2012 32nd IEEE International Conference on Distributed Computing Systems

Clustering Streaming Graphs

Ahmed Eldawy
University of Minnesota
eldawy@cs.umn.edu

Rohit Khandekar
IBM Research
rkhandekar@gmail.com

Kun-Lung Wu
IBM Research
klwu@us.ibm.com

***Abstract*—In this paper, we propose techniques for clustering large-scale "streaming" graphs where the updates to a graph are given in form of a stream of vertex or edge additions and deletions. Our algorithm handles such updates in an online and incremental manner and it can be easily parallelized.**

Several previous graph clustering algorithms fall short of handling massive and streaming graphs because they are centralized, they need to know the entire graph beforehand and are not incremental, or they incur an excessive computational overhead.

Our algorithm's fundamental building block is called *graph reservoir sampling*. We maintain a reservoir sample of the edges as the graph changes while satisfying certain desired properties like bounding number of clusters or cluster-sizes. We then declare connected components in the sampled subgraph as clusters of the original graph. Our experiments on real graphs show that our approach not only yields clusterings with very good quality, but also obtains orders of magnitude higher throughput, when compared to offline algorithms.

I. Introduction

Graphs are effective models for representing collections of relationships between entities. These relationships could be, for example, social ties between people, communications links between computer systems, links between web-pages, transportation channels between locations, or chemical bonds between atoms or molecules. These graphs are rich sources of information regarding how the entities behave and interact with each other. The field of *graph analytics* or *graph data mining* is devoted to understanding and decoding this information by studying structural properties of the graph and observing how they evolve with time. This information can then be put to use in several applications like online marketing, ranking search results, recommendation systems, churn prediction in mobile networks, disease control and drug discovery, to name a few.

In last decade, we have witnessed a significant growth in the capability of capturing and storing large volumes of real-time information. As a result, the interesting real-world graphs have grown in size to millions or even billions of vertices and edges. Furthermore these graphs may grow or change with time rapidly. For example, `Twitter`, with 200 million users as of 2011, generates over 200 million tweets and handles over 1.6 billion search queries per day [1]. Here each tweet or search query can be thought of as an edge or a collection of edges in an appropriate graph.

In this paper, we consider the problem of clustering or partitioning vertices in a "streaming" graph where the updates to the graph are given in form of a *stream* of edge or vertex additions or deletions. Handling such rapidly changing graphs is challenging because of its dynamic, online nature and massive scale. The algorithm needs to be necessarily incremental, extremely fast and amenable parallel or distributed implementations.

The graph clustering problem has been a subject of extensive research in the past [2], [3] mostly in an *offline* setting where the entire graph is given beforehand. Researchers have studied many models of clustering — overlapping [4] or non-overlapping [5]; hierarchical or multi-level [6], [7] clustering into a given number or unspecified number of vertex-clusters [8]; clustering based on vertex-proximity or cut-based measures such as clustering to minimize total cut-cost, sum/max conductance (or sparsity) [9], maximize cluster densities [10], correlation clustering to minimize classification errors [8]; clustering with constraints on the size of individual clusters [9], etc. Graph clustering has been used as a subroutine in applications like detecting communities in social networks for building recommender systems, market segmentation and product positioning; in computational biology to construct phylogenetic trees or analyze human genes; in computer science for load balancing in distributed computing, image segmentation, etc.

A. The problem description

We focus on the following variant of the graph clustering problem. Given an undirected graph $G = (V, E)$ and an integer B, partition the vertices V into subsets, called clusters, $C_1, C_2, \ldots, C_k$, such that each C_i has at most B vertices and the number of *inter-cluster* edges is minimized, i.e., minimize $|\text{cut}(\{C_1, \ldots, C_k\})|$ where $\text{cut}(\{C_1, \ldots, C_k\}) = \{e = (u, v) \in E \mid u \in C_i \text{ and } v \in C_j \text{ where } i \neq j\}$. We refer to this problem as *multi-way balanced graph partitioning problem*. In the offline version, it is assumed that the entire graph is known to the algorithm in the beginning.

Here we consider the so-called "streaming" version of this problem. In this online and dynamic setting, the graph G may change fairly rapidly with time due to additions and deletions of vertices and edges. Suppose time t increases from 0 to ∞. We use $G(t)$ to denote the version of graph G at time t. We assume that the parameter B is fixed. We are given a "stream of graph updates" ordered by increasing time-stamps. These updates could be (single or batch) addition or deletion of vertices or edges. We are also given a "stream of queries" ordered by increasing time-stamps. The algorithm is allowed to make only one pass over the streams and is required to maintain and update a clustering $\{C_1(t), \ldots, C_{k(t)}(t)\}$ of $G(t)$ incrementally. As the notation suggests, the number $k(t)$ of clusters may change with time. We sometimes drop t from

1063-6927/12 $26.00 © 2012 IEEE
DOI 10.1109/ICDCS.2012.20

the notation if it is clear from the context. The queries could be of the following types. Here the underlying graph and the clustering are corresponding to the time-stamp t of the query.

1) Given a vertex u, output i such that $u \in C_i$.
2) Given a vertex u, output all vertices in the cluster containing u.
3) Output the total number k of clusters.

Some desirable properties of the algorithm are as follows. The algorithm should be easy to parallelize for multi-core environments or be distributed for multi-host environments. In a distributed setting, for example, we may need a "front-end" host that receives the update and query streams and outputs the answer stream; and it distributes the actual computational and storage work to multiple "worker" hosts. To handle massive inputs, the algorithm should have low overall space complexity, preferably *linear* in the number of vertices and in general *sub-linear* in the number of edges or size of the entire graph. To handle high throughput streams, it should have very low time complexity per update or query.

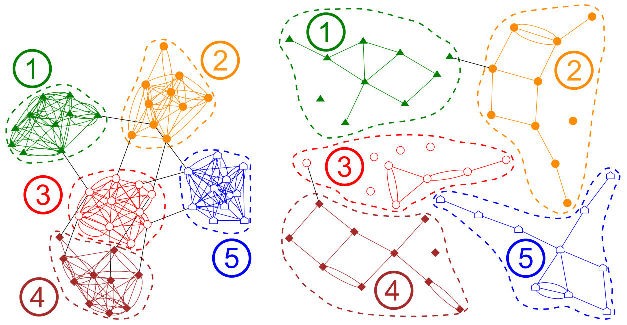
B. *Our contributions*

Our contributions can be summarized as follows.

1) We present a sampling-based algorithm that computes vertex-clusters in large-scale dynamic graphs in which edge insertions and deletions are allowed. In particular, it can be used for graphs induced by a sliding window of an edge-stream. It satisfies the constraints and attempts to optimize the objective in the multi-way balanced graph partitioning problem.
2) Our algorithm uses space that is in general *sub-linear* in the size of the entire graph, i.e., the number of edges.
3) It can be easily parallelized or distributed to exploit multi-core or multi-host environments.
4) Our experiments show that in terms of cut-quality, our algorithm performs almost as well as the well-established *offline* algorithm called METIS [7]. In terms of throughput (number of edges insertions or deletions handled per unit time), our algorithm out-performs METIS by about *three orders of magnitude*.

C. *Intuition behind our approach*

Suppose that the input graph clusters *well*, i.e., it has a hidden clustering $C_1^*, C_2^*, \ldots, C_{k^*}^*$ satisfying the cluster-size constraints such that each cluster C_i^* has a lot of internal edges and relatively fewer edges leaving it. How would one identify (an approximation of) such clusters using a simple heuristic? Use random sampling! That is, imagine that we construct a subgraph H of G by adding each edge to H independently with probability p, where $p \in (0, 1]$ is a parameter. Since each C_i^* has a lot of internal edges, vertices in C_i^* are likely to be in the same connected component in H. On the other hand, since there are relatively few inter-cluster edges in G, the vertices in C_i^* are less likely to be in the same connected component in C_j^* for $i \neq j$. Thus the connected components of H would be a good approximation of the hidden clustering. Thus our algorithm can declare the connected components of H as



(a) The original graph G (b) Random subgraph H with 20% of the edges

Fig. 1. Effect of sampling a well-clustered graph with preknown clusters. Vertices of each cluster are colored and grouped in each figure.

clusters in G. See Figure 1 for an illustration. In this figure, a graph was generated with five clusters each having ten vertices. Figure 1(a) shows the original graph G with each cluster grouped and colored with a different color. In figure 1(b), we show the subgraph H constructed randomly with $p = 0.2$. We notice that connected components in H represent clusters in G to some extent. For example, the connected component in the bottom right perfectly matches with cluster $\textcircled{5}$, while the connected component on the top left contains eight out of ten vertices from cluster $\textcircled{1}$. Although sampling was purely random, we can still recover a reasonable clustering of the original graph.

How does one extend this algorithm in presence of the constraint that size of any cluster is at most B? Here is a simple way. With each edge e, assign a random number $r_e \in (0, 1]$. Discard the edges with $r_e > p$ and order the remaining edges $\{e_1, \ldots, e_t\}$ such that $r_{e_1} \leq r_{e_2} \leq \ldots \leq r_{e_t} \leq p$. Now form a subgraph H by adding edges one-by-one in this order. While considering edge e_i, we determine if any connected component of $H \cup \{e_i\}$ has size more than B. If yes, we disregard e_i, otherwise we add e_i to H, i.e., let $H \leftarrow H \cup \{e_i\}$. After processing all the edges, declare the connected components of H as the clusters in G. We call the set of sampled edges as a *structural reservoir*. It is easy to see that an edge is "sampled" with probability *at most* p and that no cluster has size more than B. Picking r_e randomly from a uniform distribution assumes that edges are unweighted. It was shown that weighted edges can be easily supported by using an exponential distribution [11]. Note that the parameter p needs to be set carefully. If p is very small, one may encounter clusters with size significantly smaller than B, even of size 1; and obtain a large cut-size as a result. On the other hand, if p is very large, the size of the structural reservoir may be very large, thereby increasing the space complexity.

Now, how does one extend this algorithm in presence of additions and deletions of edges? Our extension to such an online setting is quite intuitive. Consider additions first. When a new edge e^* is added to G, we assign to it a random number $r_{e^*} \in (0, 1]$. We discard e^* right away if $r_{e^*} > p$. Otherwise, we insert e^* at the appropriate place in the sorted order, of all the edges with $r_e \leq p$, according to r_e values. It is clear that the inclusion of the edges in the structural reservoir, appearing before e^* in this order, will not be affected due to addition of e^*. We then start from e^* and decide which edges to include and which not to include as before. Note that, for additions to be effective, one has to keep track of the edges e with $r_e \leq p$ that are not included in the structural reservoir due to the cluster-size constraints. We call the set of such edges as a *support reservoir*. The edges may move back and forth between the structural reservoir and the support reservoir due to future additions. Now consider deletions. Suppose an edge e^* gets deleted from the graph. If e^* does not belong to either of the two reservoirs, there is nothing to do. If e^* belongs to the support reservoir, we simply delete it from there. If e^* belongs to the structural reservoir, we update the two reservoirs in a manner similar to one described above. This scheme can be slightly optimized by considering the edges in the reverse direction. Further details are given in Section III.

Since we do not store all the edges in the graph, it is not possible to compute the exact value of the objective function, i.e., the number of inter-cluster edges in the current clustering. To estimate the value of the computed clustering, we optionally maintain an independent reservoir in which each edge is added independently with probability $q \in (0, 1]$. It is easy to see that $\frac{1}{q}$ times the number of edges in the independent reservoir that go between the clusters is an unbiased estimator of the value of the objective function.

D. Organization

The rest of the paper is organized as follows. Section II discusses related works and compares them to our approach. In Section III, we present our core idea of structural sampling and how to use it for clustering. Section IV discusses experimental evaluation of our algorithm to support the claims related to cut-quality and throughput. In section V, we show how to generalize our approach for other problems in a streaming system. The paper ends with conclusions in Section VI.

II. Related Work

Traditional graph clustering algorithms fall short of handling massive and time-evolving graphs due to many reasons. Several algorithms are *offline*, i.e., they need the entire graph input before-hand. Offline algorithms cannot effectively be used in an online or streaming fashion, since they are not incremental in nature. Several algorithms are *centralized*; they cannot be or are very difficult to be parallelized or distributed. Several algorithms have super-linear, quadratic or even higher, computational overhead, probably because they are not designed to handle massive inputs.

The technique of random sampling to find small cuts in graphs is first analyzed by Karger [11]. He partitions vertices into (not necessarily balanced) k clusters in *offline* graphs to minimize the total number of inter-cluster edges. Zanghi et al. [12] present an online clustering of graphs using Erdös-Réyni model. In their model, vertices arrive online and get assigned to one of prespecified clusters so that a global likelihood function is maximized. After each arrival, previous vertices are allowed to move between clusters. They do not allow deletion or modification of vertices or edges. The algorithm also does not seem to scale well with the number of clusters. In fact, it is shown to work with only a relatively small number of clusters. Aggarwal et al. [13] considered a problem of finding clusters of structurally similar graphs in a stream of large number of small graphs.

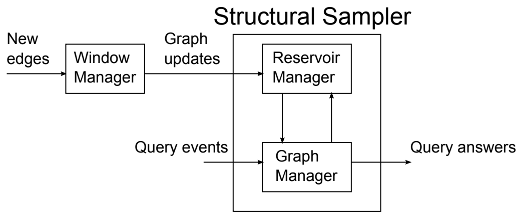


Fig. 2. Structural Sampler architecture

Recently, Aggarwal, Zhao, and Yu [14] (or AZY) use the idea of reservoir sampling with constraints. Our work can be thought of as being motivated by and an extension of AZY. We therefore compare our contributions to those of AZY in detail below. AZY is designed for detecting outliers in graph streams. Their scheme is based on finding multiple vertex clusterings in a streaming graph which only grows with edge additions. They do not consider edge deletions. Their approach maintains a random sample of the edges satisfying the same constraint that any connected component in the graph of sampled edges has at most B vertices. In order to do this, they assign a random number $r_e \in (0, 1]$ with each edge e and order the edges as $\{e_1, e_2, \ldots, e_m\}$ by non-decreasing r_e values. They then declare the largest *prefix* $\{e_1, \ldots, e_i\}$ that satisfies the connected component size constraints as their random sample. We call their scheme *prefix reservoir* sampling to reflect the fact that they do not allow samples that are non-prefix in this edge ordering. The prefix restriction allows them to update the reservoir on edge additions without the need of any support reservoir. Note however that the prefix reservoir sampling is likely to yield a clustering with many small clusters, resulting in worse objective value. In fact, in Section IV, we show that the objective value of their clustering is much worse than that of our clustering. Moreover, since they do not allow edge deletions, their scheme cannot be applied on fully dynamic graphs, e.g., graphs over sliding windows in a stream of edges.

III. Structural Sampler

In this section we describe how our scheme, called *"structural sampler"*, works for online clustering of streaming graphs. We first give a broad overview of the system, then get into the details of each component in the system.

A. System Architecture

Figure 2 shows the overall system architecture of *Structural Sampler*. The *window manager* is used only when one wants to maintain a graph over certain window of time. In a typical system, new edges arrive continuously on the *new edges* stream, e.g., new tweets or network packets. The window manager accepts this stream as input and produces *graph updates* (insert edge(s) or delete edge(s)) according to the specified window settings. If we directly receive such graph updates from a source, the window manager can be eliminated. The graph updates are passed to the reservoir manager which maintains a running sample of the "active" edges in the structural reservoir as well as some edges in a support reservoir. The sampled edges chosen by the *reservoir manager* are then passed to the *graph manager* which performs all graph related operations and keeps track of current clusters. *Query events* are sent directly to the graph manager which produces *query answers* as a stream. In our discussion, we will go quickly over window manager and graph manager as they are not our main contributions in this work. Our main contribution is the reservoir manager and we will discuss it in more details.

B. Window Manager

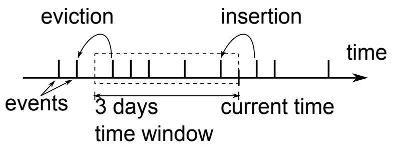


Fig. 3. A sample sliding window of three days

A *window manager* is a typical component in most streaming applications and it allows processing events according a specific window configuration. It is a preprocessing phase to determine edge insertions and deletions from a stream of edges. Figure 3 shows an example of a *'time-based sliding window of three days'*. This window keeps track of all events within the last three days. *'Sliding'* means the the window period of three days is always counted from current timestamp, i.e., it slides with time. Events that fall outside the window need to be evicted or deleted from the underlying system as the time advances. Another window configuration could be *'count-based tumbling window of* $10k$ *items'*. In this configuration new edges are added to the window without doing any processing. When the window fills up (i.e., number of items reaches $10k$), items within the current window are all processed and then the window is cleared which means all items are removed from the window, i.e., the window *tumbles*. Based on window configuration, window manager keeps the appropriate data structures to identify which edges to be evicted and when to evict them.

C. Reservoir Manager

Reservoir manager is the system component that receives edges insertion and eviction from window manager, and based on current state of the graph, it decides which edges to add to current sampled graph and which edges to remove from it. The main goal is to randomly sample a maximal number of edges while keeping size of largest connected component at most a threshold B (cluster bound). As explained earlier, connected

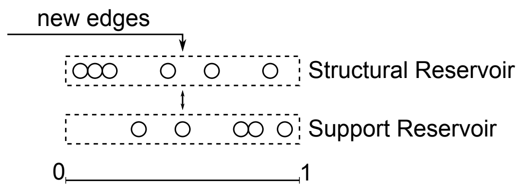


Fig. 4. Reservoir manager overview

components in this sampled graph is expected to represent good clusters of the original graph. We need to maximize the size of this sample to be a good representative to the real graph. At the same time, clustering this sample should be much easier than clustering the whole graph because each connected component is treated as one cluster.

Reservoir manager works as follows. Recall the intuition given in Section I-C and assume that $p = 1$ for now. For simplicity, let us assume we have received a set of edges and at some point of time we need to pick a random sample as described above. First, we randomly permute the list of edges. Then, we iterate over all these edges in this *random* order and start picking them up one by one and adding them to a sampled graph. We keep picking edges as long as we satisfy the clustering constraint, i.e., size of largest connected component in the sampled graph does not exceed the threshold B. If this constraint is violated by an inserted edge, we discard this edge and move on to the next edge. This discarded edge is considered conflicting with current set of sampled edges. Once we iterate over all edges, we declare connected components in the sampled graph as clusters. Note that the sampled graph maintains two properties: *conformity* and *maximality*. Conformity means it satisfies the clustering constraint, while maximality means we cannot add more edges to it from the original graph without violating the constraint. This algorithm would work well but running it again and again for every inserted or removed edge will be highly inefficient. In the rest of this section we describe how to implement this algorithm in an incremental fashion such that the sampled graph is always conformable and maximal.

Figure 4 shows an example of the state of the reservoir manager. As shown, reservoir manager keeps two reservoirs of edges, *structural reservoir* and *support reservoir*. Structural reservoir contains sampled edges that were picked up so far and they are all inserted in the graph manager. Support reservoir is used to recover the maximality of sampled edges as edges are removed from structural reservoir. As mentioned earlier, all edges are stored in a random order so that they are processed in a random order. To accomplish this, each edge is assigned a random position in the range $(0, 1]$ and edges are kept sorted according to this random position.

1) Insertion: Algorithm 1 is used to insert a new edge to the reservoir manager. This algorithm is called whenever a new edge is inserted in the original graph. It is assumed before invoking this algorithm that the structural reservoir (*sampled graph*) is both conformable and maximal. The goal of this algorithm to ensure that both conformity and maximality

Algorithm 1 Reservoir Manager insert algorithm

```
1: Function ReservoirManagerInsert($newEdge$, $structuralReservoir$, $supportReservoir$, $graphManager$)
2: Generate a random position $pos \in (0, 1]$ for the $newEdge$
3: Insert the $newEdge$ in $structuralReservoir$ at position $pos$
4: Insert the $newEdge$ in the $graphManager$
5: if constraint is not satisfied in $graphManager$ then
6:     while constraint is not satisfied in $graphManager$ do
7:         Remove $lastEdge$ with $highestPosition$ from $structuralReservoir$
8:         Insert the $lastEdge$ to $supportReservoir$ at its assigned $position$
9:         Remove $lastEdge$ from $graphManager$
10:    end while
11:    Search for all edges from $supportReservoir$ with positions higher than the last removed edge in line 7
12:    for each $edge$ in search results do
13:        Insert the $edge$ in $graphManager$
14:        if constraint is not satisfied in $graphManager$ then
15:            Delete the $edge$ from $graphManager$
16:        else
17:            Move the $edge$ from $supportResevoir$ to $structuralReservoir$
18:        end if
19:    end for
20: end if
```

properties hold after inserting the new edge. The first step in lines (2-4), the new edge is assigned a random position and added to sampled graph and structural reservoir. Next, it tests for conformity property by checking the constraint in the sampled graph (line 5). If the constraint is satisfied, we know that the current sample is conformable. It is also maximal because it was maximal before inserting the new edge which means no more edges can be added. However, if the constraint is not satisfied, the algorithm needs to restore the conformity of the sample and this is done in two phases: *remove* phase and *add* phase. In the remove phase, the conformity of the sample is restored by unsampling some edges, i.e., moving edges from structural to support reservoir. While restoring the conformity, we might have broken the maximality of the sample, thus, in the add phase, we restore the maximality by sampling some edges back from support reservoir to structural reservoir.

The remove phase is performed in the loop in lines 6-10. It keeps moving edges from structural reservoir to support reservoir until the required property is restored again. In lines 7-9, we remove the last sampled edge from structural reservoir and move it to support reservoir. As we described earlier, the technique is to sample edges in the generated random order, this implies unsampling edges in the reverse order. Removing an edge from structural reservoir is reflected to current graph by removing this edge from graph manager (line 9). We repeat this process and keep checking the required constraint until it is satisfied again.

Once the first loop finishes, we are sure that the current sample is conformable but it might not be maximal. To restore maximality, we need to check if we can move edges from support reservoir to structural reservoir without breaking the conformity of the sample. Note that we need only to check edges that come after the first deleted edge. The reason is that we always sample edges from left to right (i.e., in ascending order), which means that edges which were not sampled conflict with edges left to it. These non-sampled edges do not have a chance to be sampled unless one of these preceding edges are deleted. In line 11, we issue a search query for all edges that are candidate to be sampled. In lines 12-19, we test each of these edges in order, according to their positions, to see which ones can be sampled without breaking the conformity. In line 13, we add next edge to sampled graph. If this edge breaks the constraint, we remove it back from the sampled graph and move on to next edge in search results. If constraint is not broken as a result of adding this edge, we complete the sampling of this edge by moving it from support to structural reservoir. Once the second loop is finished, we are sure that we restored both conformity and maximality sampled graph.

Algorithm 2 Reservoir Manager delete algorithm

```
1: Function ReservoirManagerDelete($edge$, $structuralReservoir$, $supportReservoir$, $graphManager$)
2: Try to remove the edge from $supportReservoir$
3: if edge is present in $structuralReservoir$ then
4:     Remove the edge from $structuralReservoir$ and $graphManager$
5:     Search for all edges from $supportReservoir$ with positions higher than the removed edge
6:     for each $edge$ in search results do
7:         Insert the $edge$ in $graphManager$
8:         if constraint is not satisfied in $graphManager$ then
9:             Delete the $edge$ from $graphManager$
10:        else
11:            Move the $edge$ from $supportResevoir$ to $structuralReservoir$
12:        end if
13:    end for
14: end if
```

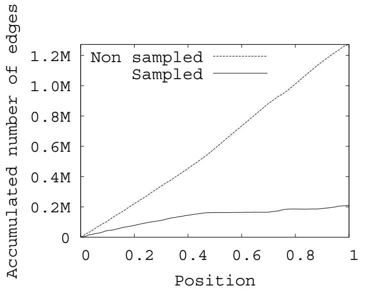


Fig. 5. Accumulated number of edges at different positions

2) Deletion: Algorithm 2 shows the deletion (eviction) algorithm for reservoir manager. The deleted edge might be either in the structural or the support reservoir. In line 2, we try to delete the edge from support reservoir. If the edge was indeed in support reservoir, we do not need to do any more steps as the sampled graph is not affected, hence it is still conformable and maximal. If the edge was not found in support reservoir, it means it is currently sampled and need to be deleted from both structural reservoir and graph manager (line 4). Deleting an edge from current sampled graph might cause it to be not maximal any more. Thus, we need to visit edges in support reservoir and check if any of these edges can be added to current sample or not. This is exactly the same as the add phase in the algorithm for insert. Lines 6 through 13 of algorithm 2 are an exact copy of lines 12 through 19 of algorithm 1.

3) Space and Processing Optimization: While running structural sampler over a wide range of real graphs, we noticed one important observation depicted in figure 5. Recall how the algorithm iterates over all edges in the random order and samples some edges in this order. This figure show total number of edges sampled so far as the algorithm is checking edges comparing it to number of non-sampled edges. We drew this figure by taking a snapshot while the system is running and accumulating number of sampled and non-sampled edges at different positions. At each position, the solid line represents total number sampled edges with lower positions, while the dotted line represents total number of edges that were not sampled or were redundant. Non sampled edges are edges that caused size of largest cluster to grow beyond B, and hence, were moved to the support reservoir. Redundant edges are those edges that are connecting two vertices already in the same cluster. Although redundant edges are sampled in the structural reservoir, they do not actually affect the final answer because they do not affect the structure of connected components. We can easily notice that more than 90% of sampled edges are at positions less than 0.5. The reason is that reservoir manager samples edges in their ascending positions, and it is most likely for edges near the end to be conflicting with previous edges and hence not sampled.

This observation inspires us to apply a sampling threshold for inserted edges. The *sampling threshold* is exactly the parameter p used in the intuition given in Section I-C. A sampling threshold is a number $p \in (0, 1]$ such that all edges with positions larger than p, are not processed or even stored. Choosing a lower sampling threshold saves more memory and processing but it might affect the quality of final answer because more edges are dropped without processing. Similarly, increasing the sampling threshold produces higher quality answers at the cost of more memory and processing time. Sampling threshold can be carefully adjusted so that we save much space while keeping clustering quality almost the same. In section IV, we show the effects of changing sampling threshold for both quality and performance.

D. Graph Manager

Graph manager is the component that handles all graph logic and is used by the reservoir manager to ensure that the clustering constraint is satisfied. As this graph manager is not actually our main contribution, we will go quickly over its implementation. The main data structures used are for storing current sampled graph and connected components of this graph. Since real graphs tend to be very sparse, we store graph structure as an adjacency list. To keep track of connected components, we use the *union find* data structure to group all vertices in one connected component together as a set. The union find structure is simply a forest where each tree represents a set. Each vertex in the *graph* is mapped to exactly one vertex in the union find structure. Each set (tree) in the union find represents one connected component in the graph. Adding a new edge might force two connected components to be merged together; in this case, all we need to do is making the root of one connected component a child of the root of the other connected component. We also keep track of the size of each connected component and update accordingly when two components are merged.

We also need to check whether the largest connected component has grown beyond the threshold B or not. To accomplish this, we always keep track of the last edge inserted. When we need to check the constraint satisfaction, we only need to check the connected component that contains last inserted edge. Whenever a connected component grows in size beyond the defined threshold B, we fix it by deleting some edges. This means that there can be only one connected component breaking the constraint at any time and this would be the last one that grew up in size.

Deleting an edge is a little bit tricky. Deleting an edge from a connected component might break it into two components or might leave it unaffected. Since the union find data structure does not store the actual edges in the underlying graph, we have no clue how deleting an edge affects connected components. Our approach is very simple and effective. We know for sure that deleting an edge from a connected component does not affect other connected components. So, we delete the affected connected component from the union find data structure and rebuild it using all edges in the adjacency of affected nodes. Although the deletion algorithm used here is not efficient, we still get orders of magnitude performance improvement which actually shows the efficiency of our approach for processing stream graphs.

We remark that one can also use a dynamic algorithm for connectivity queries given by Henzinger and King [15]. Their algorithm takes amortized poly-logarithmic time per update or query, but is quite complex to implement. We decided use a simpler union-find based algorithm because the connectivity is not the central focus in this work.

E. Parallelized/Distributed Structural Sampler

What if a single host does not have enough memory to store all the edges in the two reservoirs? In such a case, we must distribute the storage and computational requirement across multiple hosts. In this section, we briefly sketch how to do this. Full discussion is omitted due to lack of space.

Recall that our structural sampler is based on the assignment of a random number r_e with each edge e. Such a random number can be assigned using *universal hashing* [16]. One can use the simplest form of universal hashing based on modular arithmetic as explained by Carter and Wegman [16]. Once we choose a random hash function H from a universal hash family, we assign $r_e = H(id(e))$ to each edge e, where $id(e)$ is a unique identifier associated with edge e. We subdivide the range $(0, 1]$ of the hash function into h ranges $(0, R_1], (R_1, R_2], \ldots, (R_{h-1}, 1]$, where h denotes the number hosts in our distributed implementation. Here $0 < R_1 < R_2 < \ldots < R_{h-1} < 1$ are chosen based on the capacities of different hosts. Host i is responsible for maintaining the parts of the structural and support reservoirs that fall in range i. Host i also stores the graph manager that manages the subgraph i which is induced on the edges falling in range i. There is a *front-end* host that ingests the update and query streams, assigns the random indices based on the hash function and kicks off the algorithm for edge additions or deletions. During edge additions or deletions, since the updates to the structural and support reservoirs take place, first in the decreasing order of r_e values (remove phase) and then in the increasing order of r_e values (add phase). Therefore, the control travels between hosts i, first in the decreasing order of i and then in the increasing order of i.

The connectivity queries on the entire graph can be realized in this distributed implementation as follows. The graph manager on host i maintains a spanning forest of subgraph i. It forwards the edges of the spanning forest to a *back-end* host. The back-end host maintains a graph which is the union of these spanning forests for all values of i. It is easy to see that the connectivity query on the entire graph is equivalent to the connectivity query on this union. Upon edge additions or deletions from the graph, the appropriate host i adds or deletes the edges in subgraph i. It then forwards the changes to the edges in its spanning forest to the back-end host. The back-end host then updates the union as necessary.

IV. Experiments

To test the quality and performance of structural sampler, we carry out a set of experiments in this section. We compare our algorithm to two previous algorithm: AZY [14] and METIS [7]. AZY was introduced as an outlier detection algorithm but it can still be used for clustering. Since the original version of AZY does not support edge deletions, we modified it by adding a support reservoir which is used for deletions. METIS was not designed for stream applications, but we can use it in a straight forward way. As edges are inserted or deleted, we keep a list of all edges in the graph. Whenever a query is issued, we run METIS over the current set of edges and return the result. This is considered the naïve implementation for graph clustering over streams. We show the results for three sets of experiments in this section: quality, performance and tuning experiments.

- **Quality**: Using the *cut-size* quality measure, we show that our approach gives reasonable quality compared to METIS which is assumed to be close to the best clustering.
- **Performance**: We test the performance of each approach in terms of average throughput (events processed per second). We show that our approach beats METIS by several orders of magnitudes in performance.
- **Tuning**: We show that the sampling-threshold parameter p can be carefully selected to achieve higher performance while keeping the quality stable.

A. Experimental Setup

Table I summarizes the data sets used in our experiments. As seen in table, we use data sets from different domains for a wider range of structures (e.g. social networks, web graphs and paper citations). The *cit-HepPh* dataset represents citations of 'High Energy Physics' papers from arXiv. It covers papers on a range of 124 months and we use actual publish dates as timestamps. A paper x citing a paper y is represented by an edge between x and y with timestamp equal to the publish date of x. An interesting property of this dataset is that number of vertices keeps increasing with time because old papers still appear as they are cited by newer papers. The *web-NotreDame* dataset is a snapshot taken in 1999 from the web graph of University of Notre Dame. An edge between pages x and y represents a hyperlink in one page pointing to the other. This dataset has no timestamps, so we assign a random timestamp to each edge so that we can use it in our system. The *replies* dataset is a sample extracted from Twitter over five days of usage. In this dataset, users are represented by vertices and each edge between users x and y means that one of these users has replied to the other one. Since this dataset represents only 1% of actual usage of Twitter, we found that this dataset is very sparse and it does not cluster well even with METIS. The last dataset is *DNS Edges* which contains a one day's worth of DNS requests from hosts within IBM Watson Research Center to domains on the web. An interesting structure in this dataset is that it contains lots of duplicate edges at different timestamps meaning that the same host requests the IP of a certain domain multiple times a day. At some extremes, there are hundreds of edges between the same pair of vertices. To model edges insertions and deletions for these datasets, we use a count-based sliding window of size W (varies from 1K to 10K). For quality experiments, we use a tumbling window of size W and measure the cut-size when each window is filled.

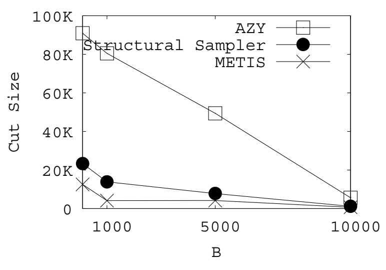
Name	Vertices	Edges	Description
cit-HepPh	$34k$	$420k$	Paper citations in physics papers
web-NotreDame	$330k$	$1.5M$	Web graph from Univ. of Notre Dame
replies	$1.9M$	$1.6M$	Replies between Twitter users
DNS Edges	$180k$	$4.8M$	DNS requests

TABLE I
Real data sets used in experiments

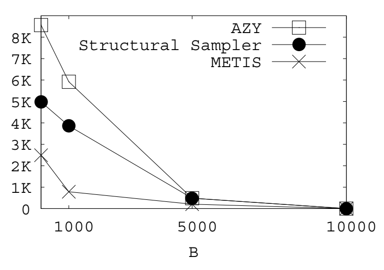
Our primary focus in these experiments is to evaluate the algorithm in terms of its cut quality. Therefore we use a single-host implementation. As described in Section III-E, the algorithm can also be implemented in a distributed manner. We use a Linux machine with Red Hat Enterprise 5.5, 32 GB physical memory and Intel Xeon X5570 2.93 GHz. We implement all the algorithms within IBM's System S [17], a platform for streaming computations. Since System S allows using C and C++ libraries within its operators, we are able to use METIS out of the box and only built a wrapper around it to be able to plug it into System S.

B. Quality Experiments

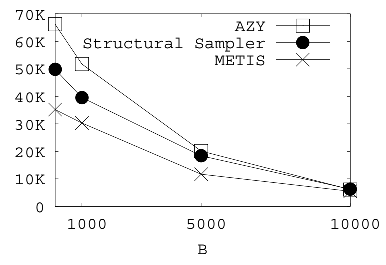
The first measure we care about is the clustering quality of our new technique. Since we work in a more restrictive environment where edges are received one by one and clustering need to be updated, it would be difficult to improve upon the quality of clustering computed by an offline algorithm such as METIS. Figure 6 shows results of quality experiments we performed over several data sets with different values of clustering parameter B while fixing p at 1.0. As mentioned earlier, we use the cut-size as our measurement for quality where lower values are better. To calculate the quality in a



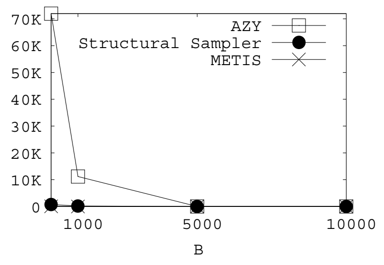
(a) Notre Dame Web graph



(b) Citations



(c) DNS requests



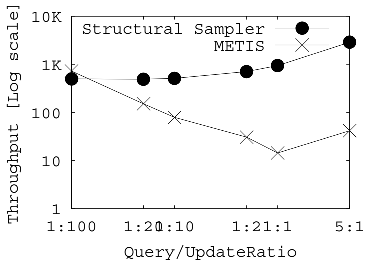
(d) Twitter replies

Fig. 6. Quality experiments on different datasets

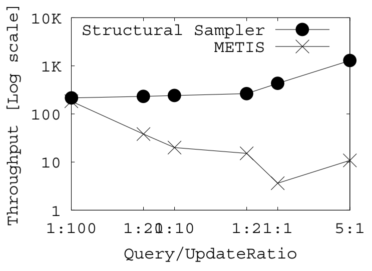
streaming system, we take a snapshot after processing every $10k$ updates and calculate the quality measurement for this snapshot. Then, we calculate the average over all snapshots to come up with one number representing the average quality for one run. Since the meaning of this measurement is specific to each dataset, we could not plot all numbers in one figure and we had to plot one figure for each dataset. As expected, METIS consistently gives the best quality on all datasets because it works in an offline mode, i.e., after receiving all edges the in current window. *Structural Sampler* comes in second place on all our experiments and gives better quality than AZY as it samples more edges. What is surprising to us is that, despite its simplicity, our approach gives results that are close to a well-celebrated offline algorithm such as METIS.

C. Performance Experiments

We designed our system to overcome the main limitation in offline algorithms which is the very low throughput. Figure 7 shows the performance of both METIS and structural sampler for different datasets and workloads. The workload employed in these experiments consists of randomly generated queries in the form "Are v_1 and v_2 connected at time t?". The query/update ratio indicates the average ratio between number of query events and number of update events in a time window. For example, a '1:5' query/update ratio indicates that there is approximately one query after each five updates. At very small values, structural sampler does a lot of work for all the updates which decreases the system throughput. At the same time, METIS does not actually do anything until a query event is received which makes it achieve high throughput values for low query/update ratios. As the workload ratio increases, METIS does worse while structural sampler gets better and



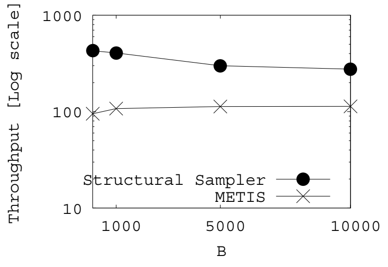
(a) Citations



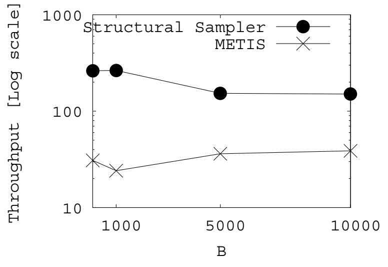
(b) Web graph

Fig. 7. Performance experiments for different workloads

better. Recall that we calculate system throughput using both query and update events. While *structural sampler* does the same work to process update events and maintain clustering, the more queries coming to the system are answered in almost constant time, hence, throughput increases. On the other hand, METIS gets slow as it need to re-cluster the graph for each new query coming decreasing the overall system throughput. Eventually, when query/update ratio grows beyond 1, the



(a) DNS requests



(b) Web Notre Dame

Fig. 8. Performance experiments for different clustering bounds B. Query/Update ratio is fixed at $1/2$.

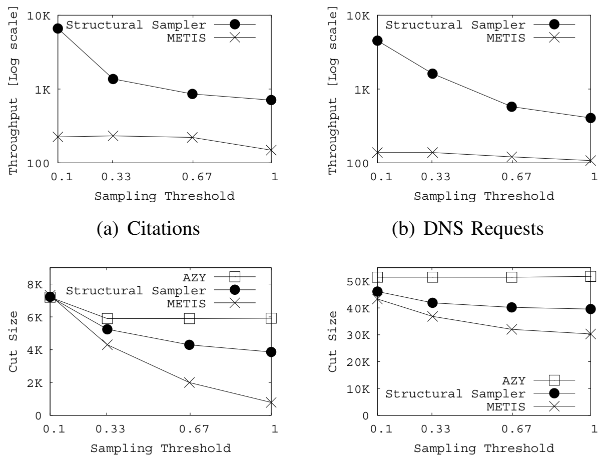


Fig. 9. Tuning experiments show the effect of sampling-threshold on both performance and quality.

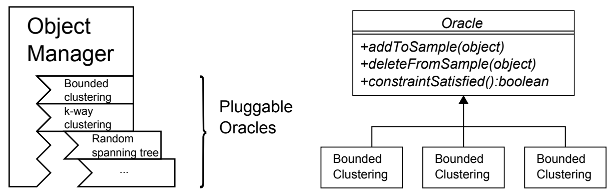
throughputs for both systems grow very rapidly. The reason is that beyond the ratio of 1, the system is expected to receive a bunch of queries before receiving the next update event. This means that the graph is clustered once to answer this bunch of queries causing the throughput of both techniques to increase as shown in figure 7.

Figure 8 shows the results of more performance experiments we carried out to compare both approaches. In these experiments, query/update ratio is fixed at $1/2$ and clustering bound is changed. We can see that structural sampler is consistently better than METIS in performance. METIS gives almost the same throughput for different values of B which is expected as its running time is controlled by size of the graph which remains constant in these experiments. On the other hand, we see in figure 8 that the performance of structural sampler decreases as the cluster bound increases. Our interpretation for this is that as B increases, computed clusters tend to be of larger sizes. As mentioned in section III-D, deletion is implemented by breaking then rebuilding a cluster which becomes time consuming with larger clusters. We believe that implementing better algorithms for graph manager could stabilize and increase performance of our approach but it does not affect our general approach.

D. Tuning Experiments

As shown in section III-C3, a sampling threshold (p) can be applied to structural sampler to increase its performance. We carried out another set of experiments to see the effect of sampling threshold on both performance and quality. Figures 9 shows its effect of changing sampling threshold on both performance (Figures 9(a) and 9(b)) and quality (Figures 9(c) and 9(d). We can see that carefully choosing the right sampling threshold can achieve orders of magnitude improvement in performance without losing much quality. Applying sampling threshold is just a preprocessing phase that does not change the underlying algorithms. In fact, we could also apply sampling threshold to METIS by storing and processing only edges that were found to be below the threshold. For example, if sampling threshold is set to 0.3, only 30% of edges are kept and METIS is applied to this subset of edges. The effect of sampling threshold on METIS is very limited but actually it is giving the same trend as our approach, i.e. a lower sampling threshold increases performance. The limited effect on METIS is because METIS running time depends mainly on number of vertices which does not change significantly in these datasets when applying a sampling threshold. On the other hand, the running time of our approach depends on number of edges in the original graph which is why we see significant improvement in performance.

The drawback of applying a lower sampling threshold is possibly a worse clustering quality. Figures 9(c) and 9(d) show the effect of sampling threshold on quality for different datasets. When we apply a sampling threshold, most probably we drop edges that do not affect the solution and hence overall quality is hardly affected. For example in figure 9(d), applying a sampling threshold of value 0.33, gives a quality very close to the best quality achieved when sampling threshold is equal to 1.0. At 0.1, all approaches do almost the same because they operate on 10% of original edges leaving them with very limited room to optimize. We include AZY here to verify the effect we expected to occur when applying sampling threshold. Recall that AZY only samples a prefix of edges in the random order. Applying a higher sampling threshold would not affect the sampled edges at all and would produce the same answer, while applying a lower sampling threshold would affect the quality. This can be verified from figures 9(c) and 9(d).



(a) Generalized object manager with pluggable oracles (b) Abstract ObjectManager class with possible extensions

Fig. 10. Architecture of the generalized structural sampler with possible extensions

V. Extensions

The approach we use in structural sampler for graph clustering can actually be generalized to several other problems. Recall the architecture in figure 2. The only component that handles graph specific logic is the graph manager. We show in this section how we can use the same approach and framework for other applications like min-cut k-way clustering. Figure 10 outlines our generalization. We replace the graph manager in figure 2 with the object manager shown in Figure 10(a). The framework itself remains almost the same. The main difference is that we no longer deal with graph edges, rather with objects of an arbitrary type. The reservoir manager keeps *structural* and *support* reservoirs of these objects as before. Whenever an object is added to or removed from the structural reservoir, this change is also reflected in the *object manager*. The constraint checking, if any, is done by the object manager.

The object manager itself may use one or more *oracles* to enforce relevant constraints. As shown in Figure 10(b), the oracle is a class that implements the three functions described below. This framework allows creating different oracles each dealing with a specific problem without having any reservoir-related logic. Then, these oracles can be easily plugged into the system and gain all the advantages of a running on a streaming system for free. As a proof of concept, we describe briefly how one can build the *min-cut k-cut oracle*. Here the objective is to partition the vertices into at least k clusters (not necessarily balanced) while minimizing the number of inter-cluster edges. As shown in Figure 10(b), all we need is to create a C++ class that implements three functions. The class stores a data structure that holds graph structure as an adjacency list. It also stores connected components of the graph in a union-find data structure. The method `addToSample` adds the given object (edge in this case) to the adjacency list and merges the connected components at the the two end points of this edge (if different) into one connected component. `deleteFromSample` will do the opposite of this. First, it removes the given edge from the adjacency list. It then updates the affected connected component by deleting it and rebuilding it again from edges stored in the adjacency list. The last method is `constraintSatisifed` and it checks whether the constraint is currently satisfied or not. This method returns `true` if there are at least k connected components.

In general, we can enforce any *monotone* constraint or property. A non-trivial property P of subsets of a universe U is called (downward) *monotone* if

- the empty set $\emptyset$ satisfies P, and
- if S satisfies P and $T \subset S$, then T also satisfies P.

For example, the property of a subset of edges that it induces at least k connected components is monotone (if there are at least k vertices). Similarly, the property of a subset of edges that it connected components, each of size at most B is monotone. This generalization is also motivated by the discussion on monotone properties in AZY [14].

VI. Conclusions

In this paper, we used graph reservoir sampling to design a streaming algorithm for clustering vertices in graphs. Our scheme can be implemented in a dynamic setting of edge additions and deletions. In spite of being extremely simple, our scheme performs fairly well as compared to METIS in terms of cut-quality and outperforms it by orders of magnitude in terms of the throughput it can handle. It can also be generalized to other problems like min-cut k-way clustering.

References

[1] Twitter Wikipedia entry. [Online]. Available: http://en.wikipedia.org/wiki/Twitter

[2] A. Jain and R. Dubes, *Algorithms for Clustering Data*. Prentice-Hall, 1988.

[3] L. Kaufman and P. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley and Sons, 1990.

[4] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins, "Trawling the Web for Emerging Cyber-Communities," *Computer Networks*, vol. 31, no. 11-16, pp. 1481–1493, 1999.

[5] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical Journal*, vol. 49, pp. 291–307, 1970.

[6] R. Sokal and P. Sneath, *Numerical Taxonomy*. Freeman, 1973.

[7] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[8] N. Bansal, A. Blum, and S. Chawla, "Correlation Clustering," *Machine Learning*, vol. 56, no. 1-3, pp. 89–113, 2004.

[9] F. T. Leighton and S. Rao, "Multicommodity Max-Flow Min-Cut Theorems and Their Use in Designing Approximation Algorithms," *Journal of the ACM*, vol. 46, no. 6, pp. 787–832, 1999.

[10] M. Charikar, "Greedy Approximation Algorithms for Finding Dense Components in a Graph," in *Intl. Workshop on Approximation Algoritms for Combinatorial Optimization*, 2000, pp. 84–95.

[11] D. Karger, "Global Min-cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm," in *ACM-SIAM Symposium on Discrete Algorithms*, Austin, TX, Jan 1993, pp. 21–30.

[12] H. Zanghi, C. Ambroise, and V. Miele, "Fast Online Graph Clustering Via Erdos-Rényi Mixture," *Pattern Recognition*, vol. 41, no. 12, pp. 3592–3599, 2008.

[13] C. Aggarwal, Y. Zhao, and P. Yu, "On Clustering Graph Streams," in *Proceedings of the SIAM International Conference on Data Mining*, Columbos, OH, Apr 2010.

[14] ——, "Outlier Detection in Graph Streams," in *Proceedings of the International Conference on Data Engineering, ICDE*, Hannover, Germany, Apr. 2011, pp. 399–409.

[15] M. Henzinger and V. King, "Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time Per Operation," *Journal of the ACM*, vol. 46, pp. 502–516, 1999.

[16] J. Carter and M. Wegman, "Universal Classes of Hash Functions," *Journal of computer and system sciences*, vol. 18, no. 2, pp. 143–154, 1979.

[17] K.-L. Wu, P. Yu, B. Gedik, K. Hildrum, C. Aggarwal, E. Bouillet, W. Fan, D. George, X. Gu, G. Luo, and H. Wang, "Challenges and Experience in Prototyping a Multi-Modal Stream Analytic and Monitoring Application on System S," in *Proceedings of the International Conference on Very Large Data Bases, VLDB*, Sep.