Scalable Dynamic Graph Summarization

Ioanna Tsalouchidou
Web Research Group, DTIC
Pompeu Fabra University, Spain
ioanna.tsalouchidou@upf.edu

Gianmarco De Francisci Morales
Qatar Computing Research Institute
gdfm@acm.org

Francesco Bonchi
Algorithmic Data Analytics Lab
ISI Foundation, Turin, Italy
francesco.bonchi@isi.it

Ricardo Baeza-Yates
Web Research Group, DTIC
Pompeu Fabra University, Spain
rbaeza@acm.org

***Abstract*—Large-scale dynamic 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, maintaining the evolution of their structure and the communication patterns. 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, distributed, and tunable algorithms for summarizing this type of graphs. We apply our methods to several real-world and synthetic dynamic graphs, and we show that they scale well on the number of nodes and produce high-quality summaries.**

I. Introduction

In a variety of application domains (e.g., social networks, molecular biology, communication networks, etc.), the data of interest is routinely represented as very large graphs 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 become computationally challenging tasks. 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 dynamic nature of most of the graphs of interest, such as social networks, communication networks, or the WWW.

Graph summarization speeds up the analysis by creating a *lossy concise representation of the graph* that fits into main memory. Answers to otherwise expensive queries can then be computed using the summary without accessing the exact representation on disk. Query answers computed on the summary incur in a minimal loss of accuracy. When multiple graph analysis tasks can be 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 [5], to create easily interpretable visualizations of the graph [9], or to store a compressed representation of the graph [10].

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 prefixed size. 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. Therefore the information of interest for the summarization is a 3-order tensor of dimension $N \times N \times w$ where N is the number of nodes and w is the prefixed length.

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 social and communication 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, an edge (i, j) which has always been associated to a value of 0 up to timestamp t, when it takes a value > 0, is an edge that *appears* for the first time at t. Similarly an edge that starts having 0 weight after t can be considered to *disappear* after t.

In this paper we introduce a new version of the *dynamic graph summarization* problem, by generalizing the definition by LeFevre and Terzi [5] (discussed next) to the dynamic graph setting in a streaming context.

A. 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, as we extend existing definitions for static graph summarization and we adopt methods coming from data stream clustering literature, in the following we cover these two areas of research.

Static graph summarization. LeFevre and Terzi [5] propose to use an enriched "supergraph" as a summary, associating an integer to each supernode (a set of vertices) and to each superedge (an edge between two supernodes), representing respectively the number of edges (in the original graph) between vertices in the supernode and between the two sets of vertices connected by the superedge, respectively. 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 properties queries, as the expectation of the answer over the set of possible worlds. Their method follows a greedy heuristic resembling an agglomerative hierarchical clustering with no quality guarantee.

Riondato et al. [10] build on the work of LeFevre and Terzi [5] 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.

Navlakha et al. [9] 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. [6] follow the definition of Navlakha et al. [9] and present the first distributed algorithm for summarizing large-scale graphs. A different approach followed by Tian et al. [12] and Liu et al. [7], for graphs with labeled vertices, is to create "homogeneous" supernodes, i.e., to partition vertices so that vertices in the same set have, as much as possible, the same attribute values.

Shah et al. [11] approach the problem of graph summarization as a compression problem, and further extend it to 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 but fast answers to queries about the original graph.

Toivonen et al. [13] propose an approach for graph summarization tailored to weighted graphs, which creates a summary that preserves the distances between vertices. Fan et al. [3] present two different summaries, one for reachability queries and one for graph patterns. Hernández and Navarro [4] focus instead on neighbor and community queries, and Maserrat and Pei [8] 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. [1] study the problem of clustering evolving data streams over different time horizons. They view the data streams as evolving processes that are clustered differently during different time horizons, rather than entities that have to be clustered at one time. They use *microclusters* 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. [14] in their *BIRCH* method.

As mentioned before, Shah et al. [11] deal with the problem of lossless dynamic-graph compression. Instead, we tackle the problem of lossy summarization of dynamic graphs. Our algorithms are distributed by design with scalability as main goal. Differently from the work by Liu et al. [6], the task distribution of our algorithm does not create dependencies or requirements for message-passing supervision.

B. Contributions

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 [5].
- We design two online, distributed, and tunable algorithms for summarizing dynamic large-scale graphs. The first one is inspired by Riondato et al. [10] and it is based on clustering. The second one overcomes the main limitation of the first one (memory requirements) by using the *micro-clusters* concept from Aggarwal et al. [1], adapted to our graph-stream setting and achieving scalability without giving up quality.
- 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, as confirmed by our experiments on several synthetic and real-world dynamic graphs.

II. Problem Formulation

In this section we first define the problem of static graph summarization, then we extend it to dynamic graphs.

A. Static graph summarization

Given a weighted graph $G(V, E, \epsilon)$ with $V = \{V_1, \dots, V_N\}$, a *weight function* $\epsilon : E \to [0, 1]$, 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., $S = \{S_1, ..., S_k\}$, with $\bigcup_{i\in[1,k]} S_i = V$ and $S_i \cap S_j = \emptyset$ if $i \neq j$. The function $\sigma : S \times S \to [0, 1]$ maintains the average edge weight among the nodes contained in two supernodes, and is given by

$$\sigma(S_i, S_j) = \frac{\sum\limits_{k\in S_i, \ell\in S_j} \epsilon(k, \ell)}{|S_i||S_j|}, S_i \neq S_j$$

and

$$\sigma(S_i, S_i) = 2\frac{\sum\limits_{k\in S_i, \ell\in S_i} \epsilon(k, \ell)}{|S_i||S_i - 1|}, S_i = S_j \ .$$

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 [5], 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':

$$RE(A_G|A_{G'}) = \frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} |A_G(V_i, V_j) - A_{G'}(s(V_i), s(V_j))|$$

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

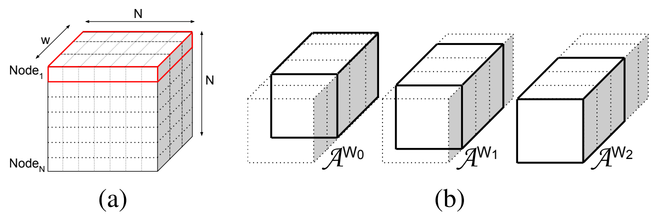


Fig. 1: (a) 3-order tensor of dimension $N \times N \times w$ where N is the number of nodes and w is the prefixed length; (b) sliding tensor-window.

Algorithm 1: kC

```
input : Graph $G^t(V, E)$ as $A_{G^t} \in [0,1]^{NN}$, number of
        supernodes $k$, length of window $w$
output: Summary graph $G'(S, S \times S)$ as $A'_G \in [0,1]^{kk}$,
        function $s : V \to S$
1  $t \leftarrow 0$
2  $\mathcal{A}^{W_0} \leftarrow$ Initialize the adjacency tensor window with zero
3  while true do
4      $A \leftarrow$ Read input graph $A_{G^t}$
5      $\mathcal{A}^{W_t} \leftarrow$ Slide window and update with $A$
6      $C \leftarrow k$-means($\mathcal{A}^{W_t}$)
7      $s \leftarrow$ Calculate mapping function from nodes to
       supernodes
8      $G'^{W_t} \leftarrow$ Calculate summary from $C$
           // Equations (1) & (2)
9      report $(G'^{W_t}, s)$
10     $t \leftarrow t + 1$
```

Riondato et al. [10] 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 list of each node: the clusters are then used as the supernodes.

B. Tensor summarization

We consider next a time series of w static graphs as described before. The time series of static graphs can be expressed as a time series of adjacency matrices $A_{G^t} \in [0,1]^{NN}$, where $t \in T$ or as a 3-order tensor $\mathcal{A}_G^W \in [0,1]^{NNw}$ as depicted in Figure 1(a). 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 $S = \{S_1, ..., S_k\}$ of V:

$$A_{G'}(S_i, S_j) = \frac{\sum\limits_{t=0}^{w} \sum_{k \in S_i, l \in S_j} \mathcal{A}_G^W(k,l,t)}{w|S_i||S_j|}, S_i \neq S_j \quad (1)$$

and

$$A_{G'}(S_i, S_j) = \frac{2\sum\limits_{t=0}^{w} \sum_{k \in S_i, l \in S_j} \mathcal{A}_G^W(k,l,t)}{w|S_i||S_j - 1|}, S_i = S_j. \quad (2)$$

The reconstruction error for tensor summarization is defined as follows:

$$RE(\mathcal{A}_G^W | A_{G'}) = \frac{\sum\limits_{t=0}^{w-1} \sum\limits_{i=0}^{N-1} \sum\limits_{j=0}^{N-1} |\mathcal{A}_G^W(V_i, V_j, t) - A_{G'}(s(V_i), s(V_j))|}{wN^2}. \quad (3)$$

C. Dynamic graph 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 $V = \{V_1, ..., V_N\}$, edges $E \subset V \times V$ and a function $f^t : E \times T \to [0,1]$ with $T = [0, t], t \in \mathbb{N}$. This can be represented as a time series of adjacency matrices where each adjacency matrix $A_G \in [0,1]^{NN}$. At each time stamp *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_i 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. Figure 1(b) shows the tensor window that indicates which of the timestamps are considered for the summarization, for three consequent timestamps.

At each time stamp t_i, we summarize the adjacency matrices that are included in the tensor window, i.e., the tensor $\mathcal{A}_G^{W_i} \in [0,1]^{NNw}$, where $W_i \in [t_i, t_i - w + 1]$. The tensor summary is defined as in Section II-B by minimizing the reconstruction error of Eq. (3). Finally, the values of the adjacency matrix $A_{G'W_i}$ are computed by Equations (1) and (2).

III. Algorithms

In this section we first describe our baseline clustering-based algorithm inspired by Riondato et al. [10], kC, which is effective but memory intensive, and then the more memory efficient and scalable μC, based on the use of micro-clusters.

A. Baseline algorithm: kC

Following Riondato et al. [10], 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. Figure 1(a) shows a tensor window of length w, and highlights one of the matrices ($Node_1$) that are the input for the clustering algorithm. After clustering the vectors, each cluster represents a super-node of the summary graph.

Algorithm 1 describes kC. For timestamp $t = 0$ were we initialize the tensor window (lines 1,2) and continue with the computation of the summary (lines 4-9). The rest of the algorithm (lines 3-10) describes the streaming behavior of the algorithm for the following timestamps (line 10).

Since the algorithm needs to work in a high-dimensional space, we prefer to use cosine distance rather than Euclidean

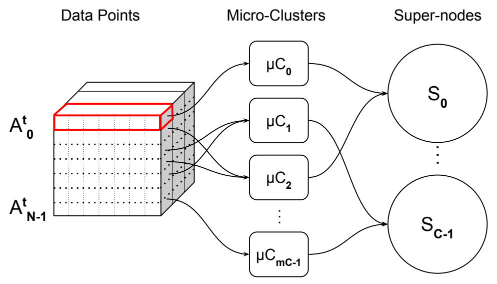


Fig. 2: Summarizing a tensor window by micro-clusters.

distance to measure the distance between two data points [2]. 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 Figure 1(b).

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 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 consideration.

B. Micro-clustering algorithm: μC

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

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

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 $s : V \to S$
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($\mu C$) // Algorithm 4
6      $C \leftarrow C$-kmeans($\mu C$)
7      $s \leftarrow$ Calculate mapping from nodes to supernodes
8      $G' \leftarrow$ Calculate summary from $C$ // Equations
           (1) & (2)
9      report ($G'$, $s$)
10     $t \leftarrow t + 1$
```

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 (i.e, the zero-th moment).
- 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 (i.e., the first moment).
- SD is a w-dimensional vector whose elements represent the standard deviation of the distances of all the vectors that are included in the micro-cluster from its centroid (i.e., the second moment), for each 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 3-10). The input of the algorithm is an adjacency matrix A_{G^t} that corresponds to the graph G^t of the current timestamp. Figure 2 shows the tensor window of length w and highlights one of the N vectors A^t_0 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 above. 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 will be explained in detail later. 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 (lines 7) and the summary nodes (lines 8,17). At the end of every timestamp the algorithm outputs the summary graph G' and the mapping function s (line 10).

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.*, $\mu C^* = \min_i \mathrm{dist}(\mu C_i, v^*)$, where dist is the cosine distance between two vectors, and μC_i is represented by its centroid (lines 6-14). The micro-cluster updates the values of the centroids and checks if their distance from their previous value exceeds a predefined threshold (lines 16-19). If this is the case, the process continues until either the centroids do not change more than this threshold or the number of the iterations exceeds a predefined value (line 3). Otherwise, the micro-cluster absorbs the vector (lines 21-28) and updates its statistics (lines 29 -32). 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 least recent 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). In 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 $IDList$ left by one so that the least significant bit (lsb) is free to be updated by the new arrivals. Additionally, we 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 (line 3).

Algorithm 3: μC-kmeans

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

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 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 less number of vectors than a threshold then it is merged. Algorithm 4 describes the maintenance phase of the μC algorithm. 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. The input of the maintenance algorithm (Algorithm 4) 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 $Merge$ list (line 1) whereas the ones with SD larger than a threshold form the $Split$ list. The next step is to rank the $Split$ list by increasing SD and select only the top $|Merge|$ elements (line 3) to form the H list. In this way we assure that we merge the same number of micro-clusters as we split, so that the total number of micro-clusters will remain q. The last step of the phase is to merge the micro-clusters that exist in the $Merge$ list with the closest micro-clusters (the distance between their centroids is minimum) (lines 4-7) and split the micro-clusters of the H list in two micro-clusters (lines 8-11). The algorithm returns the updated micro-clusters that will be used as an input for the following step (line 12).

Algorithm 4: μC maintenance

```
input  : $\mu C$, adjacency matrix A,
         split threshold $\theta_1$, merge threshold $\theta_2$
output: Updated $\mu C$
1  $Merge \leftarrow \{\mu C_i \mid F_i[0] < \theta_1\}$ // To be merged when F[0]< $\theta_1$
2  $Split \leftarrow \{\mu C_i \mid SD_i > \theta_2\}$ // Candidates to split when SD > $\theta_2$
3  $H \leftarrow$ Rank $Split$ by increasing size, take top $|Merge|$ micro-clusters
4  foreach $\mu c_i \in Merge$ do
5      Find $\mu c_j$ closest to $\mu c_i$
6      $\mu C_j \leftarrow$ Merge($\mu C_j, \mu C_i$)
7      Update statistics of $\mu C_j$
8  foreach $\mu C_i \in H$ do
9      $\mu C_{empty} \leftarrow$ Pop the first empty micro-cluster of the $Merge$ list
10     foreach $id \in \mu C_i.IDList$ do
11         Assign $A_id$ to the closest micro-cluster between $\mu C_i$ and $\mu C_{empty}$
12 return $\mu C$
```

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 graph.

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 super-node. The super-node merges all the $IDLists$ of the micro-clusters in it. Remember 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 super-node that contained it more time during the current window, *i.e.*, the assignment from node to super-node is decided by majority voting.

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 $\mathcal{O}(qwN)$.

TABLE I: Dataset names, number of nodes N, number of edges M, and density ρ.

Graph	N	M	ρ
Synth2kSparse	2005	2 522 874	0.08
Synth2kDense		4 257 061	0.10
Synth4kSparse	4023	10 646 970	0.08
Synth4kDense		16 537 369	0.10
Synth6kSparse	6015	23 505 535	0.08
Synth6kDense		37 415 417	0.10
Synth8kSparse	8243	43 979 220	0.08
Synth8kDense		68 386 928	0.10
Twitter7k	7493	15 698 940	0.03
Twitter9k	9683	19 380 438	0.02
Twitter13k	13 755	24 981 361	0.01
Twitter24k	24 650	36 015 735	0.007
NetFlow	250 021	7 882 015	1.576E-5

IV. Experimental Evaluation

A. Datasets and Experimental Setup

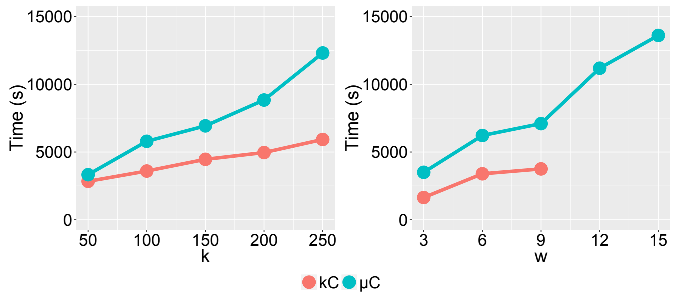
For our experiments we use the Twitter hashtag co-occurrences, Yahoo! Network Flows Data[1], and a synthetic dataset. Based on them we created 13 different datasets of various sizes and densities for 16 consecutive timestamps, which are summarized in Table I.

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 (20 000, 15 000, 10 000, 5000) we obtain four different datasets with varying sizes and densities: Twitter7K, Twitter9K, Twitter13k and Twitter24k, respectively (Table I). 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.

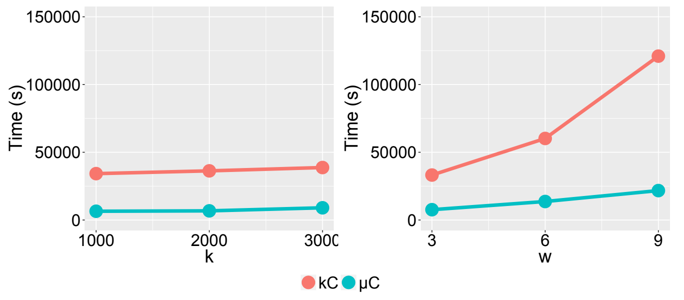
Yahoo! Network Flows Data. Provided by Yahoo Webscope for Graph and Social Data, this dataset contains communication patterns between end-users. The nodes of the graph are the IP-addresses of the users and the weights on the edges are the normalized value of the sum of octets that have been exchanged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

[1]https://webscope.sandbox.yahoo.com/catalog.php?datatype=g



(a) Twitter13k



(b) NetFlow

Fig. 3: 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.

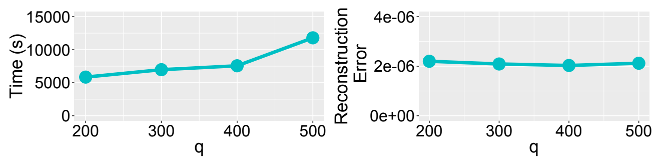


Fig. 4: Number of μC vs. Time and Reconstruction Error.

3-order tensor $T \in [0,1]^{NNw}$. The nodes of the tensor form clusters: the weights of intra-cluster edges are high (frequent communication between nodes of the same cluster), whereas the weights of inter-cluster edges are lower.

Our synthetic dataset generator takes as input, N (number of nodes), t (number of timestamps), and the number of clusters in the data. To determine the weights of the edges we choose the value of the centroid of each cluster and add random Gaussian noise with mean 0.01. 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. Finally, for each cluster we choose randomly the number of clusters that are connected, and we assign values randomly according to a Gaussian distribution with mean 0.001. For the same number of nodes, clusters, and timestamps, our algorithm is able to produce datasets of different densities. 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 I.

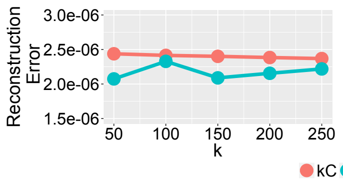
We run all the experiments on 400 cores distributed in 30 machines, each one having 24 cores Intel(R) Xeon(R) CPU E5-2430 0 @ 2.20GHz. We limited the amount of memory of the driver program to 12GB and of each executor process on the worker-nodes to 3GB.

B. Efficiency and Scalability

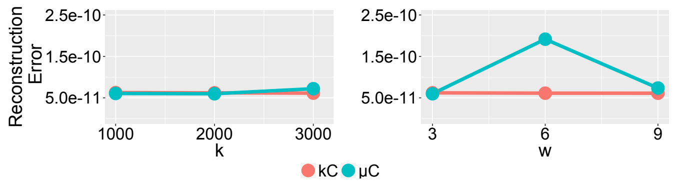
We use kC and μC to summarize Twitter13k and NetFlow datasets and we report the execution time as we increase the number of supernodes of the summaries, the length of the tensor window, and the number of the micro-clusters (for μC). We begin with the μC method and how the number of micro-clusters affect the efficiency of the algorithm. In the left plot of Figure 4 we report the execution time results for different number of micro-clusters when we set the number of supernodes equal to 150 and the tensor window equal to 9. We see that the execution time increases with the number of micro-clusters. From this plot we notice that after 400 micro-clusters the execution time increases faster. For the rest of the experiments we decided to keep the number of micro-clusters, doubling the number of supernodes.

Figure 3(a) 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. However, the big advantage of our μC is shown on the right plot of Figure 3(a) 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 for big window lengths (greater than 9) due to the linearly-increasing memory requirements. This shows the advantage of μC method, which can produce results even when the size of the window increases to 15, since it's memory requirements increase sub-linearly. Figure 3(b) shows the results for NetFlow data. In this case μC is always faster than the kC algorithm due to the much larger fraction of $N/micro-Clusters$ 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.

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. Figure 6(a) shows that the kC method is always faster than μC but fails for the Twitter24k dataset due to it's high memory requirements. However, we cannot give definitive trends on the scalability of the two algorithms since the different versions of the

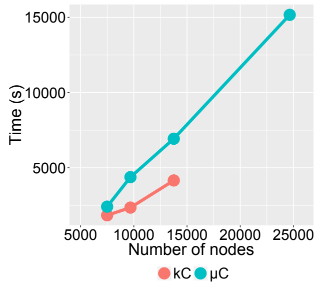


(a) Twitter13k

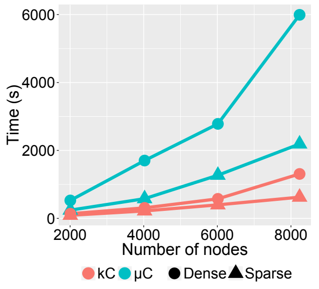


(b) NetFlow

Fig. 5: 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.



(a) Twitter data



(b) Synthetic data

Fig. 6: Scalability: (a) execution time results for different sizes of Twitter data (Twitter7k, Twitter9k, Twitter13k, Twitter24k); (b) execution time for the synthetic datasets.

Twitter datasets have different densities. Figure 6(b) 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 sets is much larger than in the sparse case.

C. *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 Equation 3. Figure 5(a) shows the results of the reconstruction error for Twitter13k dataset while we increase the number of supernodes (left plot) and the size of the window (right plot). In both plots the reconstruction error of the kC method is decreasing while we increase the number of supernodes and the size of the window. The reconstruction error of μC is always smaller but it is not always decreasing 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. On the other hand, kC allows spikes of input data to be smoothed during the window and not be noticed in the reconstruction error (right plot of Figure 5(b)). Finally, the reconstruction error decreases as we increase the number of micro-clusters while keeping fixed the number of supernodes (right plot of Figure 4).

V. Conclusions and Future Work

In this paper we introduce 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. 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. As future work we consider extending our current setting to dynamic graphs where also new nodes are inserted into the existing structure.

References

[1] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A framework for clustering evolving data streams. In *VLDB* 2003.

[2] C. C. Aggarwal, A. Hinneburg, and D. A. Keim. On the Surprising Behavior of Distance Metrics in High Dimensional Space. In *ICDT* 2001.

[3] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD* 2012.

[4] C. Hernández and G. Navarro. Compression of web and social graphs supporting neighbor and community queries. In *SNAKDD* 2011.

[5] K. LeFevre and E. Terzi. GraSS: Graph structure summarization. In *SDM* 2010.

[6] X. Liu, Y. Tian, Q. He, W.-C. Lee, and J. McPherson. Distributed graph summarization. In *CIKM* 2014.

[7] Z. Liu, J. X. Yu, and H. Cheng. Approximate homogeneous graph summarization. *J. Inf. Proc.*, 20(1):77–88, 2012.

[8] H. Maserrat and J. Pei. Neighbor query friendly compression of social networks. In *KDD* 2010.

[9] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *SIGMOD* 2008.

[10] M. Riondato, D. García-Soriano, and F. Bonchi. Graph summarization with quality guarantees. In *ICDM* 2014.

[11] N. Shah, D. Koutra, T. Zou, B. Gallagher, and C. Faloutsos. TimeCrunch: Interpretable Dynamic Graph Summarization. In *KDD* 2015.

[12] Y. Tian, R. A. Hankins, and J. M. Patel. Efficient aggregation for graph summarization. In *SIGMOD* 2008.

[13] H. Toivonen, F. Zhou, A. Hartikainen, and A. Hinkka. Compression of weighted graphs. In *KDD* 2011.

[14] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases, In *SIGMOD*, 1996.