroid. From this process we get $|ListMerge|$ empty micro-clusters that are used in the split phase. Finally, in the split phase of the algorithm (lines 4 to 4) each micro-cluster of the H list is partitioned in two. One part of it remains in the original micro-cluster and the other part is assigned to the micro-cluster that remained empty from the merge phase of the algorithm (lines 4 to 4).

From Micro-Clusters to Supernodes. The next step is to assign the micro-clusters to the supernodes. μC does so by using the k-means algorithm. The micro-clusters are considered as weighted pseudo-points. The value of the pseudo-point is the centroid of the micro-cluster, and the weight is the F value (i.e., the number of vectors) stored in each micro-cluster. The output of this step is a mapping from micro-clusters to supernodes that represents the summary.

To complete the construction of the summary, we need to assign each vector in the micro-cluster within the window (which represents one node in the input tensor) to a supernode. The supernode merges all the $IDLists$ of the micro-clusters in it. Recall that the $IDList$ of each micro-cluster contains the information of which vector is included in the specific micro-cluster. Finally, each input node is assigned to the supernode that contained it the most during the current window, i.e., the assignment from node to supernode is decided by majority voting. The mapping from vectors to supernodes is described by the following formula:

$$\Phi(i) = \arg\max_j \sum_t BitMap_{i,j}[t],$$

where $BitMap_{i,j}$ is the $BitMap$ of node i in supernode j.

Algorithm 3. μC-kmeans

input: A, μC, iterations, cutoff
output: μC
1 **foreach** $\mu C_i \in \mu C$ **do**
2 $\mu C_i.seed \leftarrow \mu C_i.\mu c[0]$
3 Shift all BitMaps of $\mu C_i.IDList$
4 Shift $\mu C_i.F$ $\mu C_i.SD \leftarrow 0$
5 Shift $\mu C_i.\mu c$
6 $rounds \leftarrow 0$
7 $mapper \leftarrow \{\}$
8 **while** $shift > cutoff$ **and** $rounds < iterations$ **do**
9 **foreach** $A_i \in A$ **do**
10 $Index \leftarrow 0$
11 $min\_dist \leftarrow cos\_dist(\mu C_0.seed, A_i)$
12 **foreach** $j \in [1, \mu - 1]$ **do**
13 $dist \leftarrow cos\_dist(\mu C_j.seed, A_i)$
14 **if** $distance < min\_dist$ **then**
15 $Index \leftarrow j$
16 $min\_dist \leftarrow distance$
17 μC_{Index} absorbs vector A_i
18 $mapper[i] \leftarrow (Index, min\_dist)$
19 $max\_shift \leftarrow 0$
20 **foreach** $\mu C_i \in \mu C$ **do**
21 $\mu C_i.centroid[0] \leftarrow$ Update with average of the absorbed points
22 shift $\leftarrow$ cos_dist$(\mu C_i.seed, \mu C_i.centroid[0])$
23 $max\_shift \leftarrow max(shift, max\_shift)$
24 $\mu C_i.seed \leftarrow \mu C_i.centroid[0]$
25 **if** $max\_shift \leq cutoff$ **or** $rounds \geq iterations$ **then**
26 **foreach** $key \in mapper$ **do**
27 $Index \leftarrow mapper[key[0]]$
28 $min\_dist \leftarrow mapper[key[1]]$
29 $\mu C_{Index}.IDList.append(key)$
30 $\mu C_{Index}.SD += min\_dist^2$
31 $\mu C_{Index}.F[0] \leftarrow \mu C_{Index}.F[0] + 1$
32 $\mu C_{Index}.\mu c[0] \leftarrow$ Calculate the average of the points that belong to μC_{Index}
33 **else**
34 $round \leftarrow round + 1$
35 **return** μC

Computational Complexity. Let q be the total number of micro-clusters, then the cost of clustering N vectors is $\mathcal{O}(qN^2)$. To remove the oldest F_i of all the micro-clusters,

we need q operations, and to update the bitmaps of all micro-clusters we need a maximum of Nw operations. As a result, μC needs $\mathcal{O}(qN^2 + Nw + q)$ operations for maintaining the existing micro-clusters. The time complexity for clustering the micro-clusters to the supernodes is $\mathcal{O}(kqN)$.

Each micro-cluster keeps an (Nw)-dimensional vector as its centroid, and two w-dimensional vectors for the frequencies and the standard deviation. Additionally, the $IDList$ of all q micro-clusters has a maximum of $\mathcal{O}(wN)$ tuples. Considering q micro-clusters, the overall space requirement of the algorithm is $\mathcal{O}(qwN)$.

Algorithm 4. μC Maintenance

input: μC, adjacency matrix A,
split threshold θ_1, merge threshold θ_2
output: Updated μC
Initialization:
1 $ListMerge \leftarrow \{\mu C_i \mid F_i[0] < \theta_1\}$
// List of μCs to be merged when number of vectors are less than θ_1
2 $ListSplit \leftarrow \{\mu C_i \mid SD_i > \theta_2\}$
// Candidates of μCs to be split when SD is beyond the threshold θ_2
3 $ListSplit \leftarrow$ Rank $ListSplit$ by non-increasing SD
4 $H \leftarrow$ take top $|ListMerge|$ micro-clusters
// List of μCs to be split of size $|ListMerge|$
Merge phase:
5 **foreach** $\mu c_i \in ListMerge$ **do**
6 Find μc_j closest to μc_i
7 $\mu C_j \leftarrow$ Merge($\mu C_j, \mu C_i$)
8 Update statistics of μC_j
Split phase:
9 **foreach** $\mu C_i \in H$ **do**
10 $\mu C_{empty} \leftarrow$ Pop the first empty micro-cluster of the $ListMerge$
11 Assign seeds to μC_{empty} and μC_i from $\mu C_i.IDList$ randomly
K-means algorithm:
12 **while** *not converge* **do**
13 **foreach** $id \in \mu C_i.IDList$ **do**
14 Assign A_{id} to the closest micro-cluster between μC_i and μC_{empty}
15 Update statistics for μC_{empty} and μC_i
16 **return** μC

5 Distributed Implementation

As described in the previous section, both kC and μC have a computational complexity which might become prohibitive on large scale graphs and for large window sizes. Our solution to this problem is to distribute the computation on a cluster of machines (CM).

The core of both algorithms is the online k-means algorithm which requires, at each timestamp, to compute the distances between all the input vectors and the centroids of all (micro-)clusters. Each vector is assigned to the closest (micro-)cluster, and the new centroids are computed as the average of the vectors in each cluster. The algorithm is repeated until it converges, or until it reaches the maximum number of iterations.

Conceptually, the algorithm is composed by three parts: (i) assignment of the vectors to the clusters, (ii) computation of the new centroids of the clusters, and (iii) computation of the distance between the old and the new centroids. In the first part, the assignment of each vector to the clusters is completely independent from each other, i.e., the computation is completely parallel, provided that the centroids of the clusters are available to all the processes. Therefore, we parallelize by partitioning the input vectors across the CM. The second part of the algorithm requires all the results from the first phase to proceed. This part is implemented by exchanging messages between parallel processes. The third and last part of the algorithm is fairly inexpensive and can be executed locally.

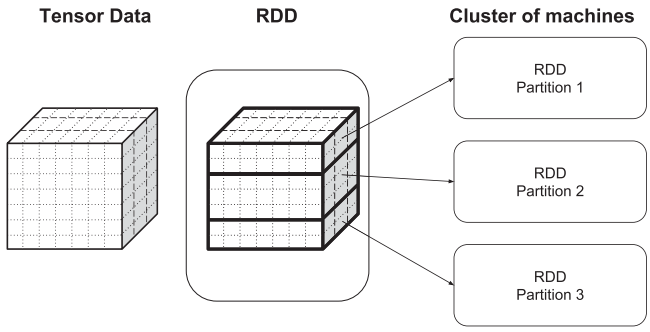


Fig. 5. The tensor data are partitioned horizontally to create the RDD and each partition of it is processed by one executor processes.

For the implementation of the distributed algorithm we use the Apache Spark framework.[1] The architecture of Spark has a master process which is connected to several executor processes. These executors can be distributed over the CM. The main (driver) program runs in the master, except for the parts of the algorithm that are explicitly distributed to the executors. Once the executors finish their distributed computation, the results are sent back to the master, which resumes the execution of the serial parts of the algorithm.

The basic abstraction in Spark is the *Resilient Distributed Dataset (RDD)* that supports two types of operations: *transformations* and *actions*. Transformations create a new RDD, based on the existing one, whereas actions evaluate a function on the RDD, and return the result to the master program. In our implementation, the first RDD is created by the vectors to be clustered. On this dataset we apply a transformation via the *map()* and the *reduceByKey()* functions to compute the distance of all vectors from the centroids, to assign the vectors to the clusters and to sum the values of the points. Then we use actions, *countByKey()*, *reduceByKeyLocally()* and *collect()*, to evaluate the results of the transformation, and return to the master the number of vectors of each cluster and the sum of the values of the vector of each cluster, which are combined to compute the centroid of each cluster. Both transformations and actions are handled by the Spark environment and therefore the algorithm does not interfere with the exchange of messages between the executors. The final part of the algorithm is executed locally in the master, by keeping the previous centroids in memory.

Fig. 5 shows the way that the tensor is partitioned across the CM. Since we cut the data horizontally, the clustering of the vectors can be done independently at each executor.

1. http://spark.apache.org

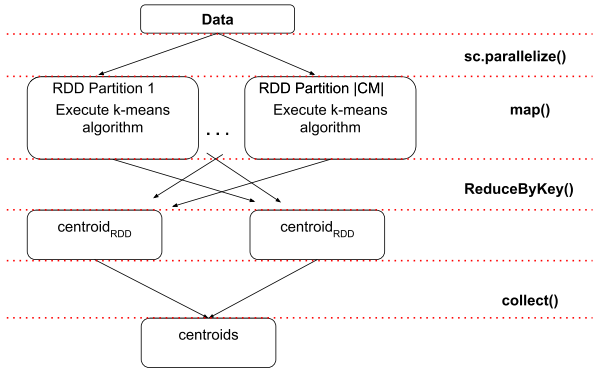


Fig. 6. High-level overview of the distributed implementation of the kC algorithm using *parallelize()*, *map()*, *reduceByKey()*, and *collect()* functions.

Fig. 6 shows the high level overview of one round of the process. The process starts by parallelizing the vectors to be clustered across the CM (*sc.parallelize()* function). Each executor maintains a copy of the centroids of the clusters and a partition of the input data that needs to be clustered. When the clustering of all the vectors in the partitions is finished (*map()* function) each vector has been assigned the cluster id which it belongs to. The next step is to calculate the new values of the centroids of each cluster. This is done with the function *reduceByKey()*. Finally, the new values of the centroids return to the master process (*collect()* function) in order to calculate the shift of the centroids of the previous round. The process is repeated until the algorithm converges.

Algorithm 5. Distributed kC

input: Graph $G^t(V, E)$ as $A_{G^t} \in [0,1]^{NN}$, number of supernodes k, length of window w, *cutoff*
output: Summary graph $G'(S, S \times S)$ as $A'_G \in [0,1]^{kk}$, function $\Phi : V \rightarrow S$

1 $t \leftarrow 0$
2 $\mathcal{A}^{W_0} \leftarrow$ Initialize the adjacency tensor window with 0
3 **while** true **do**
4 $A \leftarrow$ Read input graph A_{G^t}
5 $\mathcal{A}^{W_t} \leftarrow$ Slide window and update with A
6 **foreach** $i \in N$ **do**
7 points$_{RDD} \leftarrow$ sc.parallelize(($\mathcal{A}^{W_t}[i].coords$, $\mathcal{A}^{W_t}[i].id$)
8 centroids$_{old} \leftarrow$ random($\mathcal{A}^{W_t}, k$)
9 **while** *biggest_shift* $> cutoff$ **do**
10 points$_{assign} \leftarrow$ points$_{RDD}$.map(lambda x: kmeans(x,centroids$_{old}$))
11 centroid$_{RDD} \leftarrow$ points$_{assign}$.reduceByKey()
12 centroids $\leftarrow$ centroid$_{RDD}$.collect()
13 point.population $\leftarrow$ points$_{assign}$.countByKey()
14 biggest_shift $\leftarrow$ diff(centroids, centroids$_{old}$)
15 centroids$_{old} \leftarrow$ centroids
16 $C \leftarrow$ Cluster values from centroids
17 $\Phi \leftarrow$ Mapping function from nodes to supernodes
18 $G'^{W_t} \leftarrow$ Calculate summary from C `// Equations (1) & (2)`
19 **report** (G'^{W_t}, Φ)
20 $t \leftarrow t+1$

Algorithm 5 describes the distributed implementation of Algorithm 1. The tensor data is used to create the RDD (lines 6 & 7) that is distributed to the CM. The tensor is cut horizontally and distributed to the machines as shown in Fig. 5. Therefore, each machine is responsible for clustering $\frac{N}{|CM|}$ points with the distributed k-means (lines 9 to 15). Lines 12 and 13 are responsible for evaluating the RDDs and return the values to the main program. After the distributed k-means converges, the algorithm continues by updating the values of the clusters (line 16), calculating the summary and the mapping function (lines 17 & 18) and finally reports the summary for each timestamp (line 19). Fig. 6 also shows the Spark functions that were used to implement the distributed k-means (lines 6 to 15).

Algorithm 6. Distributed μC

input: Graph $G^t(V, E)$ as $A_{G^t} \in [0,1]^{NN}$, number of micro-clusters q, number of supernodes k, length of window w, *cutoff*
output: Summary graph $G'(S, S \times S)$ as $A'_G \in [0,1]^{kk}$, function $\Phi : V \rightarrow S$

1 $t \leftarrow 0$
2 **while** *true* **do**
3 $A \leftarrow$ Read input graph A_{G^t}
4 points$_{RDD} \leftarrow$ sc.parallelize(($A[i].coords$, $A[i].id$) for i in range(N))
5 μC-$centr_{old}$ = random(A, q)
6 **while** *biggest_shift* $> cutoff$ **do**
7 points$_{assign}$ = points$_{RDD}$.map(lambda x: μC-kmeans(x, μC-$centr_{old}$))
8 μC-$centr \leftarrow$ points$_{assign}$.reduceByKeyLocally()
9 point.population $\leftarrow$ points$_{assign}$.countByKey()
biggest_shift $\leftarrow$ diff(μC-$centr$, μC- $centr_{old}$)
10 μC-$centr_{old} \leftarrow \mu C$-$centr$;
11 $\mu C \leftarrow$ update μC values from the centroids
12 $\mu C \leftarrow \mu C$-maintenance(μC) `// Algorithm 4`
13 μC-points$_{RDD} \leftarrow$ sc.parallelize(($\mu C[i].coords$, $\mu C[i].F$) for i in range(q))
14 $centr_{old} \leftarrow$ random($\mu C.centroid, k$)
15 **while** *biggest_shift* $> cutoff$ **do**
16 μC-points$_{assign} \leftarrow \mu C$-points$_{RDD}$.map(lambda x: C-kmeans(x, $centr_{old}$)
17 $centr \leftarrow \mu C$-points$_{assign}$.reduceByKeyLocally()
18 biggest_shift $\leftarrow$ diff($centr, centr_{old}$)
19 $centr_{old} \leftarrow centr$
20 $population \leftarrow$ For each cluster sum $\mu C.F$ vectors
21 $C \leftarrow$ update C from the $centr$ and the $population$
22 $\Phi \leftarrow$ Calculate mapping from nodes to supernodes
23 $G' \leftarrow$ Calculate summary from C `// Equations (1) & (2)`
24 **report** (G', Φ)
25 $t \leftarrow t+1$

Algorithm 6 describes the distributed version of Algorithm 2. The algorithm can be divided in three parts. The first part, lines 4 to 11, describes the distributed version of clustering the input points to the micro-clusters. Line 12 refers to the maintenance Algorithm 4 and finally, lines 13 to 21 describe the distributed version of clustering the micro-clusters to the supernodes. At the end of each timestamp, the algorithm reports the mapping function and the summary graph (lines 22 & 23). In this algorithm we create two RDDs, lines 4 and 13. In line 4 the RDD is created from the input points that are clustered to the micro-clusters using a modified k-means (lines 6 to 10). In line 13 the RDD is created by the micro-clusters, that are finally clustered to the supernodes (lines 15 to 20).

TABLE 1
Dataset Names, Number of Nodes N, Number of Edges M, and Density ρ

Graph	N	M	ρ
Synth2kSparse	2005	2522874	0.08
Synth2kDense		4257061	0.10
Synth4kSparse	4023	10646970	0.08
Synth4kDense		16537369	0.10
Synth6kSparse	6015	23505535	0.08
Synth6kDense		37415417	0.10
Synth8kSparse	8243	43979220	0.08
Synth8kDense		68386928	0.10
Twitter7k	7493	15698940	0.03
Twitter9k	9683	19380438	0.02
Twitter13k	13755	24981361	0.01
Twitter24k	24650	36015735	0.007
NetFlow	250021	7882015	1.576E-5

6 Experimental Evaluation

6.1 Datasets and Experimental Setup

For our experiments we use a dataset extracted from Twitter hashtag co-occurrences, Yahoo! Network Flows Data,[2] and a synthetic dataset. Based on them we create 13 different datasets of various sizes and densities for 16 consecutive timestamps, which are summarized in Table 1.

Twitter Hashtag Co-Occurrences. We collected all hashtag co-occurrences for December 2014 from Twitter that included only Latin characters and numbers. Each hashtag represents a node of the graph and the co-occurrence with another hashtag denotes an edge of the graph. A large fraction of the hashtags appears in the dataset only few times during the entire month, making it extremely sparse. Therefore, we introduce a minimum threshold of appearances of the hashtags during the entire month. By changing the value of the threshold (20000, 15000, 10000, 5000) we obtain four different datasets with varying sizes and densities: Twitter7K, Twitter9K, Twitter13k, and Twitter24k, respectively (Table 1). We collected data for 16 days and separate it according to the day of publication in 16 consecutive timestamps. The edges of the graph are weighted and represent the number of times that two hashtags co-occurred in a day, normalized by the maximal number of co-occurrences between any two hashtags each day.

Yahoo! Network Flows Data. Provided by Yahoo Webscope for Graph and Social Data, this dataset contains communication patterns between end-users. The nodes of the graph are the IP-addresses of the users and the weights on the edges are the normalized value of the sum of bytes that have been exchanged between the nodes. The data are separated in files of 15-minute intervals. For our experiments we use the first 16 files from 8:00 to 11:30 of the 29th of April of 2008, to create our 16 consecutive timestamps. In our dataset we include only IP-addresses that appear at least 100 times.

Synthetic Data. To evaluate the scalability of our methods, we create a synthetic data-generator that can produce data with varying size, structure, and density. The synthetic dataset is a 3-order tensor $T \in [0,1]^{NNw}$, where N corresponds to the number of nodes of the dynamic graph and w is the total number of timestamps that we produce. To simulate the dynamic graph we need to take into account that each node can have frequent, rare or no communication with the rest of the nodes of the graph. The weights of the edges of the nodes with high communication will be higher than for those with rare communication. For the nodes with no communication, the edge does not exist and in the tensor the weight will be zero. To simulate this behavior, we create clusters in the tensor T as shown in Fig. 7. The values of the intra-cluster edges (areas of the tensor that are highlighted in colors in Fig. 7) are high and represent the nodes with the frequent communication. The rest of the values, the inter-cluster edges (white areas in Fig. 7), have lower or zero value. Our synthetic data generator takes as input the approximate number of nodes n (approximate size of the dataset), the number of timestamps t, the number of clusters C that exist in the tensor T, and the sparsity of the dataset $sparsity$ (Algorithm 7). The number of nodes that exist at each cluster C is given by a random number between the values $\frac{n-0.4n}{C}$ and $\frac{n+0.4n}{C}$ (Algorithm 7 lines 1, 2). Consequently, the sum of the nodes that exist in all clusters will approximate the input value N (Algorithm 7 line 3). The next step is to create the tensor T and initialize it with zeros (line 4).

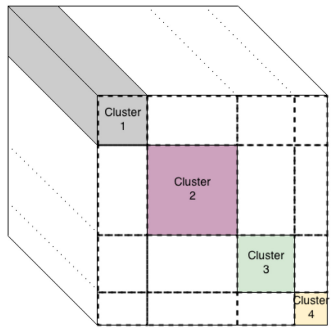


Fig. 7. Synthetic Dataset represented as a tensor. We create clusters in the tensor to simulate frequent communication patterns for nodes inside the same cluster and rare or no-communication patterns between nodes of different clusters.

Lines 5 to 10 of Algorithm 7 describe the computation of the weights of the intra-cluster edges. For each one of the clusters of the tensor we choose a random value between 0.4 and 0.8 to assign to the centroid of the cluster (line 6). At each timestamp the centroid of the cluster moves to some direction by Δ, and consequently the values of the edges change as well, so that we produce the dynamic communication patterns on the resulting graph. The Δ value is multiplied by $(-1)^j$, where j is the number of the timestamp, to avoid the movement of the centroids to only one direction (line 8). To determine the weights of the intra-edges we add to the value of the centroid of the cluster a random Gaussian noise with mean 0.01 and a small deviation (line 9). Therefore, the values of the intra-edges are similar to each other.

Finally, we take care of the inter-cluster communication of the nodes (lines 11 to 16). For each cluster we choose with how many of the rest of the clusters will communicate. This number is the outcome of a function that returns integers between 0 and $\frac{C}{sparsity}$ (line 12), where $sparsity$ is the value

2. https://webscope.sandbox.yahoo.com/catalog.php?datatype=g

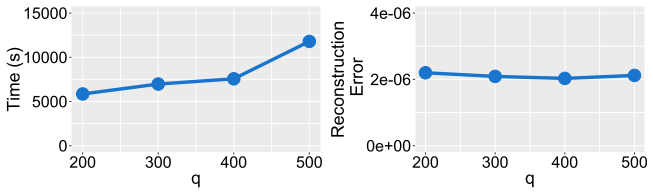


Fig. 8. Number of μC versus Time (left plot) and Reconstruction Error (right plot).

that can be tuned to create datasets with different densities. The weights of the inter-edges get a non-negative random Gaussian value with mean 0.001 (line 15) and small standard deviation. Therefore, the inter-edges have zero or very low value weights.

Algorithm 7. Synthetic Data Generator

input: approximate number of nodes n, number of clusters C, number of timestamps w, sparsity of the dataset $sparsity$
output: $Tensor$

1 **foreach** $i \in range(C)$ **do**
2 $N_i \leftarrow \text{random}(\frac{n-0.4n}{C}, \frac{n+0.4n}{C})$
3 $N \leftarrow \sum N_i$
4 $Tensor \leftarrow$ initialize with 0
5 **foreach** $i \in range(C)$ **do**
// Puts weights to the intra-cluster edges
6 $C_i \leftarrow random.uniform(0.4, 0.8)$
7 **foreach** $j \in range(t)$ **do**
8 $C_i \leftarrow C_i + (-1)^j \Delta$
9 $edges_{C_i} \leftarrow create\_intra\_edges(N_i, C_i, j, 0.01)$
10 $Tensor \leftarrow Tensor[j].\text{update}(edges_{C_i})$
11 **foreach** $i \in range(C)$ **do**
// Puts weigh to the inter-cluster edges
12 $connections_{C_i} \leftarrow random.int(0, \frac{C}{sparsity})$
// Number of connections of each cluster with the rest
13 $map[i] \leftarrow random(range(C), connections_{C_i})$
// Dictionary with connections between clusters
14 **foreach** $j \in range(t)$ **do**
15 $edges_{C_i} \leftarrow inter\_edges(map[i], 0.001)$
16 $Tensor \leftarrow Tensor.\text{update}(edges_{C_i})$
17 **return** $Tensor$

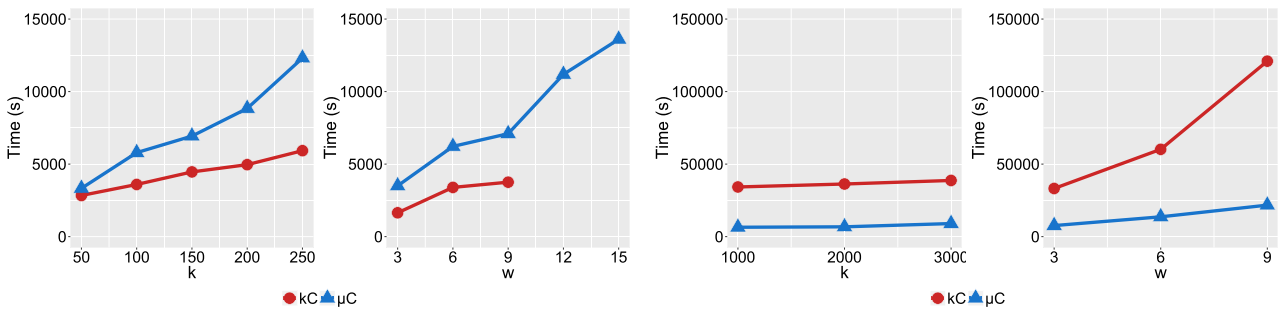
For our experiments we produce eight different datasets. For all the datasets we set $C = 500$ and $t = 16$. We produce datasets of four different sizes by setting the parameter N to $2005, 4023, 6015$, and 8243. Additionally, for each N we produce a sparse and a dense dataset. The characteristics of these datasets are also presented in Table 1.

Experimental Setup. We run our experiments on a cluster of 400 cores. These cores are contained in a total of 720 cores that are distributed uniformly across 30 machines, each one having 24 cores Intel(R) Xeon(R) CPU E5-2430 @ 2.20 GHz. Each worker node allocates 12 GB of memory and each executor on the worker node uses 3 GB of memory.

6.2 Efficiency

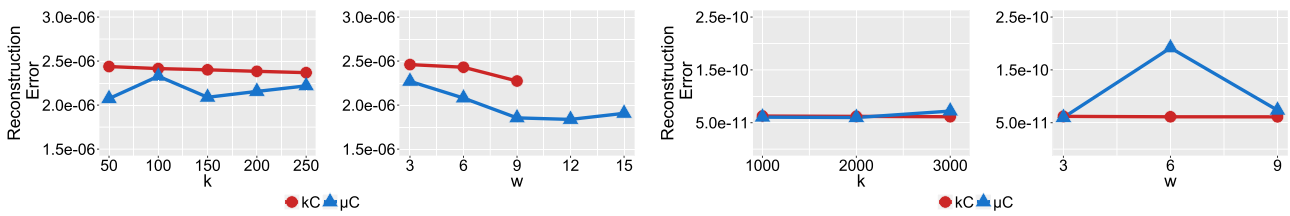
We use kC and μC to summarize Twitter13k and NetFlow datasets and we report the execution time as we increase the number of supernodes of the summaries, the length of the tensor window, and the number of the micro-clusters (for μC). We begin with the μC method and how the number of micro-clusters affect the efficiency of the algorithm. In the left plot of Fig. 8 we report the execution time results for different number of micro-clusters when we set the number of supernodes to 150 and the tensor window to 9. We see that the execution time increases with the number of micro-clusters. The maintenance phase (Algorithm 4) is non linear in the number of micro-clusters due to the overhead added from the Spark operations when serializing the results of the clustering of the vectors to the micro-clusters. This can be seen from the curve in the left plot of Fig. 8. For the rest of the experiments we decided to fix the number of micro-clusters, doubling the number of supernodes.

Fig. 9a shows the results for the Twitter13k dataset as we increase the number of supernodes from 50 to 250 (left plot) and the size of the window from 3 to 15 (right plot). The plot on the left uses window size 9 and the results of the execution time refer to the timestamp 8 which is the first one where the entire window is full of adjacency matrices (timestamp 0 is the first timestamp of the algorithm that contains one non-zero adjacency matrix). Our kC algorithm is always faster than μC and almost linear with respect to the number of supernodes, whereas the execution time of μC increases much faster. This is due to the two level clustering of the μC and more specifically because the number of micro-clusters is much bigger than the number of the final clusters. However, the big advantage of our μC is shown on the right plot of Fig. 9a where we compare the two methods while we increase the size of the window. Although kC is faster than μC, we see that it fails to execute



(a) Twitter13k **(b)** NetFlow

Fig. 9. Efficiency results for Twitter13k and NetFlow datasets. The left plots of (a) and (b) show the execution time for different number of clusters. The right plots (a) and (b) show the execution time for different window sizes.



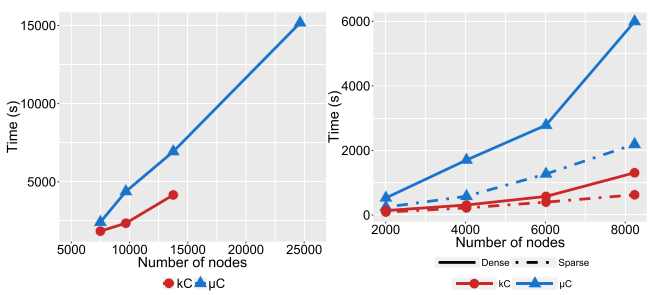
(a) Twitter13k **(b)** NetFlow

Fig. 10. Reconstruction error results for Twitter13k and NetFlow datasets. The left plots of (a) and (b) show the reconstruction error for different number of clusters. The right plots of (a) and (b) show the reconstruction error for different window sizes.

for large windows (greater than 9) due to the linearly-increasing memory requirements. This shows the advantage of μC, which can produce results even when the size of the window increases to 15, since its memory requirements increase sub-linearly. Fig. 9b shows the results for NetFlow data. In this case μC is always faster than the kC algorithm due to the much larger fraction of N/q than in the Twitter13k. Therefore, the overhead of μC due to the intermediate step of micro-clustering is not noticeable whereas the overhead from the increasing the number of nodes reduces the efficiency of the kC algorithm.

6.3 Reconstruction Error

We compute the reconstruction error, which represents the sum of the differences of the weights of the edges between the original graph and what can be reconstructed from the summary graph, according to Eq. (3). Figs. 10a and 10b show the results of the reconstruction error for Twitter13k and NetFlow datasets while we increase the number of supernodes (left plot) and the size of the window (right plot). In both Figs. 10a and 10b the reconstruction error of the kC method is decreasing while we increase the number of supernodes and the size of the window. In contrast, the reconstruction error of μC does not always decrease when we increase the number of clusters or the size of the window. This is due to the micro-cluster structure, which allows the input nodes to enter different micro-clusters at each timestamp and therefore spikes on the behavior of the communication patterns of the input data are reflected on the summary. These spikes can be noticed in the left plot of Fig. 10a and the right plot of Fig. 10b. On the other hand, kC allows spikes of input data to be smoothed during the window and are not noticed in the reconstruction error. The reconstruction error of μC is always smaller for Twitter13k, whereas this is visible in NetFlow for small k and w. Finally, the reconstruction error decreases as we increase the number of micro-clusters while keeping fixed the number of supernodes (right plot of Fig. 8).



(a) Twitter data **(b)** Synthetic data

Fig. 11. Scalability: (a) Execution time results for different sizes of Twitter data (Twitter7k, Twitter9k, Twitter13k, Twitter24k). (b) Execution time for the synthetic datasets.

6.4 Scalability

The last set of quantitative experiments present the scalability of both algorithms for different number of nodes and for different graph densities. For these experiments we use the different versions of Twitter and synthetic datasets. Fig. 11a shows that the kC method is always faster than μC, but fails for the Twitter24k dataset due to its high memory requirements. However, we cannot give definitive trends on the scalability of the two algorithms since the different versions of the Twitter datasets have different densities. Fig. 11b shows the execution time using synthetic datasets of two different densities and four different graph sizes. In both, sparse and dense datasets, kC is always faster than μC. Moreover, the difference in execution time between the two methods in the dense datasets is much larger than in the sparse ones.

6.5 Queries

We now test our methods on approximately answering interesting queries by using the generated summaries. While our framework is general in nature, here we focus on a specific class of queries that considers the tensor as a *probabilistic* data structure. Moreover, we focus on queries that have a time component which can be expressed as a sliding window operator.

A probabilistic (or uncertain) graph $G = (V, E, p)$ is an undirected graph associated with a function p: $E \rightarrow [0, 1]$ associating each edge e with a probability $p(e)$ that the edge exists in the graph. We examine three problems in this setting: *edge density*, *node degree*, and *number of triangles* on a time window $W = [1, w]$.

Edge Density. Given two subsets of vertices $\mathcal{S}_1 \subseteq V$ and $\mathcal{S}_2 \subseteq V$, where $|\mathcal{S}_1 \cap \mathcal{S}_2| = 0$, the *expected edge density* $\mathbb{E}[E_{\mathcal{S}_1,\mathcal{S}_2}]$ between $\mathcal{S}_1$ and $\mathcal{S}_2$ is defined as the normalized sum of the probabilities between the two subsets

$$\mathbb{E}[E_{S_1,S_2}] = \frac{\mathbb{E}[|\{(u,v) \in E : u \in S_1, v \in S_2\}|]}{|S_1||S_2|}.$$

Clearly, if $\mathcal{S}_1$ and $\mathcal{S}_2$ are singletons (i.e, $\mathcal{S}_1 = \{u\}$ and $\mathcal{S}_2 = \{v\}$), then $\mathbb{E}[E_{\mathcal{S}_1,\mathcal{S}_2}]$ reduces to the edge probability $p(u, v)$.

This quantity can be easily generalized to our setting, i.e., a tensor $\mathcal{A}_G^W$, by considering the expectations over the window $W = [1, w]$:

TABLE 2
Edge Density Query for Different Sizes of S for $k = 150$, $w = 9$, and $t = 9$ fort Twitter13k

Method	$\|\mathcal{S}\|$	min	max	mean	median	σ	time
kC	20	$4e-5$	3182	13.8	4	46	0.05
μC	20	$4e-5$	2643	9.5	2.9	28.5	0.05
kC	200	$1e-6$	10.3	0.8	0.6	0.7	0.05
μC	200	$3e-6$	5.8	0.4	0.3	0.4	0.05
kC	2000	$1e-4$	0.7	0.1	0.1	0.1	0.06
μC	2000	$1e-4$	0.7	0.2	0.2	0.1	0.06

We present statistics of the relative results ($\frac{|\mathbb{E}_W[E'_{\mathcal{S}_1,\mathcal{S}_2}]-\mathbb{E}_W[E_{\mathcal{S}_1,\mathcal{S}_2}]|}{\mathbb{E}_W[E_{\mathcal{S}_1,\mathcal{S}_2}]}$) and the relative average execution time when we execute the same query 100,000 times.

$$\mathop{\mathbb{E}}_{W}[E_{\mathcal{S}_1,\mathcal{S}_2}] = \frac{\sum_{t=1}^{W} \mathbb{E}[|\{(u,v) \in E_t : u \in \mathcal{S}_1, v \in \mathcal{S}_2\}|]}{w|\mathcal{S}_1||\mathcal{S}_2|}.$$

which is equal to

$$\mathop{\mathbb{E}}_{W}[E_{\mathcal{S}_1,\mathcal{S}_2}] = \frac{\sum_{t=1}^{w} \sum_{u\in\mathcal{S}_1,v\in\mathcal{S}_2} A_{G_t}(u,v)}{w|\mathcal{S}_1||\mathcal{S}_2|}.$$

The same query in the summary graph is defined as

$$\mathop{\mathbb{E}}_{W}[E'_{\mathcal{S}_1,\mathcal{S}_2}] = \frac{\sum_{u\in\mathcal{S}_1,v\in\mathcal{S}_2} A_{G'_t}(\Phi(u),\Phi(v))}{|\mathcal{S}_1||\mathcal{S}_2|},$$

where $A_{G'_t}$ is the adjacency matrix of the summary of the tensor window $\mathcal{A}_G^W$, and Φ is the mapping function from nodes to supernodes.

Node Degree. Given a subset of vertices $\mathcal{S} \subseteq V$ the *expected node degree* is defined as $\mathbb{E}[D_{\mathcal{S}}]$ of the nodes of $\mathcal{S}$:

$$\mathbb{E}[D_{\mathcal{S}}] = \mathbb{E}[|\{(u,v) : u \in S, v \in V\}|],$$

which in the case of the window $W = [1, w]$ over a tensor $\mathcal{A}_G^W$ is defined as

$$\mathop{\mathbb{E}}_{W}[D_{\mathcal{S}}] = \frac{\sum_{t=1}^{W} \mathbb{E}[|\{(u,v) : u \in S, v \in V\}|]}{w},$$

which is equal to

$$\mathop{\mathbb{E}}_{W}[D_{\mathcal{S}}] = \frac{\sum_{t=1}^{w} \sum_{u\in\mathcal{S},v\in V} A_{G_t}(u,v)}{w}.$$

The same query is defined for the summary graph as:

$$\mathop{\mathbb{E}}_{W}[D'_{\mathcal{S}}] = \sum_{u\in\mathcal{S},v\in V} A_{G'_t}(\Phi(u),\Phi(v)).$$

Probabilistic Triangles. Given a tensor $\mathcal{A}_G^W$ over a time window W we define a triangle a triplet of vertices $\{u, v, z\} \in V$ iff each of the three edges $e_1 = (u, v), e_2 = (v, z), e_3 = (u, z)$ exists in at least one timestamp of W. The probability that the edge e_1 exists in at least a timestamp of W is

$$P(e_1, W) = 1 - \prod_{t\in W}(1 - A_{G_t}(e_1)).$$

Then, we can define the probability of the triplet of vertices $\{u, v, z\}$ being a triangle as

TABLE 3
Node Degree Query for Different Sizes of S for $k = 150$, $w = 9$, and $t = 9$ for Twitter13k

Method	$\|\mathcal{S}\|$	min	max	mean	median	σ	time
kC	10	$1e-5$	92.5	1.5	0.8	2.1	0.50
μC	10	$3e-6$	33.4	0.9	0.5	1.3	0.50
kC	$e2$	$1e-5$	5.2	0.5	0.4	0.4	0.17
μC	$e2$	$6e-6$	2.4	0.3	0.3	0.2	0.17
kC	$e3$	$6e-7$	1.12	0.1	0.1	0.1	0.50
μC	$e3$	$9e-6$	0.6	0.2	0.2	0.1	0.50

We present statistics of the relative results ($\frac{|\mathbb{E}_W[D'_{\mathcal{S}}]-\mathbb{E}_W[D_{\mathcal{S}}]|}{\mathbb{E}_W[D_{\mathcal{S}}]}$) and the relative average execution time when we execute the same query 100,000 times.

$$P_{triangle}(u,v,z) = \prod_{i=1}^{3} P(e_i, W),$$

and therefore the expected number of triangles in the tensor $\mathcal{A}_G^W$ is

$$\sum_{u,v,z\in A_{G_t}} P_{triangles}(u,v,z).$$

Results. We execute the queries *edge density* and *node degree* for three different sizes of samples $\mathcal{S}$ of the Twitter13k dataset. For each size of sample we execute the same query 100000 times, each one using a different random sample of nodes. For both methods we present the relative values of the query with respect to the values of the query on the original graph, i.e., $\frac{|\mathbb{E}_W[E'_{\mathcal{S}_1,\mathcal{S}_2}]-\mathbb{E}_W[E_{\mathcal{S}_1,\mathcal{S}_2}]|}{\mathbb{E}_W[E_{\mathcal{S}_1,\mathcal{S}_2}]}$ and $\frac{|\mathbb{E}_W[D'_{\mathcal{S}}]-\mathbb{E}_W[D_{\mathcal{S}}]|}{\mathbb{E}_W[D_{\mathcal{S}}]}$. From the 100000 results for each query and sample size, we report the minimum, maximum, mean, median and the standard deviation of the relative results for both methods. The last column presents the relative average execution time, i.e., the query time execution in the summary graph divided by the one in the original graph.

Table 2 presents the relative results for the query *edge density*. We see that as the sample size increases the fraction of the median value decreases significantly. For sample size $S = 20$ the relative error between kC and the query on the original graph is 4 whereas for μC is 2.9. However, for sample size $S = 200$ and $S = 2000$ the relative error decreases to 0.6 and 0.1 for kC and to 0.3 and 0.2 for μC. The relative average execution time is between 0.05 and 0.06, which means that the query in the summary graphs runs 20 times faster than the queries in the original graph for both methods. The average execution time for the sample size S={20, 200, 2000} are {0.02, 0.2, 23} seconds for the original graph and for both methods kC and μC are {0.0001, 0.01, 1.5} seconds, respectively.

Table 3 presents the results for the query *node degree*. As the set size $S = \{10, 100, 1000\}$ increases, the fraction of the median value decreases from 0.8 to 0.1 for kC and from 0.5 to 0.2 for μC. The relative execution time, as described above, is between 0.5 and 0.17 which means that the query on the summary graph for both kC and μC runs 2 to 6 times faster than on the original graph. The average execution times on the original graph for $S = \{10, 100, 1000\}$ are {0.0009, 0.02, 0.08} seconds while in the summary graphs for both methods are {0.0005, 0.005, 0.05} seconds.

TABLE 4
Results for the Probabilistic Triangles Query for $k = 500$, $w = 9$, and $t = 9$

Graph	Query Result			Computation time (sec)			Relative error	
	Original	kC	μC	Original	kC	μC	$\frac{\lvert orig.-kC\rvert}{orig.}$	$\frac{\lvert orig.-\mu C\rvert}{orig.}$
Synth2kSparse	17843.99	19373.60	14380.57	30878	174	181	0.08	0.19
Synth4kSparse	185890.24	204059.95	214144.52	257124	175	171	0.09	0.15
Synth6kSparse	634455.22	705767.37	755808.25	900939	171	170	0.11	0.19

We present the query result and the computational time on the original graph and on the summaries for kC and μC algorithms. Finally, we present the relative error of kC and μC with respect to the original graph query result.

Table 4 shows summarized results for the query *probabilistic triangles*. We report the results of the queries in the original graph and in the two summaries that are produced by kC and μC. We also report the computation time for the calculation of the query and the relative error between the original and the summary result. For our experiments we use the entire graph so that we do not alter any of the properties of the graphs which are important for the calculation of the triangles. Due to the computational complexity of the query on the original graph, we could not calculate it on the real-world data, but only for the smallest of the synthetic datasets. The execution time is two orders of magnitude faster when computing the query on the summary graphs and the relative error of the queries remain small (between 0.15 and 0.19).

7 Conclusions and Future Work

This work introduces the problem of dynamic graph summarization via tensor streaming and propose two distributed scalable algorithms. Our baseline algorithm kC based on clustering is fast, but rather memory expensive. Our μC method reduces the memory requirements by introducing an intermediate step that keeps statistics of the clustering of the previous rounds, while paying a small cost in terms of execution time.

Extensive experiments on several real-world and synthetic graphs show that our techniques scale to graphs with millions of edges and that they produce good quality summaries with small reconstruction error. We further evaluate our methods by approximately answering probabilistic temporal queries with good accuracy in small computational time. As future work we consider extending our current setting to dynamic graphs where also new nodes are inserted into the existing structure.

References

[1] B. Adhikari, Y. Zhang, S. E. Amiri, A. Bharadwaj, and B. A. Prakash, "Propagation based temporal network summarization," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 4, pp. 729–742, Apr. 2018.

[2] M. Adler and M. Mitzenmacher, "Towards compressing web graphs," in *Proc. Data Compression Conf.*, 2001, Art. no. 203.

[3] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A framework for clustering evolving data streams," in *Proc. 29th Int. Conf. Very Large Data Bases*, 2003, pp. 81–92.

[4] C. C. Aggarwal, A. Hinneburg, and D. A. Keim, "On the surprising behavior of distance metrics in high dimensional spaces," in *Proc. 8th Int. Conf. Database Theory*, 2001, pp. 420–434.

[5] P. Boldi and S. Vigna, "The webgraph framework I: Compression techniques," in *Proc. 13th Int. World Wide Web Conf.*, 2004, pp. 595–602.

[6] W. Fan, J. Li, X. Wang, and Y. Wu, "Query preserving graph compression," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2012, pp. 157–168.

[7] C. Hernández and G. Navarro, "Compression of web and social graphs supporting neighbor and community queries," in *Proc. 6th ACM Workshop Social Netw. Mining Anal*, 2011.

[8] A. Khan and C. C. Aggarwal, "Query-friendly compression of graph streams," in *Proc. IEEE/ACM Int. Conf. Advances Social Netw. Anal. Mining*, 2016, pp. 130–137.

[9] K. LeFevre and E. Terzi, "GraSS: Graph structure summarization," in *Proc. SIAM Int. Conf. Data Mining, SDM*, Columbus, Ohio, USA, 2010, pp. 454–465, doi: 10.1137/1.9781611972801.40.

[10] X. Liu, Y. Tian, Q. He, W.-C. Lee, and J. McPherson, "Distributed graph summarization," in *Proc. 23rd ACM Int. Conf. Conf. Inf. Knowl. Manag.*, 2014, pp. 799–808.

[11] Z. Liu, J. X. Yu, and H. Cheng, "Approximate homogeneous graph summarization," *J. Inf. Proc.*, vol. 20, pp. 77–88, 2012.

[12] H. Maserrat and J. Pei, "Neighbor query friendly compression of social networks," in *Proc. 16th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2010, pp. 533–542.

[13] S. Navlakha, R. Rastogi, and N. Shrivastava, "Graph summarization with bounded error," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2008, pp. 419–432.

[14] Q. Qu, S. Liu, F. Zhu, and C. S. Jensen, "Efficient online summarization of large-scale dynamic networks," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 12, pp. 3231–3245, Dec. 2016.

[15] M. Riondato, D. García-Soriano, and F. Bonchi, "Graph summarization with quality guarantees," in *Proc. IEEE Int. Conf. Data Mining*, 2014, pp. 947–952.

[16] N. Shah, D. Koutra, T. Zou, B. Gallagher, and C. Faloutsos, "Timecrunch: Interpretable dynamic graph summarization," in *Proc. 21th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2015, pp. 1055–1064.

[17] T. Suel and J. Yuan, "Compressing the graph structure of the web," in *Proc. IEEE Data Compression Conf.*, 2001, pp. 213–222.

[18] J. Sun, D. Tao, and C. Faloutsos, "Beyond streams and graphs: dynamic tensor analysis," in *Proc. 12th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2006, pp. 374–383.

[19] N. Tang, Q. Chen, and P. Mitra, "Graph stream summarization: From big bang to big crunch," in *Proc. Int. Conf. Manag. Data*, 2016, pp. 1481–1496.

[20] Y. Tian, R. A. Hankins, and J. M. Patel, "Efficient aggregation for graph summarization," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2008, pp. 567–580.

[21] H. Toivonen, F. Zhou, A. Hartikainen, and A. Hinkka, "Compression of weighted graphs," in *Proc. 17th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2011, pp. 965–973.

[22] I. Tsalouchidou, G. De Francisci Morales, F. Bonchi, and R. A. Baeza-Yates, "Scalable dynamic graph summarization," in *Proc. IEEE Int. Conf. Big Data*, 2016, pp. 1032–1039.

[23] N. Zhang, Y. Tian, and J. M. Patel, "Discovery-driven graph summarization," in *Proc. 26th Int. Conf. Data Eng.*, 2010, pp. 880–891.

[24] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: An efficient data clustering method for very large databases," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 1996, pp. 103–114.

Ioanna Tsalouchidou received the PhD degree from the Department of Information and Communication Technologies, University Pompeu Fabra, in 2018. Her research focuses on data streams and data mining, with an emphasis on large-scale and dynamic graphs.

Francesco Bonchi is research leader with the ISI Foundation, Turin, Italy, where he leads the Algorithmic Data Analytics Group. He is also a (part-time) principal scientist for data mining with Eurecat Barcelona. He was director of research with Yahoo Labs in Barcelona, Spain, leading the Web Mining Research Group. He is the general chair of IEEE Data Science and Advanced Analytics 2018. He was PC chair of IEEE ICDM16 and ACM HT17 and is a member of the ECML PKDD Steering Committee, and an associate editor of many journals in the data management and mining area (the *IEEE Transactions on Big Data*, the *IEEE Transactions on Knowledge and Data Engineering*, the *ACM Transactions on Knowledge Discovery from Data*, the *ACM Intelligent Systems and Technology*, and the *Data Mining and Knowledge Discovery*). More information about him can be found at http://www.francescobonchi.com/.

Gianmarco De Francisci Morales is a research leader with the ISI Foundation. In the past, he worked as a scientisti with QCRI, visiting scientist with Aalto University, research scientist with Yahoo Labs, and research associate with the HPC Lab of ISTI-CNR. He is an active member of the Apache Software Foundation working on the Hadoop ecosystem (Giraph, S4), and a committer for the Apache Pig project. He co-organizes the Workshop Series on Social News on the Web (SNOW) co-located with the WWW conference. He is a lead developer of SAMOA, an open-source platform for mining big data streams. More information about him can be found at http://gdfm.me.

Ricardo Baeza-Yates is CTO of NTENT, a semantic search technology company based in California. He is also the director of CS Programs with Northeastern University, Silicon Valley campus. He is also a part-time professor with Universitat Pompeu Fabra in Barcelona and the Universidad de Chile in Santiago. Before that, he was VP of research with Yahoo Labs, based in Barcelona, Spain, and later in Sunnyvale, California, from January 2006 to February 2016. In 2009, he was named an ACM fellow and in 2011 IEEE fellow, among other awards and distinctions. His areas of expertise are web search and data mining, information retrieval, data science, and algorithms. More information about him can be found at http://www.baeza.cl/