Graph Stream Summarization

From Big Bang to Big Crunch

Nan Tang Qing Chen Prasenjit Mitra
Qatar Computing Research Institute, HBKU
{ntang, qchen, pmitra}@qf.org.qa

ABSTRACT

A graph stream, which refers to the graph with edges being updated sequentially in a form of a stream, has important applications in cyber security and social networks. Due to the sheer volume and highly dynamic nature of graph streams, the practical way of handling them is by summarization. Given a graph stream G, directed or undirected, the problem of *graph stream summarization* is to summarize G as S_G with a much smaller (*sublinear*) space, *linear* construction time and *constant* maintenance cost for each edge update, such that S_G allows many queries over G to be approximately conducted efficiently. The widely used practice of summarizing data streams is to treat each stream element independently by *e.g.,* hash- or sample-based methods, without maintaining the connections (or relationships) between elements. Hence, existing methods can only solve *ad-hoc* problems, without supporting diversified and complicated analytics over graph streams. We present **TCM**, a novel graph stream summary. Given an incoming edge, it summarizes both node and edge information in constant time. Consequently, the summary forms a graphical sketch where edges capture the connections inside elements, and nodes maintain relationships across elements. We discuss a wide range of supported queries and establish some error bounds. In addition, we experimentally show that **TCM** can effectively and efficiently support analytics over graph streams beyond the power of existing sketches, which demonstrates its potential to start a new line of research and applications in graph stream management.

CCS Concepts

•Mathematics of computing → Network flows; •Information systems → Data streams; •Theory of computation → Sketching and sampling;

Keywords

Graph streams; Data streams; Sketch; Summarization

SIGMOD '16, June 26–July 1, 2016, San Francisco, CA, USA.

 ISBN 978-1-4503-3531-7/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2882903.2915223

1. INTRODUCTION

Nowadays, massive graphs arise in many applications *e.g.,* network traffic data, social networks, and transportation networks. These graphs are highly dynamic: Network traffic data averages to about 10^9 packets per hour per router for large ISPs [23]; Twitter sees 100 million users login daily, with around 500 millions tweets per day [1].

As conventionally formulated, we consider a graph stream as a sequence of elements $(x, y; t)$[1], which indicates that the edge (x, y) is encountered at time t. A sample graph stream $\langle (a, b; t_1), (a, c; t_2), \cdots, (b, a; t_{14}) \rangle$ is depicted in Fig. 1, where all timestamps are omitted for simplicity. Each edge is associated with a weight, which is 1 by default.

Practically, real-time analytics over large graph streams has wide applications, such as monitoring cyber security attacks and social network opinions. To a large extent, in the era of big data, the increasing expansion of data velocity is an open issue, and many big data is graphs inherently. Unfortunately, in a recent survey for businesses in analyzing big data streams [2], although 41% of respondents stated that the ability to analyze and act on streaming data in minutes is critical, 67% admitted that they do not have the infrastructure to support that goal. The situation is more complicated and challenging when processing graph streams. It poses unique space and time constraints on storing, summarizing and querying graph streams, due to their sheer volumes, high velocity, and the complication of analytics.

Fortunately, for the applications of data streams, fast and approximated answers are often preferred than exact answers. Not surprisingly, sketch synopses have been widely studied for data streams: approximate frequency counts [29], AMS [5], and Bottom-k [13]. Although they allow false positives, the space savings often outweigh this drawback, when the probability of an error is sufficiently low.

We illustrate by an example how approximate frequency counts [29] works. CountMin [14] improves it simply by using multiple hash functions, and gSketch [39] further improves CountMin by assuming the presence of sample data/queries that help better partition the input graph stream.

Example 1: Consider the graph stream in Fig. 1. Approximate frequency counts based methods can be used to treat each node or edge in the stream independently, by mapping them to w hash buckets. Let $w = 4$ in this example.

(1) Node sketch: It treats node labels as hash keys, *i.e.,* $\{a, b, c, d, e, f, g\}$. A hash function h is used to map these

[1]Without loss of generality, we use a directed graph for illustration. Our method can also work on undirected graphs.

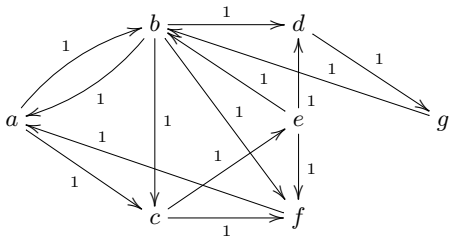


Figure 1: A sample graph stream

$h(\cdot)$	4	5	2	3
keys	a, c	b, d	e, g	f

(a) ***Node sketch***

$h'(\cdot)$	5	5	3	1
keys	ab, ac, ed, eb, ef	bc, bd, ba, bf, fa	ce, cf, gb	dg

(b) ***Edge sketch***

Figure 2: Examples of frequency counts

values to 4 buckets, *i.e.,* $h(\cdot) \rightarrow [1, 4]$, as shown in Fig. 2 (a). The first bucket sums up the in-flow weights to nodes a and c, which is 4, and similar for other buckets.

(2) Edge sketch: It treats pairs of node labels as hash keys, *e.g.,* $\{ab, ac, bc, \cdots\}$, where ab is to concatenate the two node labels for $(a, b; t_1)$. A hash function h' is used to map these values in 4 buckets, as shown in Fig. 2 (b). The value of the first bucket is the sum of weights for edges $\{ab$, $ac, ed, ed, ef\}$, which is 5, and similar for other buckets. □

The power of frequency counts based approaches is limited inherently by its structure: It is a vector of (hashed key, value) pairs and each hashed key is independent of each other. That is, one cannot reason for any relationships between two hashed keys. In fact, CountMin or gSketch cannot support any more queries than their frequency counts counterparts, since they are proposed to improve the accuracy. We give more details of supported queries below.

Example 2: Consider the two sketches in Example 1 that are depicted in Fig. 2. Below, we denote by ✓ the types of queries a sketch supports, and ✗ for those not supported.

(1) Node sketch (see Fig. 2 (a)).

✓ *Node queries*: It can only be used to estimate an in-flow weight of a node, *e.g.,* the estimated in-flow weight to node a is in the first bucket since $h(a) = 1$, which gives 4.

✗ *Conditional node queries*: It cannot find locally important node, *e.g.,* which is the most frequent node linking to node a, which relates to the conditional heavy hitter problem.

(2) Edge sketch (see Fig. 2 (b)). (Note: gSketch only supports the type of queries that an edge sketch supports.)

✓ *Edge queries*: What is the estimated edge weight? For instance, it is 5 for edge (a, b) from the first bucket since $h'(ab) = 1$.

✓ *Aggregate subgraph queries*: What is the aggregated weight for a graph with all edges being given explicitly? For instance, for a graph with two edges (a, c) and (c, e), the estimated weight for ac (resp. ce) is 5 (resp. 3), which sums up to be 8.

✗ *Node connectivity:* Whether two nodes are connected? It cannot answer whether there is a path from a to g. □

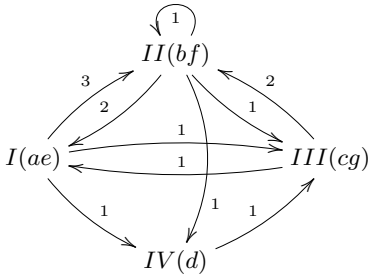


Figure 3: An example of our graphical sketch

Challenges. Designing a generalized sketch for graph streams to support all above analytics and beyond requires to satisfy the following constraints. (1) *Space constraint: sublinear* upper bound is required. (2) *Time constraint: linear* construction time is a must. Notably, this is stronger than the condition with only a constant passes over the stream. (3) *Maintenance constraint:* to maintain it for one element insertion/deletion must be in *constant* time. (4) *Graphical connectivity:* it should have a graphical model.

To solve the above challenges, we propose **TCM**, a generalized graph stream summary. Given an incoming element (*i.e.,* an edge), it summarizes both node and edge information in constant time. In contrast to previous sketches that have *one-dimensional* data structures, **TCM** is a graphical sketch with a *two-dimensional* data structure.

Example 3: Again, consider the graph stream in Fig. 1. Our proposal is shown in Fig. 3. For each edge $(x, y; t)$, **TCM** uses a hash function to map each node label to 4 node buckets *i.e.,* $h''(\cdot) \rightarrow [1, 4]$. Node I is the summary of two node labels a and e, assuming $h''(a) = 1$ and $h''(e) = 1$. The other compressed nodes are computed similarly. The edge weight denotes the aggregated weights from stream elements, *e.g.,* the number 3 from node I to II means that there are three elements as $(x, y; t)$ where the label of x (resp. y) is a or e (resp. b or f). □

From Example 3, it is easy to see that the queries supported by existing node or edge sketches are naturally supported, *e.g.,* the estimated in-flow weights to node a is 3 and the estimated weight of edge (a, b) is 3. Moreover, the other queries that are not supported by existing sketches are also supported, *e.g.,* the node that sends the most weights to node a is either b or f, and there is a path from a to d. The *essential* difference of **TCM**, compared with existing sketches, is that **TCM** keeps all structural connectivities of the original graph stream, *not only* edges, *but also* paths. For instance, the path $a \rightarrow b \rightarrow d \rightarrow g$ in Fig. 1 is captured in Fig. 3 as $I \rightarrow II \rightarrow IV \rightarrow III$. This salient property provides rich structural information to support various graph analytics (see Section 4 for a detailed discussion).

Contributions. This paper presents a novel generalized graphical sketch for summarizing graph streams, with the following contributions.

(1) We formally introduce **TCM** (Section 3). As illustrated in Fig. 3, the proposed sketch naturally preserves the graphical connections of the original graph stream, which makes it a better fit than traditional sketches in supporting analytics over graph streams.

(2) We describe algorithms to process various graph analytics on top of **TCM** (Section 4). The general purpose is, instead of proposing new algorithms, to show that **TCM**

can be easily used to support many graph queries, and *off-the-shelf* graph algorithms can be seamlessly integrated.

(3) We describe internals of **TCM** implementation (Section 5). This is important to ensure that it can be built and maintained under the hard time/space constraints for graph stream scenarios. Moreover, we propose to use non-square matrices for storage, based on which we show that CountMin is a special case of **TCM**. Furthermore, we describe its extension to support high-dimensional data streams.

(4) Using real-world and synthetic data sets, we have experimentally evaluated the power of **TCM** in supporting different analytics (Section 6), which confirms that the new sketch is indeed effective and efficient, and more general than existing sketches for graph stream applications.

In this history of graph problems over streams, unfortunately, most results showed that a large amount of space is required for *complicated* graph problems [4,30]. The present study makes an important attempt to build a generalized sketch to support various graph analytics, using only a *small* amount of space, which sheds considerable light on graph stream applications.

Organization. Section 2 discusses related work. Section 3 introduces **TCM**. Section 4 presents its supported queries. Section 5 explains implementation details and extensions. Section 6 gives experimental findings. Finally, Section 7 concludes this work, followed by our agenda for future work.

2. RELATED WORK

We categorize related work as follows.

Sketch synopses. Given a data stream, the aim of sketch synopses is to apply (linear) projections of the data into lower dimensional spaces that preserve the salient features of the data. A considerable amount of literature has been published on general data streams such as AMS [5], lossy counting [29], CountMin [14] and bottom-k [13]. The work gSketch [39] improves CountMin for graph streams, by assuming that data or query samples are given. Bloom filters have been widely used in a variety of network problems (see [10] for a survey). There are also sketches that maintain counters only for nodes, *e.g.,* using a heap for maintaining the nodes with the largest degrees for heavy hitters [15].

As remarked earlier, **TCM** is more general, since it maintains connections of all streaming edges, *without* assuming any sample data or query is given. None of existing sketches for graph streams has a graphical model.

Graph summaries. Summarizing graphs has been widely studied. The most prolific area is in web graph compression. The papers [3,35] encode Web pages with similar adjacency lists using reference encoding, so as to reduce the number of bits needed to encode a link. The work [32] groups Web pages based on a combination of their URL patterns and k-means clustering. The paper [18] compresses graphs based on specific types of queries. There are also many clustering based methods from data mining community (see *e.g.,* [24]), with the basic idea to group *similar* nodes together. Another line of work is graph sparsificaiton, which is to approximate a graph G by a sparse graph H, where H and G have the same set of vertices and H is "close" to H in some specific metric, *e.g.,* cut sparsifiers [7,8] and spectral sparsifiers [6,34].

These data structures are designed for less dynamic graphs, which are not suitable for graph streams. Another reason that graph sparsification is not used for graph streams is the metric studied for sparsifying a graph is not suitable for graph stream applications, *e.g.,* network monitoring.

Graph pattern matching over streams. There have been several work on matching graph patterns over graph streams, based on either the semantics of subgraph isomorphism [12, 20,38] or graph simulation [33]. The work [38] assumes that queries are given, and builds node-neighbor tree to filter false candidate results. The paper [20] leverages a distributed graph processing framework, Giraph, to approximately evaluate graph quereis. The work [12] uses the subgraph distributional statistics collected from the graph streams to optimize a graph query evaluation. The paper [33] uses filtering methods to find data that potentially matches for a specific type of queries, namely degree-preserving dual simulation with timing constraints.

Firstly, all the above algorithms are designed for specific types of graph queries. Secondly, most of them assume the presence of queries, so they can build indices to accelerate. In contrast, **TCM** aims at summarizing graph streams in a generalized way, so as to support various types of queries, *without* any assumption of queries.

Graph stream algorithms. There has also been work on algorithms over graph streams (see [30] for a survey). This includes the problems of connectivity [19], trees [36], spanners [16], sparsification [26], counting subgraphs *e.g.,* triangles [9,37]. However, they mainly focus on theoretical study for best approximation bound, mostly on $O(n \text{ polylog } n)$ space, with one to multiple passes over the data stream.

TCM is complementary to the above algorithms. As will be seen later (Section 4), **TCM** can treat existing algorithms as *black-boxes* to help solve existing problems.

Distributed graph systems. Many distributed graph computing systems have been proposed to conduct data processing and data analytics in massive graphs, such as Pregel [28], Giraph[2], GraphLab [27], Power-Graph [21] and GraphX [22].

They have been proved to be efficient on static graphs, but are not fully geared for analytics over big graph streams with real-time response. In practice, they are complementary to and can be used for **TCM** in distributed settings.

3. THE TCM MODEL

We first define graph streams (Section 3.1). We then formulate the studied problem (Section 3.2). Finally, we introduce our proposed graphical sketch model (Section 3.3).

3.1 Graph Streams: Big Bang

A *graph stream* is a sequence of elements $e = (x, y; t)$ where x, y are node identifiers (labels) and edge (x, y) is encountered at time-stamp t. Such a stream,

$$G = \langle e_1, e_2, \cdots, e_m \rangle$$

naturally defines a graph $G = (V, E)$ where V is a set of nodes and E is a set of edges as $\{e_1, \cdots, e_m\}$. We write $\omega(e_i)$ the weight for the edge e_i, and $\omega(x, y)$ the aggregated edge weight from node x to node y. We call m the *size* of the graph stream, denoted by $|G|$. In this work, we assume that edge weight is non-negative (*i.e.,* $\omega(e_i) \geq 0$).

Intuitively, the node label, being treated as an *identifier*, uniquely identifies a node, which could be *e.g.,* IP addresses

[2]http://giraph.apache.org

in network traffic data or user IDs in social networks. Note that, in the graph terminology, a graph stream is a *multi-graph*, where each edge can occur many times, *e.g.,* one IP address can send multiple packets to another IP address. We are interested in properties of the underlying graph. This causes a main challenge with graph streams where one normally does not have enough space to record the graph that has been seen so far. Summarizing such a graph stream in one pass is important to many applications. Moreover, we do not explicitly differentiate whether the edge (x, y) is an ordered pair or not. In other words, our approach applies naturally to both directed and undirected graphs.

For instance, in network traffic data, a stream element arrives at the form: $(192.168.29.1, 192.168.29.133, 62, 105.12)$[3], where node labels 192.168.29.1 and 192.168.29.133 are IP addresses, 62 is the number of bytes sent *from* 192.168.29.1 *to* 192.168.29.133 in this captured packet (*i.e.,* the weight of the directed edge), and 105.12 is the time in seconds that this edge arrived when the server started to capture data. Please see Fig. 1 for a sample graph stream.

3.2 Graph Streams Summaries: Big Crunch

The problem of *summarizing graph streams* is, given a graph stream G, to compute another data structure S_G from G, such that:

1. $|S_G| \ll |G|$: the size of S_G is far less than G, preferably in sublinear space.
2. The time to construct S_G from G is in linear time.
3. The update cost of S_G for each edge insertion/deletion is in constant time.
4. S_G is a graph.

Intuitively, a graph stream summary has to be built and maintained in real time, so as to deal with big volume and high velocity graph stream scenarios. In fact, the CountMin [29] and its variant gSketch [39] satisfy the above conditions 1-3 (see Example 1 for more details). Unfortunately, CountMin and gSketch can support only limited types of graph analytics, since condition 4 is not satisfied.

3.3 Graphical Sketches

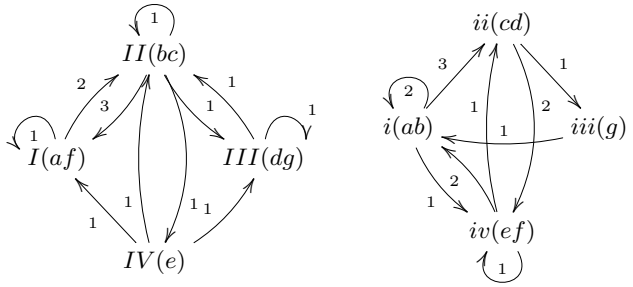
The graph sketch. A *graph sketch* is a graph $S_G(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes the set of vertices and $\mathcal{E}$ its edges. For a vertex $v \in \mathcal{V}$, we simply treat its label as its node identifier (the same as the graph stream model). Each edge e is associated with a weight, denoted as $\omega(e)$.

In generating the above graph sketch S_G from a graph G, we first set the number of nodes in the sketch, *i.e.,* let $|\mathcal{V}| = w$. For an edge $(x, y; t)$ in G, we use a hash function h to map the label of each node to a value in $[1, w]$, and the aggregated edge weight is calculated correspondingly.

Please refer to Fig. 3 as an example, where we set $w = 4$.

Edge weight. The edge weight for an edge e in the graph sketch is computed by an *aggregation* function of all edge weights that are mapped to e. Such an aggregation function could be $\mathsf{min}(\cdot)$, $\mathsf{max}(\cdot)$, $\mathsf{count}(\cdot)$, $\mathsf{average}(\cdot)$, $\mathsf{sum}(\cdot)$ or other functions. In this paper, we use $\mathsf{sum}(\cdot)$ by default to explain our method. The other aggregation functions can be similarly applied. In practice, which aggregation function to use is determined by applications. For a more general setting that requires to maintain multiple aggregated functions in a graph sketch, we may extend our model to have multiple edge weights *e.g.,* $\omega_1(e), \cdots, \omega_n(e)$, with each $\omega_i(e)$ corresponds to a distinct aggregation function.

[3]We omit port numbers and protocols for simplicity.



(a) Sketch S_1 (b) Sketch S_2

Figure 4: TCM with 2 hash functions

One may observe that the graph sketch model is basically the same as the graph stream model, with the main difference that the time-stamps are not maintained. This makes it very useful and applicable in many scenarios when querying a graph stream for a given time window. In other words, for a graph analytical method M that needs to run over a graph stream G, denoted as $M(G)$, one can run it directly on its sketch S_G, *i.e.,* $M(S_G)$, to get an approximate result, *without* modifying the method M.

Example 4: Consider the graph stream in Fig. 1 and its sketch in Fig. 3. Assume that query Q_1 is to estimate the aggregated edge weight from b to c. In Fig. 3, one can map b to node II, c to node III, and get the estimated weight 1 from edge (II, III), which is precise. Now consider Q_2 that is to compute the aggregated weight from g to b. One can find edge (III, II), and the estimated result is 2, which is not accurate since the real weight of (g, b) in Fig. 1 is 1. □

The above result is expected, since given the compression, no hash function can ensure that the estimation on the sketch can be done precisely. Along the same line of CountMin [14], we use multiple independent hash functions to collectively reduce the probability of hash collisions.

The TCM model. A **TCM** sketch is a set of graph sketches $\{S_1(\mathcal{V}_1, \mathcal{E}_1), \cdots, S_d(\mathcal{V}_d, \mathcal{E}_d)\}$. Here, we use d hash functions $h_1, \cdots, h_d$, where h_i $(i \in [1, d])$ is used to generate S_i. Also, $h_1, \cdots, h_d$ are pairwise independent hash functions (see Section 5.2 for more details).

Example 5: Figure 4 shows another two sketches for Fig. 1. Again, consider the query Q_2 in Example 4. Using S_1 in Fig. 4 (a), one can locate edge (III, II) for (g, b), which gives 1. Similarly, S_2 in Fig. 4 (b) will also give 1 from edge (iii, i), where g (resp. b) maps to iii (resp. i). The minimum of the above two outputs is 1, which is correct. □

Example 5 shows that using multiple hash functions can indeed improve the accuracy of estimation. In practice, using pairwise independent hash functions is a significant advance, since pairwise independent hash functions are generally easy to implement and quite efficient.

4. TCM POWERED GRAPH ANALYTICS

In this section we shall show how **TCM** can be easily used to support different graph analytics. Here, we consider the graph stream as G, and d graph sketches $\{S_1, \cdots, S_d\}$ of G, where S_i is constructed using a hash function $h_i(\cdot)$. Also, we

assume that an adjacency matrix $\mathbf{M}_i$ is used for storing one graph sketch S_i. We use $\mathsf{sum}(\cdot)$ as the default aggregation function for constructing graph sketches. Whilst the exercise in this section only substantiates several types of graph queries to be addressed in this work, it is evident that the power of **TCM** is far beyond the queries listed below.

As mentioned in Section 3.3, for any graph analytics method M to run over G, *i.e.,* $M(G)$, it is possible to run M on each sketch directly, and then merge the result as: $\tilde{M}(G) = \Gamma\big(M(S_1), \cdots, M(S_d)\big)$, where $\tilde{M}(G)$ denotes the estimated result over G, and $\Gamma(\cdot)$ an aggregation function (*e.g.,* min, max, conjunction) to merge results returned from d sketches. Moreover, we provide the proofs of error bounds for different queries in Appendix A.

4.1 Edge Queries

Aggregated edge weights. Given two node labels a and b, we denote by $f_e(a, b)$ the exact aggregated edge weight from node a to node b. We write $\tilde{f}_e(a, b)$ the estimated weight.

One application of such queries, taking social networks for example, is to estimate the communication frequency between two specific friends.

It is straightforward to evaluate $\tilde{f}_e(a, b)$, the weight of edge (a, b). It first gets estimated edge weight $\mathbf{M}_i[h_i(a)][h_i(b)]$ from each matrix $\mathbf{M}_i$ and then uses a corresponding function $\Gamma(\cdot)$ to merge them. The notation $\mathbf{M}_i[h_i(a)][h_i(b)]$ is the cell value relative to position $h_i(a)$ and $h_i(b)$ in matrix $\mathbf{M}_i$. It works as follows:

$$\tilde{f}_e(a, b) = \Gamma\big(\mathbf{M}_1[h_1(a)][h_1(b)], \cdots, \mathbf{M}_d[h_d(a)][h_d(b)]\big)$$

In this case, the function $\Gamma(\cdot)$ is to take the minimum.

Complexity. Estimating the aggregate weight of an edge query is in $O(d)$ time, where d is a constant.

4.2 Node Queries

Aggregated node flows. For a directed graph, given a node label a, we denote by $f_v(a, \rightarrow)$ (resp. $f_v(a, \leftarrow)$) node out-flow query (resp. in-flow query), which is to tell the aggregated edge weight *from* (resp. *to*) a node with label a in the graph stream G. For an undirected graph, we write $f_v(a, -)$. Similarly, we use $\tilde{f}_v(a, \rightarrow), \tilde{f}_v(a, \leftarrow), \tilde{f}_v(a, -)$ for estimated results using sketches.

One important application of such queries is to find heavy hitters (*i.e.,* top-k nodes with highest weights of their flows) in *e.g.,* DoS (Denial-of-service) attacks for cyber security, which typically flood a target source (*i.e.,* a computer) with massive external communication requests.

Note that, CountMin has been used to solve the problem of finding heavy hitters, *i.e.,* which nodes are associated with a large number of edges or weights. Again, CountMin is a linear structure, which cannot tell any information more than a count. For a simple extended problem such as conditional heavy hitters [31], which is to find heavy hitters that are *locally* popular by considering edge connections, CountMin does not work. In contrast, the graphical structure of **TCM** provides rich information to find *conditional heavy hitters*. Please refer to Appendix-B.1 for the detailed algorithm for computing conditional heavy hitters using **TCM**.

Here, we only discuss the case $\tilde{f}_v(a, \rightarrow)$, which is to estimate the aggregate weight from node a. The other two cases, *i.e.,* $\tilde{f}_v(a, \leftarrow)$ and $\tilde{f}_v(a, -)$, can be processed analogously.

We denote by $\tilde{f}_v^i(a, \rightarrow)$ the estimated out-flow from the i-th sketch. $\tilde{f}_v^i(a, \rightarrow)$ can be computed by first locating the row in the adjacency matrix corresponding to label a (*i.e.,* $h_i(a)$), and then summing up the values in that row, *i.e.,* $\tilde{f}_v^i(a, \rightarrow) = \sum_{j=1}^{w} \mathbf{M}_i[h_i(a)][j]$. Here, w is the width of the adjacency matrix $\mathbf{M}_i$. Then,

$$\tilde{f}_v(a, \rightarrow) = \Gamma\big(\tilde{f}_v^1(a, \rightarrow), \cdots, \tilde{f}_v^d(a, \rightarrow)\big)$$

The function $\Gamma(\cdot)$ is also to take the minimum.

Complexity. It is easy to see that it takes $O(d \cdot w)$ time, where both d and w are constants.

4.3 Path Queries

Reachability. Given two node a and b, a **boolean** reachability query $r(a, b)$ is to tell whether there exists a path from a to b. Also, we write $\tilde{r}(a, b)$ as the estimated reachability.

One important monitoring task, for the success of multicast deployment in the Internet, is to verify the availability of network service, which is usually referred to as *reachability monitoring*. Another application that needs to consider edge weights is IP routing, which is to determine the path of data flows in order to travel across multiple networks.

We consider the reachability query $\tilde{r}(a, b)$, which is to estimate whether b is reachable from a. For such queries, we treat any *off-the-shelf* algorithm $\mathsf{reach}(x, y)$ as a *black-box*, which returns either *true* to indicate that b is reachable from a, or *false* to represent that b is unreachable from a. It works as follows for our sketch.

P1. [Execute in parallel.] Invoke $\mathsf{reach}(h_i[a], h_i[b])$ on each sketch S_i (for $i \in [1, d]$), to decide whether the mapped node $h_i[b]$ is reachable from the mapped node $h_i[a]$ on sketch S_i.

P2. [Merge.] Merge individual results as follows:

$$\tilde{r}(a, b) = \mathsf{reach}(h_1[a], h_1[b]) \wedge \cdots \wedge \mathsf{reach}(h_d[a], h_d[b])$$

Here, the "$\wedge$" is for a boolean conjunction. Stating in another way, the estimated reachability is *true* only if the mapped nodes are reachable from all d sketches.

The complexity of the above strategy is determined by the third party solution $\mathsf{reach}()$.

4.4 Subgraph Queries

Aggregate subgraph queries. The *aggregate subgraph query* is considered in gSketch [39]. It is to compute the aggregated weight of the constituent edges of a subgraph $Q = \{(x_1, y_1), \cdots, (x_k, y_k)\}$, denoted by $f_g(Q) = \Omega(f_e(x_1, y_1), \cdots, f_e(x_k, y_k))$. Here, the function $\Omega(\cdot)$ is to sum up the weights from all $f_e(x_i, y_i)$ for $i \in [1, k]$. We write $\tilde{f}_g(Q)$ for the estimated result using sketches.

We adopt the following query semantics: if $\tilde{f}_e(x_i, y_i) = 0$ for any edge, the estimated aggregate weight should be 0, since the query graph Q does not have an exact match.

Example 6: Consider a subgraph with two edges as $\{(a, b), (a, c)\}$. The query $Q_3 : \tilde{f}_g(\{(a, b), (a, c)\})$ is to estimate the aggregate weight of the graph. The precise answer is 2, which is easy to verify from Fig. 1. □

Extensions. We consider an extension of the above aggregate subgraph query, which allows a wildcard $*$ in the node labels that are being queried. More specifically, for the subgraph

Symbols	Notations
G, S_G	a graph stream, and a graph sketch
$\omega(e)$	weight of the edge e
$f_e(a,b)$	edge weight
$f_v(a,\rightarrow)$	node out-flow weight (directed graphs)
$f_v(a,\leftarrow)$	node in-flow weight (directed graphs)
$f_v(a,-)$	node flow weight (undirected graphs)
$r(a,b)$	whether b is reachable from a
$f_g(Q)$	weight of subgraph Q

Table 1: Notations

query $Q = \{(x_1, y_1), \cdots, (x_k, y_k)\}$, each x_i or y_i is either a constant value, or a wildcard $*$ (*i.e.,* match any label). A further extension is to *bound* the wildcards to be matched to the same node, by using a subscript to a wildcard as $*_j$. That is, two wildcards with the same subscripts enforce them to be mapped to the same node.

Example 7: A subgraph query $Q_4 : \tilde{f}_g(\{(a,b),(b,c),(c,a)\})$ is to estimate the triangle, *i.e.,* a 3-clique with three vertices labeled as a, b, and c, respectively. Another subgraph query $Q_5 : \tilde{f}_g(\{(*,b),(b,c),(c,*)\})$ is to estimate paths that start at node with an edge to b, and end at any node with an edge from c, if the edge (b,c) exists in the graph. If one wants to count the common neighbors of (b,c), the following query $Q_6 : \tilde{f}_g(\{(*_1,b),(b,c),(c,*_1)\})$ can enforce such a condition, which again is a case of counting triangles. □

The extension is apparently more general, with the purpose of covering more useful queries in practice. Unfortunately, gSketch cannot support such extensions.

We next discuss the aggregate subgraph query $\tilde{f}_g(Q)$, which is to compute the aggregate weight of the constituent edges of a sub-graph Q. The process is similar to the above path queries, by using any existing algorithm subgraph(Q).

S1. [Execute in parallel.] Invoke subgraph(Q) at each sketch to find subgraph matches, and calculate the aggregate weight, denoted by weight$_i(Q)$ obtained from the i-th sketch.

S2. [Merge.] Merge individual results as follows:

$$\tilde{f}_g(Q) = \Gamma\big(\mathsf{weight}_1(Q), \cdots, \mathsf{weight}_d(Q)\big)$$

Here, the function $\Gamma(\cdot)$ is also to take the minimum.

Note that, running a graph algorithm on a sketch is only applicable to **TCM**. It is not applicable to gSketch since gSketch by nature is an array of frequency counts, without maintaining the graphical structure as **TCM** does. Also, in the case that weight$_i(Q)$ from some sketch says that a subgraph match does not exist, we can terminate the whole process, which provides chances for optimization that is discussed below.

Optimization. Consider a subgraph Q is defined as constituent edges as $\{(x_1, y_1), \cdots, (x_k, y_k)\}$. An alternative way of estimating the aggregate subgraph query is to first compute the minimum value among all estimated edge weights. If any $\tilde{f}_e(x_i, y_i)$ $(i \in [1,k])$ is 0, we have $\tilde{f}'_g(Q) = 0$. Otherwise, we sum them up as:

$$\tilde{f}'_g(Q) = \textstyle\sum_{i=1}^{k} \tilde{f}_e(x_i, y_i)$$

Intuitively, instead of taking the minimum weighted subgraph from all sketches, this optimization is to decompose a big problem (*e.g.,* a graph) to small pieces (*e.g.,* an edge) and locally optimize each piece. Naturally, we have

from \ to	$I(af)$	$II(bc)$	$III(dg)$	$IV(e)$
$I(af)$	1	2	0	0
$II(bc)$	3	1	1	1
$III(dg)$	0	1	1	0
$IV(e)$	1	1	1	0

Figure 5: The adjacency matrix of S_1

$\tilde{f}'_g(Q) \leq \tilde{f}_g(Q)$. Consider the two extensions of subgraph queries discussed above. For the first extension that a wildcard $*$ is used, the above optimization can be used. For instance, the edge frequency of $\tilde{f}_e(x,*)$ for a directed graph is indeed $\tilde{f}_v(x,\rightarrow)$. For the second extension that multiple wildcards $*_i$ are used to bound to the same node, this optimization cannot be applied.

Summary of notations. The notations of this paper are summarized in Table 1, which are for both directed and undirected graphs, unless being specified otherwise.

4.5 Wrap-Up

Let's conclude this section by giving some insight.

(1) One can see that we can treat existing algorithms as *black-boxes* to be applied directly on top of our sketches, *e.g.,* reach() in Section 4.3 and subgraph() in Section 4.4.

(2) One can also optimize a specific query by leveraging multiple sketches to take care of some piece of the query, *e.g.,* the edge estimation used in Section 4.4, which provides flexibility and opportunity to tune **TCM** for other specific applications.

(3) Many other graph analytics can be potentially beneficial from **TCM**, beyond what have been presented above. Please refer to Appendix B.1 about an algorithm for conditional heavy hitters, and Appendix B.2 about an algorithm for heavy triangle connections, which are important in community detection.

5. INTERNALS

Now let's shift gears and concentrate on the internals of **TCM** and its extensions (Section 5.1). We also discuss pairwise independent hash functions (Section 5.2), and a discussion of **TCM** in a distributed environment (Section 5.3).

5.1 Adjacency Matrix and Beyond

Using hash-based methods, it is evident that it only requires one-pass of the graph stream to construct/update a **TCM**. However, the linear time complexity of constructing and updating a **TCM** depends on the data structures used. For example, the adjacency list may not be a fit, since searching a specific edge on a list is not in constant time.

5.1.1 Adjacency Matrix

An adjacency matrix is a means of representing which nodes of a graph are adjacent to which other nodes.

Example 8: Consider the sketch S_1 in Fig. 4 (a) for example. Its adjacency matrix is shown in Fig. 5. □

Example 8 showcases a directed graph. In the case of an undirected graph, it will be a symmetric matrix.

Construction. Consider a graph stream $G = \langle e_1, e_2, \cdots, e_m\rangle$ where $e_i = (x_i, y_i; t_i)$. Given a number of nodes w for a graph sketch and a hash function $h(\cdot) \rightarrow [1, w]$. We use the following strategy.

from \ to	$i(abcd)$	$ii(efg)$
$I(a)$	2	0
$II(b)$	3	1
$III(c)$	0	2
$IV(d)$	0	1
$V(e)$	2	1
$VI(f)$	1	0
$VII(g)$	1	0

Figure 6: A non-square matrix

C1. [Initialization.] Construct a $w \times w$ matrix $\mathbf{M}$, with all values being initialized as 0.

C2. [Insertion of edges.] For each edge e_i ($i \in [1, m]$), compute $h(x_i)$ and $h(y_i)$, and increase the value of $\mathbf{M}[h(x_i)][h(y_i)]$ by $\omega(e_i)$, the weight of e_i.

In the above strategy, **C1** takes constant time to allocate a matrix. **C2** takes constant time for each e_i. Hence, the time complexity is $O(m)$ where m is the number of edges in G. The space complexity is $O(w^2)$.

Deletions. [Deletion of e_i.] Insertions have been discussed in the above **C2**. For the deletion of an e_i that is not of interest (*e.g.,* out of a certain time window), it suffices to decrease the value of $\mathbf{M}[h(x_i)][h(y_i)]$ by $\omega(e_i)$ in $O(1)$ time.

Alternatively, one may consider to use an adjacency *hash-list* for each node, instead of a list, to maintain its adjacent nodes using a hash table. Given an edge $e_i(x_i, y_i; t_i)$, two hash operations are needed: The first is to locate x_i, and the second is to find y_i from x_i's hash-list. Afterwards, it updates the corresponding edge weight. adjacency list is known to be suitable when graph is sparse. However, in terms of compressed graph in our case, most sketches are relatively dense, which makes the adjacency matrix the default data structure to manage our graph sketches.

5.1.2 Non-Square Matrices

In this section, we discuss the limitations of using standard adjacency matrices, followed by our proposal to relax them.

Let R be the available space for storing a graph. When using a classical adjacency matrix, we have a $n \times n$ matrix, where $n = \sqrt{R}$. Now, consider all edges from node a such as $(a, *)$. Using any hash function will inevitably hash all of these edges to the same row that is relative to node a in a matrix. For example, in Fig. 5, all edges $(a, *)$ will be hashed to the first row of the matrix.

In fact, for real-world graph data, the node degrees are skewed. It is desirable that, the nodes with higher degrees should be allocated more buckets, so as to reduce the probability of hash collisions. However, due to the lack of *a priori* knowledge of data distribution in highly dynamic graphs, it is hard to decide the right shape of a matrix. To cope with this problem, we propose to use non-square matrices.

Non-square matrices. We use a $p*q$ matrix with two hash functions: $h_1(\cdot) \to [1, p]$ on the *from* nodes and $h_2(\cdot) \to [1, q]$ on the *to* nodes. Let $p*q \approx R$ where R is the available space.

Example 9: Consider the graph stream in Fig.1. Assume that we use two hash functions: $h_1(\cdot) \to [1, 7]$ and $h_2(\cdot) \to [1, 2]$. The non-square matrix is shown in Fig. 6. □

In practice, when generating multiple sketches, we simply use matrices $n * n$, $2n * n/2$, $n/2 * 2n$, $4n * n/4$, $n/4 * 4n$, etc, without any assumption of data distribution.

5.1.3 Data Stream Sketches: A Bird's Eye View

Using non-square matrices, we can see clearly the picture of data stream summarization from a border perspective. Keep in mind that a sketch for data streams should be in *sublinear* space, *linear* construction time, and *constant* time maintenance cost per update.

One-dimensional sketch. Traditional sketches typically use a linear structure to maintain a certain type of characteristics of data streams, in real time. Taking approximate frequency counts [29] for example, they are special cases of **TCM** when setting one of our hash functions with only one bucket, *i.e.,* $h_2(\cdot) \to [1, 1]$ (see Section 5.1.2 for details).

Two-dimensional sketch. When handling two-dimensional data such as graph streams (see Section 3.1 for our formal definition) where we need to maintain the information between two items in one element, we propose to extend the sketch from a one-dimensional structure (*e.g.,* a vector) to a two-dimensional structure (*e.g.,* a matrix). The increased power with this small change is evident (see Section 4).

High-dimensional sketch. See further: When dealing with data streams that for each element $(v_1, v_2, \ldots, v_x)$, there are x intra-connected values that need to be maintained. It is natural to extend our idea to use x independent methods, $m_1, \cdots, m_x$, with each m_i working for one dimension. Here, m_i could be either a hash function, a sample-based method, or other methods such as predefined hash tags, *e.g.,* different application protocols (*e.g.,* TCP or UDP) for the Internet, or different years as a time dimension.

5.1.4 Hash Key Materialization

Obviously, there are many ways to extend **TCM**. One intuitive way is to materialize the node labels. It comes from the following observation: A hash function is a *one-way* function that is easy to compute, but hard to invert. Therefore, we hit a wall between hash keys and hash values such that it is difficult for us to do more reasonings given only hashed values. Especially when using d hash functions, we lose important connections among them since we do not keep the information of hash keys. This naturally leads to the idea of maintaining hash keys.

Extended graph sketch. Consider the graph sketch $S_G(\mathcal{V}, \mathcal{E})$ defined in Section 3.3. An *extended graph sketch* is a graph sketch $S_G(\mathcal{V}, \mathcal{E})$ that each node $v \in \mathcal{V}$ is extended with a set of node labels, denoted by $\mathsf{ext}(v)$, which records all node labels that are hashed to v.

Take Fig. 3 for example, we have $\mathsf{ext}(I) = \{a, e\}$. In fact, in Fig. 3, although we depict that with each node, the labels that it represents, *e.g.,* node I for label a and e, are not explicitly stored in the original graph sketch, but in the extended graph sketch.

Of course, this benefit comes at a cost, the extra node labels. However, this takes only $O(|V|)$ space where $|V|$ is the number of nodes in the graph stream. In terms of graph streams, the graph model is typically a multigraph with multiple edges between two nodes. Under such circumstance, the number of nodes is typically much smaller than the number of edges and maintaining them is affordable.

We will show later the usage of the extended graph sketch

in supporting the problem of heavy triangle connections (see Appendix B.2).

5.2 Pairwise Independent Hash Functions

Here, we only borrow the definition of pairwise independent hash functions.

A family of functions $\mathbf{H} = \{h|h(\cdot) \rightarrow [1, w]\}$ is called a *family of pairwise independent hash functions* if for two different hash keys x_i, x_j, and $k, l \in [1, w]$,

$$\mathsf{Pr}_{h \leftarrow \mathbf{H}}[h(x_i) = k \wedge h(x_j) = l] = 1/w^2$$

Intuitively, when using multiple hash functions to construct sketches, the hash functions used should be pairwise independent in order to reduce the probability of hash collisions. Please refer to [14] for more details.

5.3 Distributed Setup

One can see that **TCM** can be naturally deployed to a distributed environment, since the construction and maintenance of each sketch is independent of each other. Assuming we have d sketches in one computing node, when m computing nodes are available, we can use $d \times m$ sketches, which can significantly reduce the probability of hash collisions. Moreover, many analytics using **TCM** can be independently performed on multiple sketches in parallel (see Section 4), which makes a distributed environment a good fit for **TCM**.

6. EXPERIMENTS

We will start by explaining the environments where our experiments were conducted (Section 6.1). We then compare with existing approaches (Section 6.2). Finally, we summarize our experimental findings (Section 6.3).

The chief purpose of our experimental study is to prove that **TCM** should serve as the new backbone for graph stream management: **TCM** is as good as *ad-hoc* sketches for specific problems, but is much more general in supporting a wide range of analytics not supported by existing sketches.

6.1 Experimental Setup

6.1.1 Data Sets and Environment Setup

❶ ***DBLP.*** We extracted 933,530 authors as nodes and 4,918,090 author-pairs as edges from the latest DBLP archive[4]. It is an undirected graph with the weight of each streaming edge to be 1, indicating a co-authorship.

❷ ***IP flow.*** This data set contains anonymized passive traffic traces from CAIDA's equinix-chicago monitor on high-speed Internet backbone links[5]. We use one minute's traces that contain 281121 IPs and 17,721,707 traces. A trace represents a directed edge that one IP sent a packet to another IP, such as $(192.168.29.1, 192.168.29.133, 62, 105.12)$. Here, each IP address is a node, each trace represents a directed edge, and the edge weight is the packet size, *e.g.,* 62 bytes.

❸ ***GTGraph.*** We used the well-known graph generator GTGraph[6] to generate directed graphs. We first used the R-MAT model [11] to generate a large network with power-law degree distributions. We then added weights to the edges using Zipfian distribution and the weight for each edge means the times the edge appeared in the streams. The generated network contains 10^7 vertices and $1.444 * 10^9$ edges.

❹ ***Twitter link structure.*** We used the Twitter link structure Twitter[7]. Since the released data is anonymized, we mainly used it for efficiency study. It has around 5.26 million nodes and 2 billion undirected edges, and we added random edges to simulate friendship links to make it 10 billion edges, which is 200GB on disk.

Setup. All algorithms were implemented in C++. We used the same hash functions as adopted in an open source CountMin code[8]. All our experiments were conducted on an Intel PC with a 3.4 GHz CPU and 32GB RAM, running Ubuntu.

6.1.2 Comparison with State-of-the-Art

In sketches for data streams, there are two main types: frequency counts based (*e.g.,* CountMin and gSketch) and sample-based [29]. Take frequency counts based approaches for example, one needs to implement a node sketch by hashing nodes to support node queries, and another edge sketch by hashing edges to support edge queries (see Example 1 for more details). It is the same for sample-based solutions that different sketches are needed for *ad-hoc* problems.

We used two strategies to compare with other approaches.

(1) *Same space for an ad-hoc problem.* We used the same space for **TCM**, frequency counts-based, and sample-based approaches, for each specific problem.

(2) *Same space for a set of problems.* When giving a set of problems, *e.g.,* both edge and node queries, we only build one **TCM**, while using the same space to constructing two frequency counts-based sketches, a node sketch and an edge sketch. To better understand this comparison, please refer to Table 3 in Appendix C.1 to see the types of queries that **TCM** supports and what existing solutions support.

Note that, frequency counts-based approaches such as CountMin and **TCM** overcount the encountered elements. While sample-based approaches, on the other hand, typically undercount the elements. In fact, sample-based approaches, from the application perspective, are mainly used to estimate the *hot elements*, *i.e.,* heavy hitter problems.

6.1.3 Effectiveness Metrics

Average relative error. This measure is used for the set of queries that returns estimated frequencies, such as edge or subgraph frequencies. We used the measure defined in [39]. Given a query Q, the *relative error* is formalized as:

$$\mathsf{er}(Q) = \frac{\tilde{f}'(Q) - f(Q)}{f(Q)} = \frac{\tilde{f}'(Q)}{f(Q)} - 1$$

The *average relative error*, given a set of m queries as $\mathcal{Q} : \{Q_1, \ldots, Q_m\}$, is determined by averaging the relative errors over all queries Q_i for $i \in [1, m]$ as:

$$\mathsf{e}(\mathcal{Q}) = \frac{\sum_{i=1}^{k} \mathsf{er}(Q_i)}{m}$$

Intersection accuracy. When evaluating top-k results for heavy nodes/edges and reachability queries, we used a simple *intersection metric* [17]. Let X be the set of top-k results computed by an algorithm, and Y be the ground truth top-k

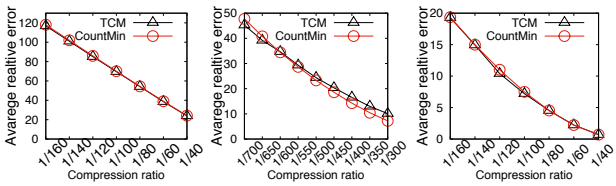
[4] http://dblp.dagstuhl.de/xml/

[5] http://www.caida.org/data/passive/passive_2015_dataset.xml

[6] http://www.cse.psu.edu/~kxm85/software/GTgraph/

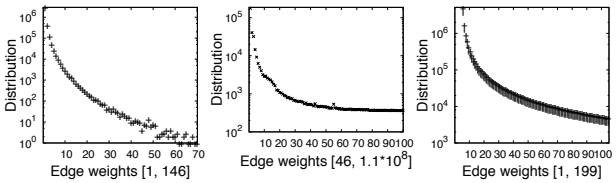
[7] http://twitter.mpi-sws.org/data-icwsm2010.html.

[8] https://github.com/alabid/countminsketch/



(a) DBLP (b) IP flow (c) GTGraph

Figure 7: Edge queries (varying compression ratios)



(a) DBLP (b) IP flow (c) GTGraph

Figure 8: Edge weight distribution

results. The inter-accuracy is defined as $|X \cap Y|/k$, which is in $[0, 1]$, where 1 means that all real top-k results are found.

Furthermore, we used another popular measure of ranking quality for top-k results, namely normalized discounted cumulative gain (NDCG), see Appendix C.3 for more details.

6.2 Experimental Results

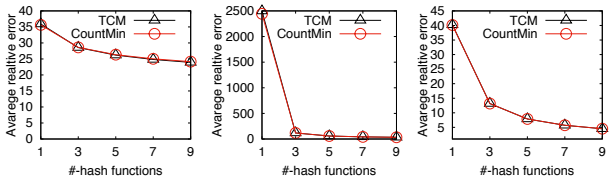
In this section, we show the results from five types of experiments: effectiveness for edge queries (Exp-1), effectiveness for node queries (Exp-2), effectiveness for path queries (Exp-3), effectiveness for graph queries (Exp-4), and efficiency study (Exp-5).

Exp-1: Edge queries. In the first set of experiments, we studied the performance of **TCM** in estimating edge frequencies by varying different parameters. More specifically, we varied d that is the number of hash functions, and w that is the width of the matrix.

(a) [Fixed $d = 9$, and varying w.] Here, we fixed the number of hash functions d to be 9. We varied the sizes of **TCM** by different compression ratios. Consider a graph with $|E|$ edges, the compression ratio c means that we use $|E| \times c$ space for storage. Take DBLP with $2,3425,800$ edges for example, the compression ratio $1/100$ indicates that the matrix used takes $2,3425,800 \times 1/100 = 234,258$ space, which is a $\sqrt{234,258} \times \sqrt{234,258}$ (*i.e.,* 484×484, or $w = 484$) matrix. For each CountMin sketch we use a vector with $234,258$ cells, the same size of the matrix used by **TCM**. Figures 7(a), 7(b) and 7(c) show the results of edge frequency estimation (Section 4.1) for the data sets DBLP, IP flow, and GTGraph, respectively. The x-axis is the compression ratio and the y-axis is the average relative error of distinct edges in the graph streams.

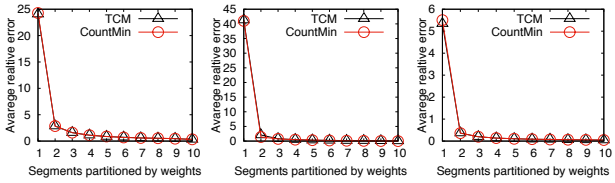
The above results show that **TCM** and CountMin achieve comparable performance. The reason is that both approaches hashed (or compressed) all values to the same spaces having the same size $w \times w$, and the error bounds are the same (see Section A.1).

One may observe that the average relative error is relatively high for all data sets. The reason is that both **TCM** and CountMin overcount stream elements due to the compressed space. For an edge e, if its real frequency $f_e(e)$ is low, its estimated frequency $\tilde{f}(e)$ is typically much higher



(a) DBLP (b) IP flow (c) GTGraph

Figure 9: Edge queries (fixed w, varying d)



(a) DBLP (b) IP flow (c) GTGraph

Figure 10: In-depth: partition edge weights

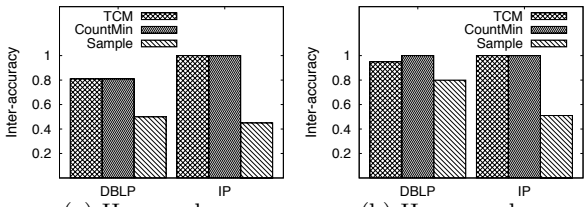
than $f_e(e)$ since it has chances to collide with edge e' whose real frequency is much higher than e. Figures 8(a), 8(b), and 8(c) show the edge weight distribution for DBLP, IP flow, and GTGraph, respectively. For each data set, the x-label indicates the range of edge weights. For instance, the edge weights, *i.e.,* the number of co-authorships between two authors for DBLP in Fig. 8(a), are in $[1, 146]$. There are 70 distinct weights, which are ranked in ascending order along the x-axis, and the y-axis indicates the number that such frequency appears. In Fig. 8(b), the edge weights have a big range $[46, 1.1 \times 10^8]$ that for the aggregated bytes transferred between two IPs. We sorted the weights in ascending order, partitioned the sorted weights in 100 equal sized bucket, and counted the number of edges in each bucket. For the value 10 in the x-axis, its y-axis value corresponds to the total number of edges appeared in the first 10 buckets. Similar for the GTGraph data in Fig. 8(c).

These figures tell us that the edge frequencies of all data sets satisfy the Zipf distribution, and edges with low frequencies dominate the results in Fig. 9. However, this will not hurt a lot in real applications. As will be shown later, the average relative error is very low for edges with high frequencies. Also, we will show that in important applications like heavy hitters, these sketches will find top-k frequent edges with high accuracy.

(b) [Fixed w, and varying d from 1 to 9.] In this set of experiments, we studied the effect of multiple pairwise independent hash functions. Figures 9(a), 9(b), and 9(c) show the results of average relative error for data sets DBLP, IP flow and GTGraph, by fixing the compression ratios to $1/40$, $1/600$ and $1/80$, respectively. In the x-axis, we varied the number of hash functions d from 1 to 9 with a step of 2.

These results tell us that using multiple pairwise independent hash functions can indeed reduce the average relative error, since the probability of hash collisions is reduced significantly. The effect is particularly clear when the edge weights are large numbers, *e.g.,* the bytes of sent packets in IP flow data in Fig. 9(b), for both **TCM** and CountMin.

(c) [In-depth: partition edge weights.] We tested the average relative error for different ranges of edge weights, as a complementary experiment for Exp-1 (a)(b). For each data set, we first sorted all edges in ascending weight order, and partitioned the sorted edge weights in 10 groups of equal size.



(a) Heavy edges (b) Heavy nodes

Figure 11: Heavy hitters

	$d=1$	$d=3$	$d=5$	$d=7$	$d=9$
CountMin	2434	124	62	44	36
TCM	2551	122	63	45	36
gSketch	31	13	10	9.3	8.6
TCM (edge sample)	33	13	11	9.9	9.2

Table 2: Average relative errors (IP flow)

Figures 10(a), 10(b) and 10(c) show the results for data sets DBLP, IP flow, and GTGraph, respectively. Here, we fixed the number of hash functions $d = 9$, and the compression ratios 1/40, 1/600 and 1/80 for the data sets DBLP, IP flow, and GTGraph, respectively.

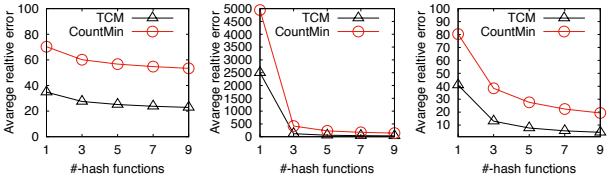
These figures show consistent results that the average relative error is high for edges with low weights, *e.g.,* the 10% lowest weight edges *w.r.t.* x-axis value 1 in all figures. Along with the increasing edge weights, *i.e.,* the x-axis values from 2 to 10, the average relative error is significantly reduced.

(d) [Heavy edges.] We studied the performance of **TCM**, CountMin and sample-based approaches for estimating heavy edges. For each data set, we first computed top-100 edges from the original data as the ground truth. For monitoring heavy edges, we adopted a priority queue to maintain the estimated top-100 heavy edges, for each sketch. The results from the priority queue were then computed for the inter-accuracy as defined in Section 6.1.3. Also, we fixed the number of hash functions $d = 9$ and the compression ratios 1/40 and 1/600 for DBLP and IP flow, respectively. Figure 11(a) shows the results for DBLP and IP flow. GTGraph is a synthetic data, and hence its heavy edges are less of interests, which are thus not shown here. Moreover, we used the non-square matrices (Section 5.1.2) for this experiment, without assumption of the presence of any sample data. The results of using square matrices are a little bit worse than using non-square matrices, since non-square metrics can heuristically reduce the probability of hash collisions, as remarked earlier in Section 5.1.2.

The result in Fig. 11(a) tells us that **TCM** and CountMin are perfect (inter-accuracy = 1) in finding heavy edges for IP flow, for the case that the edge weights have a big range $[46, 1.1 \times 10^8]$ to represent the bytes sent between two IPs. For the edge weights that have a smaller range, *e.g.,* $[1, 146]$ for DBLP, the inter-accuracy of both **TCM** and CountMin reduced, but are still good that is 80%. For sample-based approach, we use a uniform sampling to sample a large portion (50%) of data. For both data sets, sample-based approaches perform worse than the other two.

(e) [Compare with gSketch.] We have implemented gSketch, which uses data samples to partition the graph stream by putting edges with similar weights in one partition, such that low weight edges will not collide with high weight edges when being hashed.

The result for IP flow is given in Table 2, by varying the number d of hash functions. Here, we used 10 data partitions. The result shows that the techniques proposed by gSketch can be easily integrated to **TCM**, *i.e.,* **TCM** (edge sample) in Table 2, to significantly reduce the average relative errors, when data samples are available. Moreover, the average relative errors of gSketch and improved **TCM** using edge samples are very close. The results for other data sets are given in Appendix C.2, which show similar trend.



(a) DBLP (b) IP flow (c) GTGraph

Figure 12: Edge queries (sketches for > 1 function)

Figure 13: Conditional heavy hitters (DBLP)

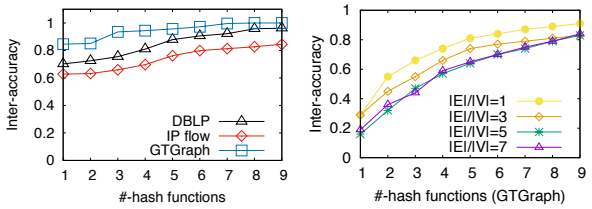
(f) [Same space for a set of problems.] As stated in Section 6.1.2, using the same space, we built one **TCM** (with compression ratio 1/600), but two CountMin based sketches, one for edge and the other for node with equal size. The result of comparing **TCM** and edge sketches in this setting is given in Fig. 12. The result for node comparison is similar and thus omitted due to space constraints.

The result shows that **TCM** clearly outperforms CountMin in this setting. That is, by using the same space for a set of problems, **TCM** can support *ad-hoc* problems with better accuracy. Meanwhile, **TCM** can still support more analytics (see more discussion in Table 3 in Appendix C.1).

Exp-2: Node queries. In this set of experiments, we evaluated the support for heavy nodes of different sketches, as well as showing the results of using **TCM** for the conditional heavy hitter problem.

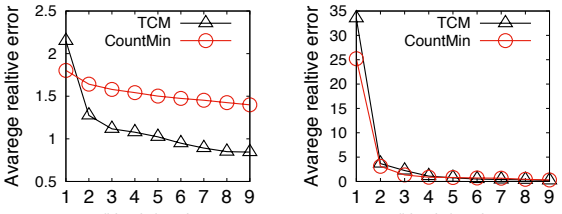
(a) [Heavy nodes.] We first studied the monitoring task of finding top-k heavy nodes. Note that, we adopted the same **TCM** sketches used in Exp-1 (d). However, for CountMin and sample-based approaches, they need to rebuild sketches for collecting node information. In other words, in order to support edge queries and point queries, both CountMin and sample-based approaches need to use extra spaces. For CountMin, we used the same space for **TCM**. For sample-based approaches, we used 50% rate for uniform sampling. For each data set, we first computed top-100 nodes from the original data as the ground truth. Similar to monitoring heavy edges, we used a priority queue to maintain the estimated top-100 heavy nodes, for each sketch. The results from the priority queue were then computed for the inter-accuracy as defined in Section 6.1.3. Figure 11(b) shows the results for DBLP and IP flow. Also, we ignored GTGraph, since it is a synthetic data, which is less of interests for heavy hitter problems.

The results show that **TCM** and CountMin perform similar with good accuracy in terms of estimating heavy nodes.



(a) All Data Sets (b) True Negatives

Figure 14: Reachability queries (varying d)



(a) DBLP (b) IP flow

Figure 15: Subgraph frequencies (varying d)

In particular, they clearly outperform sample-based approaches in both data sets, which agrees with the results from heavy edges given in Fig. 11(a).

(b) [Conditional heavy hitters.] Besides finding which nodes are most popular as heavy hitters do, conditional heavy hitters is to further find the most popular neighbors to the most popular nodes. Please find a detailed discussion and its corresponding Algorithm 1 in Appendex-B.1 for detecting conditional heavy hitters. Also, please refer to the discussion in Section 4 about why other sketches fall short of supporting the problem of conditional heavy hitters.

Since the results of DBLP are easy to understand, we only show the conditional heavy hitters detected for DBLP, as given in Fig. 13. It depicts the top-5 most productive authors such as H. Vincent Poor and Wen Gao, each of them indeed has > 700 publications. Also, it reports for each author and his top-5 most frequent collaborators. We have manually checked DBLP: 3 collaborators (Shlomo Shamai, Sanjeev R. Kulkarni, and Yingbin Liang) are indeed in H. Vincent Poor's top-5 frequent collaborators, and the other 2 are in his top-10 frequent collaborators.

Exp-3: Path queries. In this set of experiments, we evaluated the power of **TCM** in supporting reachability queries. Note that the other two types of sketches, CountMin and sample-based, cannot be used for estimating reachability queries. We fixed the compression ratios to 1/40, 1/600 and 1/80 for DBLP, IP flow, and GTGraph, respectively. We varied the number of hash functions from 1 to 9. For each data set, we randomly picked 100 pairs of nodes. The estimation is correct if either **TCM** reports that two nodes are reachable and they are indeed reachable (*i.e.,* true positives), or estimates that two nodes are not reachable and they are not connected in the original graph (*i.e.,* true negatives). The results for inter-accuracy are shown in Fig. 14(a).

The above results tell us two things. Firstly, along with the increasing number of hash functions d, the inter-accuracy of estimating reachability queries will increase, as expected. Secondly, **TCM** achieved good inter-accuracy for all data sets. For example, when $d = 9$, the inter-accuracy values for DBLP, IP flow and GTGraph are 96%, 84.5% and 100%, respectively.

In-depth: true negatives. Since **TCM** keeps all connectives of the graph stream, it only has *true positives* but no *false positives*, *i.e.,* all reachable node pairs will be detected as reachable. However, due to hash collisions, there may exist hash collisions such that **TCM** may have *false negatives*, *i.e.,* not reachable node pairs may be reported as reachable. In order to control the graph shape, we used synthetic data GTGraph by varying the ratio $|E|/|V|$ from 1 to 7, with a step of 2. Figure 14(b) shows the results for 100 not reachable pairs. It shows that when the number of hash functions is small, the inter-accuracy of *true negatives* is low. Along with increasing number of hash functions, the inter-accuracy increases significantly, with low false negatives. In fact, when $|E|/|V| > 7$, most node pairs were reachable to each other, *i.e.,* no negatives and **TCM** naturally performs well.

Exp-4: Graph analytics. In this set of experiments, we study two types of graph analytics, subgraph queries and heavy triangle connections.

(a) [Subgraph queries.] We first tested the performance of both **TCM** and CountMin in estimating subgraph queries (Section 4.4). Since subgraph queries are considered as summing up the estimated edge frequencies of all graph edges, the results are expected to be similar to the ones for edge queries. We verified the above observation by the following experiments, using only real-world data sets, *i.e.,* DBLP and IP flow. For each data set, we generated, from the original graph stream, 20 connected graphs with different shapes, such as paths, star-shaped graphs, and general graphs. Also, they have various sizes from 2 to 8 edges. We fixed the compression ratios to 1/40 and 1/600 for DBLP and IP flow, and varied the number of hash functions d from 1 to 9. Figures 15(a) and 15(b) show the results of average relative error for DBLP and IP flow, respectively. Here, the x-axis indicates the number of hash functions used, and the y-axis represents the average relative error for all tested graph queries.

The above results tell us that along with the increasing number of hash functions, the average relative error will decrease, which is expected and agrees with the results for edge queries (Exp-1 (b)). Also, the results show that the average relative error is lower than the results in Fig. 9 for edge queries. The reason is that, given a graph with several edges, if the real weight of one edge is large, it will potentially dominate the estimation of average relative error for other edges, and the overall value is thus lower. This also agrees with the results shown in Fig. 10.

(b) [Heavy triangle connections.] Intuitively, heavy triangle connections (see Appendix B.2 for a detailed discussion and its corresponding algorithm) are the set of analytics that first identify heavy edges, and then for each heavy edge (x, y), finding the nodes that communicate frequently with both x and y. We showcase DBLP here, since it is easy to understand by database researchers. We first identified top-20 heavy edges, indicating frequent collaborations, from which we selected five edges about database researchers as depicted in Fig. 16. The top-5 heavy connections with both authors are also shown. Take Charu C. Aggarwal and Philip S. Yu for example, for the 5 discovered authors that collaborate

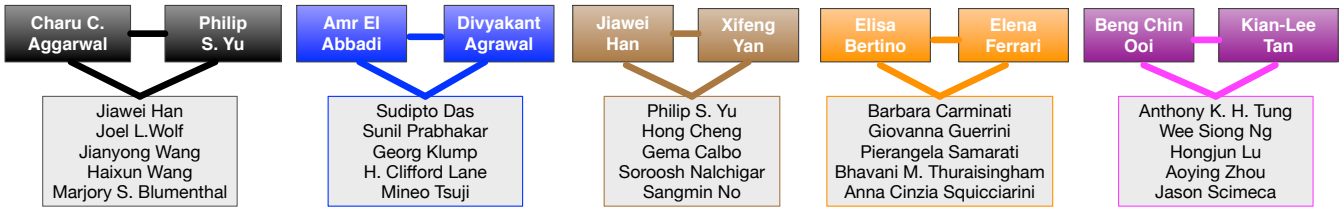
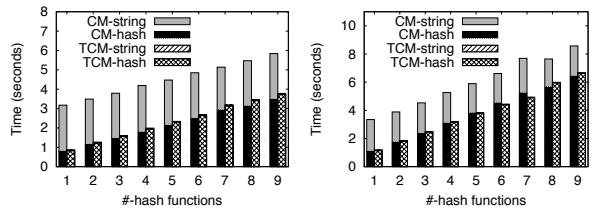
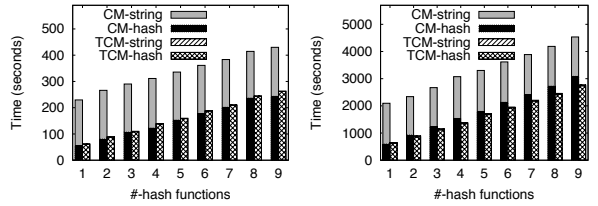


Figure 16: Heavy Triangle Connections (DBLP)



(a) DBLP ($|E| = 4.9 * 10^6$) (b) IP flow ($|E| = 1.7 * 10^7$)



(c) GTGraph ($|E| = 1.44 * 10^9$) (d) Twitter ($|E| = 10^{10}$)

Figure 17: Efficiency

frequently with both of them, 4 are in the ground truth of top-5 results. This tells us that (i) **TCM** can discover interesting and useful results for practical applications; and (ii) the accuracy is good in practice.

Exp-5: Efficiency. In the last set of experiments, we first evaluated the efficiency of building **TCM**, as discussed in Section 5.1.1. The results of varying the number of hash functions from 1 to 9 are given in Fig. 17. In all streaming scenarios, data does not reside in disk. Hence, in our experiments, we counted the time after data is in memory.

Take DBLP in Fig. 17(a) for example. When the x-axis value is 1 that is for 1 hash function, the left (resp. right) bar represents gSketch, *i.e.,* CountMin based edge sketch, (resp. **TCM**). Each bar consists of two parts, where the upper part (-string) is for the time of string operation and the lower part (-hash) is for hashing and updating the sketch. gSketch concatenates two strings but **TCM** does not need to, that is why the upper part of CountMin takes much time but **TCM** has almost zero cost. For the cost of hashing, *i.e.,* the lower parts of both bars, the time is comparable. Moreover, when concatenating two strings a and b, sometimes the time of running the same hash function h on $h(ab)$ takes longer than on two strings $h(a)$ and $h(b)$ separately, which explains why in Fig. 17(d) the hashing time of **TCM** is even less than CountMin. Hence, the overall time of constructing gSketch is even longer than **TCM**.

The above results show that the maintenance cost of **TCM** is comparable to the state-of-the-art and also scalable, which indicates that **TCM** can be efficiently built to support real-time applications in graph streams.

Query time. After effectiveness study in Exps 1-4, we now discuss the efficiency of using **TCM** against query evaluation using the original graph stream. Note that, the **TCM** is stored as an adjacency matrix. The original graph stream, however, can only be stored using an adjacency list for two reasons: (1) one cannot know the number of nodes *a-priori*, and (2) memory limitation. Naturally, query estimation using **TCM** is orders of magnitude faster than evaluation over the graph stream. Please see Appendix C.4 for more empirical results.

6.3 Summary of Experimental Findings

We find the followings from our experimental study.

(1) **TCM** can achieve performance comparable with the state-of-the-art specialized sketches (Exp-1 (a-d), Exp-2 (a) and Exp-4 (a)). Also, the idea of gSketch can be applied to **TCM** to improve accuracy (Exp-1 (e)). Moreover, when using the same space to support different applications, **TCM** is more effective than its counterpart (Exp-1 (f)).

(2) **TCM** is much general than existing sketches, such as the conditional heavy hitters studied in Exp-2 (b), the reachability queries discussed in Exp-3, and the heavy triangle relations studied in Exp-4 (b).

(3) **TCM** is economic to maintain and efficient for query estimation (Exp-5).

These experimental results confirmed the generalization, effectiveness and efficiency of **TCM** in supporting various graph stream applications, which sheds new light on summarizing and analyzing graph streams.

7. CONCLUSION AND FUTURE WORK

We have proposed a graphical sketch **TCM** for summarizing graph streams in a sublinear space, using linear construction time, and with constant maintenance cost per update. We have demonstrated its wide applicability to many emerging applications, theoretically and experimentally. In addition, we have shown that **TCM** has comparable performance for specialized sketches for *ad-hoc* problems, and is more effective when considering a set of problems. We have also shown problems that existing sketches fail to support.

One topic for future work is to revise it in order to serve specific applications, *e.g.,* to integrate it to OpenSOC for cyber security framework. Another topic is, instead of treating it as a sketch, we plan to store extra information, use it as a filter for general (exact) query evaluation (see Section 5.1.4 for an initial discussion). Also, since evaluating queries over multiple sketches of **TCM** is naturally parallelizable, we plan to implement it on a distributed platform *e.g.,* GraphX, so as to handle big graph streams in practice, in a more scalable way. Last but not least, we plan to use it for revisiting a set of graph mining problems, *e.g.,* finding the evolution of graphs, and monitoring networks using temporal snapshots of our sketches.

8. REFERENCES

[1] Tweet statistics. *http://expandedramblings.com/index.php/march-2013-by-the-numbers-a-few-amazing-twitter-stats/10/*.

[2] Vitria. *http://www.vitria.com/solutions/streaming-big-data-analytics/benefits/*.

[3] M. Adler and M. Mitzenmacher. Towards compressing web graphs. In *Data Compression Conference*, pages 203–212, 2001.

[4] C. C. Aggarwal, editor. *Data Classification: Algorithms and Applications*. CRC Press, 2014.

[5] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[6] J. D. Batson, D. A. Spielman, and N. Srivastava. Twice-ramanujan sparsifiers. In *STOC*, pages 255–262, 2009.

[7] A. A. Benczúr and D. R. Karger. Approximating s-t minimum cuts in $\tilde{O}(n^2)$ time. In *STOC*, pages 255–262, 1996.

[8] A. A. Benczúr and D. R. Karger. Randomized approximation schemes for cuts and flows in capacitated graphs. *CoRR*, cs.DS/0207078, 2002.

[9] V. Braverman, R. Ostrovsky, and D. Vilenchik. How hard is counting triangles in the streaming model? In *ICALP*, pages 244–254, 2013.

[10] A. Z. Broder and M. Mitzenmacher. Survey: Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2003.

[11] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SDM*, pages 442–446, 2004.

[12] S. Choudhury, L. B. Holder, G. C. Jr., K. Agarwal, and J. Feo. A selectivity based approach to continuous pattern detection in streaming graphs. In *EDBT*, pages 157–168, 2015.

[13] E. Cohen and H. Kaplan. Tighter estimation using bottom k sketches. *PVLDB*, 1(1):213–224, 2008.

[14] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[15] G. Cormode and S. Muthukrishnan. Space efficient mining of multigraph streams. In *PODS*, pages 271–282, 2005.

[16] M. Elkin. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. *ACM Transactions on Algorithms*, 7(2):20, 2011.

[17] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. In *SODA*, pages 28–36, 2003.

[18] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD*, pages 157–168, 2012.

[19] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2-3), 2005.

[20] J. Gao, C. Zhou, J. Zhou, and J. X. Yu. Continuous pattern detection over billion-edge graph using distributed framework. In *ICDE*, pages 556–567, 2014.

[21] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[22] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[23] S. Guha and A. McGregor. Graph synopses, sketches, and streams: A survey. *PVLDB*, 5(12):2030–2031, 2012.

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

[25] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Trans. Inf. Syst.*, 20(4), 2002.

[26] J. A. Kelner and A. Levin. Spectral sparsification in the semi-streaming setting. *Theory Comput. Syst.*, 53(2):243–262, 2013.

[27] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[28] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[29] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, pages 346–357, 2002.

[30] A. McGregor. Graph stream algorithms: a survey. *SIGMOD Record*, 43(1):9–20, 2014.

[31] K. Mirylenka, G. Cormode, T. Palpanas, and D. Srivastava. Conditional heavy hitters: detecting interesting correlations in data streams. *VLDB J.*, 24(3):395–414, 2015.

[32] S. Raghavan and H. Garcia-Molina. Representing web graphs. In *ICDE*, pages 405–416, 2003.

[33] C. Song, T. Ge, C. X. Chen, and J. Wang. Event pattern matching over graph streams. *PVLDB*, 8(4):413–424, 2014.

[34] D. A. Spielman and N. Srivastava. Graph sparsification by effective resistances. In *STOC*, pages 563–568, 2008.

[35] T. Suel and J. Yuan. Compressing the graph structure of the web. In *Data Compression Conference*, pages 213–222, 2001.

[36] R. E. Tarjan. Data structures and network algorithms. In *SIAM*. 1983.

[37] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. DOULION: counting triangles in massive graphs with a coin. In *SIGKDD*, pages 837–846, 2009.

[38] C. Wang and L. Chen. Continuous subgraph pattern search over graph streams. In *ICDE*, pages 393–404, 2009.

[39] P. Zhao, C. C. Aggarwal, and M. Wang. gSketch: On query estimation in graph streams. *PVLDB*, 5(3):193–204, 2011.

APPENDIX

A. ERROR BOUNDS

We discuss the error bounds for two basic types of queries:

edge queries and node queries. Although **TCM** is more general, we show that theoretically, it has the same error bounds as CountMin.

A.1 Edge Queries

Our proof for error bound of edge queries is an adaption of the proof used in CountMin (Section 4.1 of [14]).

Theorem 1: *The estimation $\tilde{f}_e(x,y)$ of the cumulative edge weight of the edge (x,y) has the following guarantees, $f_e(x,y) \leq \tilde{f}_e(x,y)$ with probability at least $1-\delta$, s.t., $f_e(x,y) \leq \tilde{f}_e(x,y) + \epsilon * n$, where $f_e(x,y)$ is the exact answer to the cumulative edge weight of the edge (x,y), n denotes the number of nodes, and the error in answering a query is within a factor of ϵ with probability δ*[9]. □

PROOF. Consider an edge query $f_e(x,y)$. For each stream edge $e:(x,y;t)$ and a hash function h_i, by construction, the edge weight $\omega(e)$ is added to $\mathcal{M}_i[h_i(x), h_i(y)]$, where $\mathcal{M}_i$ is the adjacency matrix corresponding to the hash function, h_i. Therefore, by construction, the answer to $f_e(x,y)$ is less than or equal to $\mathsf{min}_{i\in[1,d]}\ \mathcal{M}_i[h_i(x), h_i(y)]$, where d is the number of applied hash functions.

Consider two edges (x,y) and (z,w). We define an indicator vector $\vec{I}_{x,y,z,w}$ with d variables $[I^1_{x,y,z,w}, \ldots, I^d_{x,y,z,w}]$, where $I^i_{x,y,z,w}$ corresponds to the hash function h_i. We define $I^i_{x,y,z,w}$ to be 1, if there is a collision between two disjoint edges, *i.e.,* $(x \neq z) \wedge (y \neq w) \wedge (h_i(x) = h_i(z)) \wedge (h_i(y) = h_i(w))$, and 0 otherwise. By independence assumption of hash buckets *w.r.t.* distinct hash keys, we have

$$E(I^i_{x,y,z,w}) = \mathsf{Pr}[(h_i(x) = h_i(z) \wedge h_i(y) = h_i(w)] \leq (1/\mathsf{range}(h_i))^2 = \epsilon'^2/\mathsf{e}^2$$

where e is the base used to compute natural logarithm, and $\mathsf{range}(h_i)$ is the number of hash buckets of function h_i. Define the variable $X^i_{x,y}$ (random over choices of h_i) to count the number of collisions with the edge (x,y), which is formalized as $X^i_{x,y} = \Sigma_{z=1\ldots n, w=1\ldots n} I^i_{x,y,z,w} a_{z,w}$, where n is the number of nodes in the graph and $a_{z,w}$ is the sum of non-zero weight entries of edge (z,w). Since $a_{z,w}$ is non-negative, $X^i_{x,y}$ is non-negative. By construction, $\mathcal{M}_i[h_i(x), h_i(y)] = f_e(x,y) + X^i_{x,y}$. So clearly, $\big(\mathsf{min}_{i\in[1,d]}\ \mathcal{M}_i[h_i(x), h_i(y)]\big) \geq f_e(x,y)$. By pairwise independence of h_i, and linearity of expectation, we have

$$E(X^i_{x,y}) = E\big(\textstyle\sum_{z=1\ldots n, w=1\ldots n} I^i_{x,y,z,w} f_e(z,w)\big) \leq \big(\textstyle\sum_{z=1\ldots n, w=1\ldots n} E(I^i_{x,y,z,w}) f_e(z,w)\big) \leq (\epsilon'/\mathsf{e})^2 * n$$

Let, $\epsilon'^2 = \epsilon$. By the Markov inequality, we get

$$\begin{aligned}
&\mathsf{Pr}[\tilde{f}_e(x,y) > f_e(x,y) + \epsilon * n] \\
&= \mathsf{Pr}[\forall_i.\ \mathcal{M}_i[h_i(x), h_i(y)] > f_e(x,y) + \epsilon * n] \\
&= \mathsf{Pr}[\forall_i.\ f_e(x,y) + X^i_{x,y} > f_e(x,y) + \epsilon * n] \\
&= \mathsf{Pr}[\forall_i.\ X^i_{x,y} > \mathsf{e} * E(X^i_{x,y})] < \mathsf{e}^{-d} \leq \delta
\end{aligned}$$

□

Hence, **TCM** generates the same number of collisions and the same error bounds under the same probabilistic guarantees as CountMin for edge queries.

[9]The parameters ϵ and δ are usually set by the user.

A.2 Node Queries

We first discuss the query for node out-flow, *i.e.,* $f_v(a, \rightarrow)$. Consider the stream of edges $e:(a,*;t)$, *i.e.,* edges from node a to any other node indicated by a wildcard $*$. Drop the destination (*i.e.,* the wildcard) of each edge. The stream now becomes a stream of tuples $(a, \omega(t))$ where $\omega(t)$ is the sum of the weights of the outgoing edges from node a at time t. When a query is posed to find the weighted out-degree (*i.e.,* the sum of the weights of all edges whose source node is a) of node a, CountMin [14] returns the minimum of the weights in different hash buckets as the estimation of the flow out of node a. The unweighted out-degree (*i.e.,* the number of edges with source a) can be calculated similarly by setting $\omega(t)$ above to 1. Clearly, by construction, the answer obtained is an over-estimation of the true out-degree because of two reasons: (a) collisions in the hash-buckets, and (b) self-collision, *i.e.,* we do not know if an edge has been seen previously and thus count the outgoing edge again even if we have seen it.

The cases for in-flow point queries $f_v(a, \leftarrow)$ for directed graphs and flow point queries $f_v(a, -)$ for undirected graph queries can be analyzed using a similar discussion as shown above, which are thus omitted here.

Given the same number of hash buckets, the error estimates for point queries (see Sec. 4.1 of [14]) hold for these cases.

Lemma 1.2: *The estimated out-flow (in-flow) is within a factor of ϵ of the actual out-flow (in-flow) with probability δ if we use $d = \lceil ln(1/\delta) \rceil$ pair-wise independent hash functions and the number or rows $w = \lceil \mathsf{e}/\epsilon \rceil$.* □

B. ADDITIONAL ALGORITHMS

B.1 Conditional Heavy Hitters

As discussed in Section 4, different from heavy hitters, conditional heavy hitters [31] are to find heavy hitters that are *locally* popular by considering edge connections. In other words, instead of finding which nodes are most popular as heavy hitters do, it is to find the most popular neighbors to the most popular nodes.

In the following, we will present an algorithm of how **TCM** can be used to monitor a graph stream and to report conditional heavy hitters. Before we give the algorithmic details, let us define the problem first.

Conditional heavy hitters. Consider a graph stream $G = \langle e_1, e_2, \cdots, e_m \rangle$ as defined in Section 3.1. Assume *w.l.o.g.* that each edge $e_i : (x_i, y_i; t_i)$ is a directed edge from node x_i to node y_i. The problem of *conditional heavy hitters* is to find top-k heavy hitters with the most aggregate in-flow weight, and for each detected heavy hitter y, find its associated top-l nodes that send the highest weights to y.

The other two versions, the one to find nodes with top-k out-flow weights, and the other version with undirected edges, can be similarly defined. Therefore, we only present the algorithm for the problem defined above, which can be readily converted to solve other versions of conditional heavy hitter problems.

Algorithm 1. The algorithm to monitor a graph stream G to report conditional heavy hitters, is shown in Algorithm 1. Given a graph stream G, two natural numbers k and l, it outputs (estimates) top-k heavy nodes, with each one asso-

Algorithm 1: Monitoring Conditional Heavy Hitters

Input: a graph stream G, a parameter k, and a parameter l.
Output: top-k heavy nodes, with each top-l heavy neighbors.

```
1  Initialize a TCM sketch, denote by M;
2  Initialize a global sorted list hh := ∅;
3  for each edge e_i : (x_i, y_i; t_i) in G do
4    Update the TCM sketch M using e_i;
5    Let in_weight := f̃_v(y_i, ←);
6    Let neigbor_weight := f̃_e(x_i, y_i);
7    if y_i is in hh then
8      if x_i is in y_i.hn then
9        Update the value in y_i.hn w.r.t. x_i to neigbor_weight;
10     elseif neigbor_weight > y_i.hn.min_weight then
11       if y_i.hn.size = l then
12         Delete the one in y_i.hn having the min_weight
13       Insert (neigbor_weight, x_i) to y_i.hn;
14   else                                   // y_i is not in hh
15     if in_weight > hh.min_weight and hh.size = k then
16       Delete the one in hh having the min_weight;
17     if hh.size < k then
18       Initialize a sorted list hn for y_i;
19       Insert (neigbor_weight, x_i) to y_i.hn;
20       Insert (in_weight, y_i) to hh;
21 return hh;
```

ciated with top-l heavy neighbors. It first builds a **TCM** sketch (line 1), and initializes a sorted list to maintain in order to maintain top-k heavy hitters (line 2). It then processes the incoming graph stream edges (lines 3-20). For each incoming edge, it first updates the sketch (line 4; see Section 5.1.1), and then estimates its in-flow weight (line 5; see Section 4.2) and the edge weight (line 6; see Section 4.1). If the node is already in the maintained top-k heavy hitters, it updates the associated heavy neighbors (lines 7-14). If its neighbor is already a hot neighbor (line 8), it simply updates using the new estimated edge weight (line 9). Otherwise, when either current visited node has maintained less than top-l hot neighbors, or it already has top-l hot neighbors but the new neighbor has a higher weight, it will update the hot neighbors correspondingly (lines 10-13). In the case the the currently visited node is not a heavy hitter (line 14), it first removes the maintained heavy hitter with the smallest in-flow weight, if the newly inserted node has a higher weight (lines 15-16). The heavy hitter will be updated by the new heavy hitter correspondingly (lines 17-20). Note that, if the new heavy hitter has a less weight than the minimum weight in the maintained k heavy hitters, it will not be inserted since the size of sorted heavy hitter list will not be updated (*i.e.,* lines 15-16 will not be executed). Consequently, the lines 17-20 will not be executed since the size of the maintained heavy hitter is k. Finally, it returns the top-k heavy edges, with each one associated with top-l heavy neighbors, which reflects the triangle relationships.

Complexity. It is readily to see that each step in the **for** loop (lines 3-20) will take constant time. Observe the followings. The operations for estimating the in-flow weight (line 5; see Section 4.2) and the edge weight (line 6; see Section 4.1) are in constant time. In addition, the cost of maintaining a sorted list (either the hh or a hn) is also in cost time, since the sorted list has a bounded size, k for hh and l for hn, where both k and l are constants. Hence, in total, the algorithm runs in $O(|E|)$ time, which is linear to the input.

Algorithm 2: Finding Heavy Triangle Connections

Input: a graph stream G, a parameter k, and a parameter l.
Output: top-k heavy edges, with each top-l heavy connections.

```
1  Initialize a TCM sketch, denote by M(only one w*w matrix);
2  Find top-k heavy edges he while processing G;
3  for each edge e : (x, y) in he do
4    connections := ∅;
5    for i ∈ [1, w] do
6      if M[i][h(x)] > 0 and M[i][h(y)] > 0
7        connections := connections ∪ ext(i);
8      Rank connections by (f̃_e(z,x)×f̃_e(z,y))/(f̃_e(z,x)+f̃_e(z,y));
9      Associated top-l connections from the above ranking;
10 return heavy edges and their associated top-l connections;
```

B.2 Heavy Triangle Connections

In this section, we study a new problem for community detection.

Heavy triangle connections. Consider a graph stream $G = \langle e_1, e_2, \cdots, e_m \rangle$. The problem of *top-k triangle connections* is first to find top-k heavy edges, and then for each detected heavy edge (x, y), to find top-l nodes z ranked by the function $\big(f_e(z,x) \times f_e(z,y)\big)/\big(f_e(z,x) + f_e(z,y)\big)$.

Intuitively, the problem is to identify frequently communicated node pairs, and then identify nodes that communicate frequently with both nodes. Naturally, the detected relationships are in the shapes of triangles. In cyber security, when finding two suspicious IPs (*i.e.,* heavy edges), it is to also report other suspicious IPs. Take DBLP for another example, when identifying two authors who wrote many papers together *e.g.,* Philip S. Yu and Jiawei Han, or Divy Agrawal and Amr El Abbadi, it is to find the ones who work closely with both of them.

Note that the strategy of using **TCM** to estimate top-k heavy edges has been discussed in Section 6.2, Exp-1 (d). Given top-k heavy edges and only the **TCM**, it is not possible to identify the common neighbors since only hashed values (not the original values) are maintained. To do so, we show the power of the simple extension as discussed in Section 5.1.4, the extended graph sketch.

Algorithm 2. The algorithm to compute heavy triangle connections is shown in Algorithm 2. Given a graph stream G, two natural numbers k and l, it outputs (estimates) top-k heavy edges, with each one associated with top-l heavy connections. It first initializes a **TCM** sketch (line 1). To simplify the discussion, we consider only one hash function (*i.e.,* $d = 1$), and the case for $d > 1$ hash functions can be easily adapted. It then processes the graph stream G, maintains the sketch, and finds top-k heavy edges (line 2; see the discussion in Section 6.2, Exp-1 (d)). It then iteratively processes each heavy edge (lines 3-9). It first computes the connections for the edge being processing, using the extended graph sketch (lines 5-7). It then ranks all connections using estimated edge weights (line 8) and picks top-l of them (line 9). Finally, it returns the discovered heavy triangle connections.

Complexity. It is readily to see that for each edge, finding

	edge	node	conditional heavy nodes	path (reachability)	graph (explicit edges labels)	heavy triangle connections
TCM	✓	✓	✓	✓	✓	✓
CountMin (edge) or gSketch	✓	✗	✗	✗	✓	✗
CountMin (node)	✗	✓	✗	✗	✗	✗
sample-edge	✓	✗	✗	✗	✗	✗
sample-node	✗	✓	✗	✗	✗	✗

Table 3: Analytics supported by different sketches

	$d=1$	$d=3$	$d=5$	$d=7$	$d=9$
CountMin	35.6	28.6	26.3	25	24
TCM	35.9	28.6	26.1	24.8	23.9
gSketch	29.8	25.2	23.5	22.5	21.8
TCM (edge sample)	29.4	24.7	23	22	21.4

Table 4: Average relative errors (DBLP)

	$d=1$	$d=3$	$d=5$	$d=7$	$d=9$
CountMin	40.2	13.2	7.9	5.7	4.5
TCM	41	13	7.7	5.6	4.4
gSketch	5.7	3.6	3	2.63	2.4
TCM (edge sample)	5.9	3.6	3	2.6	2.3

Table 5: Average relative errors (GTGraph)

all connections will take $O(|V|)$ time. Ranking all connections will take $O(|V|log|V|)$ time. Since both k and l are constants, the total running time of Algorithm 2 is thus $O(|V|log|V|)$.

C. ADDITIONAL EXPERIMENTS

C.1 Sketch Expressiveness

The types of analytics supported by different sketches are summarized in Table 3, which tells the followings:

- For existing sketches, either frequency counts-based or sample-based, in order to support different analytics, they have to build an *ad-hoc* sketch to support a specific task, *e.g.,* edge or node.
- gSketch has been proposed to improve CountMin by considering data distribution. However, due to its inherent one-dimensional data structure, they have limited powered in supported graph analytics, which has been illustrated in Example 2.
- **TCM** is much more generalized in supporting many types of analytics in a single sketch, with the reason that **TCM** is inherent a two-dimensional structure, which keeps all graph connectivities in the original graph, as explained in Example 3. The improved power of using a two-dimensional structure vs. a one-dimensional structure is evident in supporting graph based applications.

In summary, the above analysis shows that **TCM** is much more general than *state-of-the-art* sketches.

C.2 More Comparisons with gSketch

We further show the comparisons with gSketch using DBLP and GTGraph data sets.

The result for DBLP is given in Table 4 and for GTGraph is shown in Table 5, by varying the number d of hash functions from 1 to 9 with a step of 2. Here, we used 10 data partitions.

The benefit of using data partitions for GTGraph is more significant than DBLP, in terms of the improved average relative errors. The main reasons are that (1) the range of weights for DBLP is smaller, and (2) the size of DBLP is smaller. In such case, the effect of partitioning the data to reduce the probability of hash collisions between high and low weight elements is not significant.

The above results, together with Section 6.2 Exp-1(e), show that the techniques proposed by gSketch can be easily integrated to **TCM** to reduce the average relative errors. Moreover, the average relative errors of gSketch and improved **TCM** using edge samples are very close. Again, it proves the positive result that our generalized sketch can get very close performance, when being compared with an *ad-hoc* solution.

C.3 Effectiveness with NDCG Measure

Normalized discounted cumulative gain (NDCG) [25] measures of ranking quality, which calculates the gain of a result based on its position in the result list and normalizes the score to $[0, 1]$ where 1 means perfect top-k results. Using the same setting of IP flow for heavy edges/nodes in Section 6.2, we show the NDCG results below, where he (resp. hn) is for heavy edges (resp. heavy nodes). It shows that **TCM**, as well as CountMin based method, can return top-k results with good NDCG scores. The results for other data sets are omitted for similar results and lack of space.

k	he (**TCM**)	he (CountMin)	hn (**TCM**)	hn (CountMin)
10	0.99	0.99	1	1
100	0.99	0.99	0.99	1
500	0.99	0.99	0.99	0.99

C.4 Query Time

We show the query time of edge queries for GTGraph on both the sketch and the original graph in the below table. Each row indicates the number of edge queries. For each set of edge queries, we first equally partitioned all edges in 10 buckets based on edge weights. We then picked 1/10 edges from each bucket. **TCM** is stored in an adjacency matrix and each query is in constant time, hence it is very efficient. For the original graph, it can only be stored as an adjacency list (see Section 6.2 Exp-5 for the discussion), each query needs a scan to locate the edge, which is thus very expensive. We then built a hash index on the nodes of adjacency list to favor it, such that an edge is first to use the hash index to locate the node and then scan its adjacency list to find the edge. However, as shown in the last column, it is still an order of magnitude slower than **TCM**.

#-queries	**TCM** (summary)	adjacency list (original graph)	hashed list (original graph)
100	0.001 secs	5.832 secs	0.013 secs
1000	0.005 secs	164.414 secs	0.072 secs
10000	0.028 secs	2022.682 secs	0.213 secs

The results for the other two data sets, DBLP and IP flow, are similar and thus omitted due to space constraints. Also, the efficiency for the other types of queries have similar conclusion, which is thus omitted.