Counting Butterflies in Fully Dynamic Bipartite Graph Streams

Serafeim Papadias[#] Varun Pandey[#] Zoi Kaoudi[+] Jorge-Arnulfo Quiané-Ruiz[+*] Volker Markl[#]
{s.papadias, varun.pandey, volker.markl}@tu-berlin.de[#] {zoka, joqu}@itu.dk[+]
Technische Universität Berlin[#], ITU Copenhagen[+]

***Abstract*—A bipartite graph extensively models relationships between real-world entities of two different types, such as user-product data in e-commerce. Such graph data are inherently becoming more and more streaming, entailing continuous insertions and deletions of edges. A butterfly (i.e., 2×2 bi-clique) is the smallest non-trivial cohesive structure that plays a crucial role. Counting such butterfly patterns in streaming bipartite graphs is a core problem in applications such as dense subgraph discovery and anomaly detection. Yet, existing approximate solutions consider insert-only streams and, thus, achieve very low accuracy in fully dynamic bipartite graph streams that involve both insertions and deletions of edges. Adapting them to consider deletions is not trivial either, because different sampling schemes and new accuracy analyses are required. We propose Abacus, a novel approximate algorithm that counts butterflies in the presence of both insertions and deletions by utilizing sampling. We prove that Abacus always delivers unbiased estimates of low variance. Furthermore, we extend Abacus and devise a parallel mini-batch variant, namely, ParAbacus, which counts butterflies in parallel. ParAbacus counts butterflies in a load-balanced manner using versioned samples, which results in significant speedup and is thus ideal for critical applications in the streaming environment. We evaluate Abacus/ParAbacus using a diverse set of real bipartite graphs and assess its performance in terms of accuracy, throughput, and speedup. The results indicate that our proposal is the first capable of efficiently providing accurate butterfly counts in the most generic setting, i.e., a fully dynamic graph streaming environment that entails both insertions and deletions. It does so without sacrificing throughput, and even improves it with the parallel version.**

***Index Terms*—butterfly counting, bipartite streaming graphs, fully dynamic streams, edge deletions, approximate estimation**

I. Introduction

Bipartite graphs are a natural fit when it comes to modeling the relationship between two different types of entities in real-world applications [1], [2]. For instance, Alibaba's e-commerce platform models relationships between users and products via bipartite graphs [3]. These graphs consist of billions of vertices (e.g., products, buyers, and sellers) and hundreds of billions of edges (e.g., representing clicks, orders, and payments). Nowadays, real-world bipartite graphs are inherently *streaming*, entailing continuous insertions and deletions of vertices and edges [4] For example, Alibaba's user-product interactions are streams of very high velocity: Reports of customer purchase activities specify that during a heavy period in 2017, 320 PB of log data were generated within only a six-hour period [5]. Consequently, it is vital to swiftly analyze the huge volume of incoming data and identify underlying trends or patterns in bipartite graph streams in order to gain valuable insights.

The butterfly is the most basic substructure in bipartite graphs, similar to the triangle in unipartite graphs. A butterfly (i.e., 2×2 biclique) is a complete bipartite subgraph with two vertices belonging to one entity type and two vertices belonging to another entity type. Butterfly count is a metric that plays an important role in many applications. For instance, it is used to measure the *butterfly clustering coefficient* in a bipartite graph, which indicates how cohesive a bipartite graph is and can highlight how entities are clustered [6], [7], [8], [9]. This metric is important in many real-world applications, such as: in online recommendation systems to identify similar items [10], [11], cluster users, and enhance collaborative-filtering [12]); in real-time anomaly detection [13], [14], [15], [16]; in fraud detection [2]. Furthermore, counting butterflies for each edge is required for the computation of k-bitrusses [17], [18], [19], which is also used in a varienty of applications, such as community and spam detection [20], [21], [22], [23].

Approaches that exactly count butterflies in static graphs [24], [19], [1], [25] are prohibitive for streams because they necessitate storing the whole graph in memory and take quadratic time to enumerate butterflies in the worst-case. Wang et al. [19], [1] devise a vertex-priority-based algorithm that considers both insertions and deletions of edges in a batch-dynamic setting. However, their per-batch computation mechanism cannot keep up with the pace of bipartite graph streams and results in stale counts in streaming applications. Approximate streaming methods that estimate the butterfly counts also exist [25], [26], [15]. However, all of them are strictly focusing on insert-only bipartite graph streams, and none of them can support both deletions and insertions of edges. This is primarily due to the inability of their sampling algorithms to maintain uniform random samples in presence of deletions. As a result, they fail to provide accurate butterfly counts in the most general and realistic setting, the fully dynamic one. This also directly results in a degradation of the output quality of many algorithms that rely on butterfly counts. Consider, for instance, the precision and recall metrics, which are utilized by anomaly detection algorithms to measure the quality of detected anomalies over time. Typically, an anomaly in bipartite graph streams appears when a certain number of butterflies that are formed is above some threshold [15], [16],

[*] The paper is dedicated in memory of Jorge; a mentor and a colleague who passed away so unexpectedly in May 2023.

[27]. Therefore, precision and recall will degrade significantly if the butterfly counts are maintained inaccurately, which will happen if edge deletions are ignored and not treated accordingly. In order to improve the quality of anomaly detection, it is vital to address the edge deletions appropriately.

Counting butterflies in fully dynamic bipartite graph streams is challenging for three main reasons: (i) Due to the complexity of the butterfly counting problem and the nature of bipartite graphs, exact algorithms are prohibitive as they require the whole graph to be stored in main memory. (ii) An approximate solution is more suitable for achieving high throughput and maintaining a small memory footprint, but should provably deliver unbiased estimations. Doing so by using simple sampling techniques that only work for insert-only streams (e.g., reservoir sampling) falls short for a bipartite graph stream that also involves deletions; (iii) Devising a parallel algorithm for increased throughput is appealing for graph streams but non-trivial. Specifically, all threads should have almost the same workload to not introduce stragglers while at the same time reducing contention among them.

We propose ABACUS, which tackles all the above-mentioned challenges by providing accurate estimates for the butterfly counts in bipartite graph streams with deletions. Specifically, we use Random Pairing (RP) [28] to maintain a uniform random sample of bounded size from a fully dynamic graph stream over which we estimate the butterfly counts. We prove that our estimates are unbiased. Importantly, ABACUS eliminates the need for extra hashmaps that bookkeep wedges [19], [25] by refining its butterfly counts via set intersection operations. Furthermore, we present a parallel variant of ABACUS, called PARABACUS, which processes a graph stream in mini-batches [29], [30] using all available threads. More precisely, we maintain a versioned sample, which incorporates different states of the maintained sample that correspond to different edges in the mini-batch. Finally, we conduct the per-edge butterfly counting attributed to each edge in a mini-batch in parallel to enhance throughput.

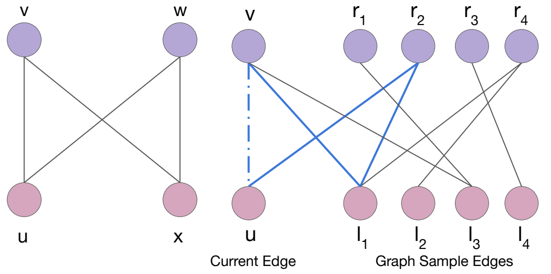
In summary, we make the following major contributions:

(1) We formalize the problem of counting butterflies in fully dynamic streams, entailing both insertions and deletions of edges (Section II).

(2) We present ABACUS, an approximate algorithm handling bipartite graph streams with both insertions and deletions. We present a set intersection-based process for updating the butterfly counts that makes our algorithm more space-efficient as it alleviates the need for bookkeeping. We refine our estimates by calculating the probability that a butterfly is formed between an edge that arrives and the sample we maintain (Section III).

(3) We prove that ABACUS always maintains unbiased estimates for butterfly counts. We provide the time and space complexity of our algorithm (Section IV).

(4) We present the parallel version of ABACUS, namely, PARABACUS, which processes the graph stream in mini-batches using all available threads to enhance throughput. Specifically, we maintain sample versions and distribute the counting of per-edge butterflies attributed to each edge in a mini-batch to all available threads (Section V).



(a) Butterfly (b) Running Example

Fig. 1: **(a)** Butterfly structure. **(b)** Running example showing the graph sample $\mathcal{S}$, an incoming edge $\{u, v\}$, and the butterflies that $\{u, v\}$ potentially forms with the edges in $\mathcal{S}$.

(5) We conduct comprehensive experiments on a variety of real-world bipartite graph workloads. We show that ABACUS and PARABACUS give both efficient and accurate estimates, which align with our theoretical analysis (Section VI).

We then discuss related work in Section VII, where we stress that existing butterfly counting algorithms for graph streams fail to address the above-mentioned challenges attributed to edge deletions. We conclude the paper in Section VIII.

II. PROBLEM STATEMENT

Let us first introduce the notation we use throughout the paper (shown in Table I) and the core definitions necessary to formally define the problem we address. We consider undirected and unweighted bipartite graphs without zero-degree vertices, and without duplicate edges. We begin by defining a fully dynamic bipartite graph stream as follows:

Definition 1. *A **fully dynamic bipartite graph stream** Π is a sequence of elements $(e^{(1)}, e^{(2)}, \ldots)$. Let $G^{(t)} = (V^{(t)}, E^{(t)})$ be a bipartite graph that contains all edges $E^{(t)}$ that appear in the sequence Π up to time t (inclusive), and let their corresponding set of vertices $V^{(t)} = L^{(t)} \cup R^{(t)}$, which is separated into two disjoint partitions; a left one, $L^{(t)}$, and a right one, $R^{(t)}$, where $L^{(t)} \cap R^{(t)} = \emptyset$. It holds that $E^{(t)} \subseteq L^{(t)} \times R^{(t)}$. Also, $N_v^{(t)} = \{w \in V^{(t)} | (v, w) \in E^{(t)}\}$ denotes the set of neighbours of a vertex $v \in V^{(t)}$. For each discrete timestamp $t \geq 0$, let $e^{(t)} = (\{u^{(t)}, v^{(t)}\}, \delta)$ be the t^{th} element in the sequence Π, where $\{u^{(t)}, v^{(t)}\}$ is the actual edge and $\delta \in \{+, -\}$ denotes the change in G at time t, i.e., whether the edge is inserted or deleted. More precisely, $(\{u^{(t)}, v^{(t)}\}, +)$ signifies that the edge did not exist up to time $t-1$, i.e., $(\{u^{(t-1)}, v^{(t-1)}\} \notin E$, but will be inserted at time t, i.e., $(\{u^{(t)}, v^{(t)}\} \in E$. Similarly, $\{u^{(t)}, v^{(t)}\}, -)$ indicates that an existing edge $\{u^{(t)}, v^{(t)}\} \in E$ is about to get deleted.*

Implicitly, we assume that only new edges can be inserted (i.e., multigraphs with parallel edges are out of scope) and only edges that already exist can be deleted. Also, vertices that end up with degree zero, are deleted from $V^{(t)}, \forall t$. A butterfly is a complete 2×2 bipartite subgraph, where two vertices of one bipartition are connected with two vertices of the other

TABLE I: Notations.

Π	fully dynamic bipartite graph stream
$G^{(t)} = (V^{(t)}, E^{(t)})$	bipartite graph at time t
$V^{(t)} = L^{(t)} \cup R^{(t)}$	set of vertices (left and right partition)
δ	edge insertion ($\delta = +$) or deletion ($\delta = -$)
$\{u^{(t)}, v^{(t)}\}$	edge between two vertices u and v at time t
$e^{(t)} = (\{u^{(t)}, v^{(t)}\}, \delta)$	an element of Π at time t
$\{u^{(t)}, v^{(t)}, w^{(t)}, x^{(t)}\}$	butterfly formed by vertices u, v, w, x at time t
$B^{(t)}$	set of butterflies in $G^{(t)}$ at time t
$\mathcal{S}^{(t)}$	sample of the stream Π at time t
$N_u^{(t)}$	set of neighbours of vertex $u \in V^{(t)}$ in $G^{(t)}$
$N_u^{\mathcal{S},(t)}$	set of neighbours of vertex u in the sample $S^{(t)}$
$d_u^{(t)}$	degree of vertex $u \in V^{(t)}$
$d_u^{\mathcal{S},(t)}$	degree of vertex u in the sample $\mathcal{S}^{(t)}$
c	butterfly count estimate
c_b	#uncompensated ("bad") deletions $\in \mathcal{S}^{(t)}$
c_g	#uncompensated ("good") deletions $\notin \mathcal{S}^{(t)}$
k	memory budget (max #edges in $\mathcal{S}$)
$\mathcal{C}^{(t)}$	set of created butterflies up to time t
$\mathcal{D}^{(t)}$	set of deleted butterflies up to time t
M	number of edges in a mini-batch
α	percentage of edges that are deletions

bipartition. For instance, a butterfly subgraph is depicted in Figure 1a. Let us now formally define a butterfly pattern that appears in such bipartite graphs as follows:

Definition 2. *Given a bipartite graph $G^{(t)}$ and four vertices $u, v, w, x \in V^{(t)}$, where $u, x \in L^{(t)}$ and $v, w \in R^{(t)}$, a butterfly is the induced subgraph $\{u, v, w, x\}$ that is formed by the edges $(u, v), (u, w), (v, x), (w, x) \in E^{(t)}$.*

We now define the problem of estimating the number of butterflies in a fully dynamic graph stream Π under infinite window semantics [31]. Let $B^{(t)}$ denote the set of all butterflies in a bipartite graph $G^{(t)}$. We, thus, formally define the problem we focus on, as follows:

Problem Statement. *Given a fully dynamic bipartite graph stream $(e^{(1)}, e^{(2)}, \dots)$ comprised of a sequence of edge insertions and deletions in a bipartite graph G, we aim to maintain butterfly count estimates $|B^{(t)}|$, using bounded memory, such that the estimates are unbiased and the errors are minimized.*

Note that we assume the traditional data stream model where the changes in the input stream can be accessed only once in the given order unless they are explicitly stored in memory.

III. Abacus

We now present Abacus, an algorithm designed to facilitate the efficient and accurate counting of butterflies in fully dynamic bipartite graph streams. Abacus employs sampling to maintain a subset of the edges of bounded size, and estimates butterfly counts through the maintained sample. First, we give an overview of the general workflow of Abacus. Subsequently, we present the sampling scheme Abacus employs for maintaining a uniform random sample. Finally, we illustrate the exact methodology we use to refine butterfly estimates.

Algorithm 1 Abacus

```
1:  Input: fully dynamic input bipartite graph stream $\Pi = (e^{(1)}, e^{(2)}, \dots)$, graph sample $\mathcal{S}$, memory budget $k \geq 2$
2:  Output: butterfly count estimate $c$
3:  $\mathcal{S} \leftarrow \emptyset$, $|E| \leftarrow 0$, $c \leftarrow 0$, $c_b \leftarrow 0$, $c_g \leftarrow 0$
4:  for each element $e^{(t)} = (\{u, v\}, \delta)$ in $\Pi$ do
5:      // Update the Butterfly Count
6:      $increment = \frac{sgn(\delta)}{Pr(|E|, c_b, c_g)}$        ▷ $sgn(\delta)$ is the sign of $\delta$
7:      if $\sum_{x \in S_u} d_x < \sum_{x \in S_v} d_x$ then choose $v$        ▷ Else, $u$
8:      for each vertex $w \in N_u^{\mathcal{S}} \setminus v$ do        ▷ $u$'s neighbors $\in \mathcal{S}$
9:          $\mathcal{CN}$ = SetIntersection($N_w^{\mathcal{S}}, N_v^{\mathcal{S}}$)
10:         for each vertex $x \in \mathcal{CN}$ do
11:             $c$ += $increment$
12:     // Update the Sample
13:     if $\delta = +$ then InsertToSample($\{u, v\}$, $k$)
14:     else if $\delta = -$ then DeleteFromSample($\{u, v\}$)
```

A. Overview

Let us now elaborate on Abacus's main workflow as shown in Algorithm 1 (lines 4-14). It ingests a fully dynamic graph stream, element by element, and maintains a uniform random sample $\mathcal{S}$ of bounded size k by utilizing the Random Pairing (RP) [28]. Note that the memory budget k is the maximum possible sample size in Abacus, and thus, we use the terms *memory budget* and *sample size* interchangeably throughout the paper. The RP sampling scheme is suitable for streams that contain both insertions and deletions and ensures that the sample is always uniform. Uniformity is a property that enables us to extrapolate our estimations to the whole graph stream. Under deletions, the usual sampling schemes, such as reservoir sampling [32], do not guarantee uniformity. The processing happens per element (line 4), and once processed, Abacus evicts it from the main memory. On a high level, for each incoming element, Abacus first refines the maintained butterfly count estimates and then updates the sample $\mathcal{S}$. Specifically, for each incoming edge (either edge insertion or deletion), Abacus finds all the butterflies that the edge forms with the edges in the sample $\mathcal{S}$ (lines 8-11) and updates the butterfly count (lines 6, 11). Due to the inherent complexity of the butterfly structure itself, it is challenging and important to spot the formed butterflies efficiently. Additionally, one must prove that the estimates provided via the sample are unbiased, and thus, accurate and robust. Afterwards, Abacus updates the sample $\mathcal{S}$ (lines 13-14) by essentially deciding whether to insert the edge $\{u, v\}$ to $\mathcal{S}$ (if $\delta = +$) or delete it from $\mathcal{S}$ (if $\delta = -$), where δ indicates an edge insertion or deletion.

B. Counting Butterflies per Edge

We now explain how Abacus finds the number of butterflies an incoming edge forms with the edges in the sample, as shown in Algorithm 1 (lines 4-11). As we refine our butterfly counts using every incoming edge, irrespective of whether the edge is later included in the sample, the operation of per-edge butterfly counting must be as efficient as possible.

For each incoming edge $\{u, v\}$, Abacus chooses to compute the butterflies formed using the vertices on u's side that are neighbors of v, or using the vertices on v's side that are neighbors of u (line 7). ABACUS chooses to explore vertices on the side of the incoming edge's endpoint that has the smallest cumulative degree. This is a common heuristic [25], [19], and allows for choosing the cheapest side to conduct the counting. In the context of ABACUS, choosing the cheapest side leads to conducting cheaper set intersections using vertices that belong to the bipartition with the smaller cumulative degree. The fact that the complexity of a set intersection operation between two sets is the size of the smallest set, leads to improved performance. As we see in our running example in Figure 1b, we choose the side of vertex v or equivalently we conduct the counting using the neighbors of u in $\mathcal{S}$ as they have the smallest cumulative degree. Specifically, in our example u has only one neighbor in the sample $\mathcal{S}$, i.e., r_2 with degree 2, whereas v has two neighbors in $\mathcal{S}$, i.e., l_1 and l_2 with cumulative degree equal to 5.

Consider that ABACUS chooses v (line 7). If so, it explores every neighbor w of u in the sample $\mathcal{S}$ (i.e., excluding v) (line 8). Subsequently, ABACUS finds the common neighbors, $\mathcal{CN}$, as the result of the set intersection between the set of neighbors of v and the set of neighbors of w (line 9). The result, $\mathcal{CN}$, of a set intersection (if not empty) contains all the vertices that serve as the fourth vertex x forming a butterfly along with the edge's endpoints u, v, and the current vertex w that ABACUS explores amongst the neighbors of u (line 10-11). In case the result of the set intersection is empty, this means that no butterfly that contains the incoming edge $\{u, v\}$ is formed through the vertex w. In our running example in Figure 1b, since ABACUS chooses v, the only vertex that is neighbor of u in $\mathcal{S}$ is r_2, belonging to the right bipartition (shown in the upper part). Vertex v has neighbours $N_v^{\mathcal{S}} = \{u, l_1, l_2\}$ and vertex r_2 has neighbors $N_{r_2}^{\mathcal{S}} = \{u, l_1\}$. We exclude vertex u from the corresponding neighboring sets as $u \notin \mathcal{S}$, and we observe that their common neighbor, l_1, indicates that the butterfly $\{u, v, l_1, r_2\}$ has been formed. Finally, ABACUS adjusts the butterfly count for each discovered butterfly with a certain *increment* (line 11), as we describe in the sequel.

C. Random Pairing for Uniform Samples

Intuitively, the larger the size of the sample $\mathcal{S}$ that ABACUS maintains, the more accurate the butterfly count estimations. This is in consistence with what has been demonstrated for triangle count estimation methods on fully dynamic graph streams [14], [33]. Therefore, to minimize the information loss, ABACUS strives to keep as many elements in the sample $\mathcal{S}$ as possible within a predefined memory budget $k \geq 2$. Random Pairing (RP) [28] is a sampling scheme that always maintains a uniform random sample containing at most k edges, given a fixed memory budget k and a fully dynamic bipartite graph stream. Initially, the sample $\mathcal{S}$ as well as the bipartite graph stream are empty. Note that we initialize the compensation counters, c_b and c_g, to zero (Alg. 1, line 3). Assuming that there is a set E of edges in the input bipartite stream that have not yet been deleted, we now describe the core functionality for inserting or removing an edge to or from S that ABACUS maintains, as illustrated in Algorithm 2. More precisely, when an edge deletion appears (lines 12-16), if the edge is in $\mathcal{S}$, then RP increases the c_b (line 15); otherwise, it increases c_g (line 16). Intuitively, the counters c_b and c_g signify the number of deletions that need "compensation" from upcoming insertions. On the other hand, when an edge insertion appears (lines 2-10) and there are no deletions to compensate for, i.e., $c_b + c_g = 0$ (line 17), ABACUS processes the upcoming edge insertion as in reservoir sampling [32] (lines 4-6). In particular, if ABACUS has not exhausted the memory budget yet, i.e., $|\mathcal{S}| < k$, then we append the newly arrived edge to $\mathcal{S}$ (line 4); else, we replace a random edge in $\mathcal{S}$ with the new edge with a probability $k/|E|$ (lines 5-6). In case the sum of the compensation counters is not zero, then we consider them when calculating the probability of replacing an edge from $\mathcal{S}$ with the new one (line 7), and update the counters accordingly (lines 9-10). Note that the uniformity of $\mathcal{S}$ allows for exactly calculating the discovery probability of each butterfly in a deterministic way.

Algorithm 2 RANDOM PAIRING [28]

```
1: procedure InsertToSample($\{u, v\}$, $k$)
2:     $|E| \leftarrow |E| + 1$
3:     if $c_b + c_g = 0$ then
4:         if $|\mathcal{S}| < k$ then $\mathcal{S} \leftarrow \mathcal{S} \cup \{\{u, v\}\}$
5:         else if $Bernoulli(\frac{k}{|E|}) = 1$ then
6:             replace a random edge in $\mathcal{S}$ with $\{u, v\}$
7:     else if $Bernoulli(\frac{c_b}{c_b + c_g}) = 1$ then
8:         $\mathcal{S} \leftarrow \mathcal{S} \cup \{\{u, v\}\}$
9:         $c_b \leftarrow c_b - 1$
10:    else $c_g \leftarrow c_g - 1$
11: procedure DeleteFromSample($\{u, v\}$)
12:    $|E| \leftarrow |E| - 1$
13:    if $\{u, v\} \in \mathcal{S}$ then
14:        $\mathcal{S} \leftarrow \mathcal{S} \setminus \{\{u, v\}\}$
15:        $c_b \leftarrow c_b + 1$
16:    else $c_g \leftarrow c_g + 1$
```

D. Butterfly Count Update Mechanism

We now elaborate on how ABACUS updates its butterfly count estimates. As we described in Section III-B, when an edge (insertion or deletion) arrives, ABACUS first finds the butterflies that the edge forms with the edges in $\mathcal{S}$. Here, we show how ABACUS utilizes the spotted butterflies to update the butterfly count estimate. We explain *how much* ABACUS modifies its butterfly count estimate, such that it always remains unbiased.

In Algorithm 1, the specific increment amount with which ABACUS refines the maintained butterfly count is important for providing accurate estimations (lines 6, 11). More precisely, each incoming edge contributes to the creation of some new butterflies if it is an insertion, or causes the deletion of some existing butterflies if it is a deletion. We can reason about the created or deleted butterflies with the edges that exist in the sample, yet for the butterfly counts in the whole graph stream, we can only extrapolate using the actual counts we gain through the sample. Furthermore, the fact that we are maintaining a uniform random sample implies that the created or deleted butterflies are discovered with a certain probability, which we can exactly calculate. Specifically, each time an element $e^{(t)} = (\{u,v\},\delta)$ arrives (Algorithm 1, line 4), every created or deleted butterfly $\{u,v,w,x\}$, where $u,w \in L^{(t)}$ and $v,x \in R^{(t)}$, is successfully discovered (lines 8-11) if and only if three specific edges exist in the sample $\mathcal{S}$, namely, the edges $\{u,x\}$, $\{w,x\}$, and $\{v,w\}$. Assuming that $y = \min(k, |E^{(t)}| + c_g + c_b)$, which is the size of the sample $\mathcal{S}$, we prove that the above-mentioned butterfly discovery probability through the sample is as follows:

$$Pr(|E^{(t)}|, c_b, c_g) = \frac{y}{T} \cdot \frac{y-1}{T-1} \cdot \frac{y-2}{T-2}, \text{ with } T = |E^{(t)}| + c_b + c_g \quad (1)$$

where c_g, c_b are compensation counters for the edge deletions in the RP sampling, and $E^{(t)}$ is the set of edges that remain in the input bipartite streaming graph (without being deleted) after the t-th element of the stream is processed. For each butterfly that ABACUS discovers using the sample, it updates the corresponding butterfly estimates with the reciprocal of the probability that the butterfly is discovered (line 6). Utilizing the reciprocal of the discovery probability as the amount of change per butterfly discovered makes the expected amount of changes in the estimated butterfly counts of ABACUS exactly one, and thus, enables us to provide unbiased estimates. Therefore, for each incoming edge, ABACUS calculates the reciprocal *increment* based on Equation 1 and uses it to refine the count estimates for the butterflies that it discovers with the incoming edge (line 6,11).

IV. Accuracy and Complexity

We now study the accuracy of our algorithm and prove that ABACUS consistently maintains unbiased estimates of the butterfly count, with expected values that are the same as the true counts. Demonstrating the absence of bias in ABACUS is very important, as it ensures reliable estimations and accurate representations of true butterfly counts. We then present the time and space complexity of our algorithm.

A. Accuracy Analysis

Before proving the unbiasedness of ABACUS, we first provide the following lemma for the butterfly discovery probability.

Lemma 1 (Butterfly Discovery Probability)**.** *In* ABACUS*, any three distinct edges in the bipartite graph $G^{(t)} = (V^{(t)}, E^{(t)})$ are sampled with the probability shown in Equation 1. Therefore, assuming that $\mathrm{p}^{(t)}$ is the probability of the Equation 1 and $S^{(t)}$ is the sample of the bipartite graph, respectively, after the t-th element $e^{(t)}$ is processed by* ABACUS *(Algorithm 1), then the following holds:*

$$Pr(\{u,v\} \in \mathcal{S}^{(t)} \cap \{w,x\} \in \mathcal{S}^{(t)} \cap \{y,z\} \in \mathcal{S}^{(t)}) = \mathrm{p}^{(t)},$$
$$\forall t \geq 1, \forall \{u,v\} \neq \{w,x\} \neq \{y,z\} \in E^{(t)} \quad (2)$$

Proof. See Appendix A. □

We now formally prove that ABACUS maintains unbiased butterfly count estimates as stated in the following theorem:

Theorem 1 (Unbiasedness)**.** ABACUS *provides unbiased butterfly count estimates at any point in time. Specifically, for Algorithm 1 it holds:*

$$\mathbb{E}(c^{(t)}) = |B^{(t)}|, \forall t \geq 1 \quad (3)$$

where $|B^{(t)}|$ is the true butterfly count at time t.

Proof. See Appendix B. □

Theorem 2 (Variance)**.** ABACUS *provides estimates of bounded variance at any point in time. Specifically, for Algorithm 1 it holds:*

$$Var[c] = \gamma\mathbb{E}[c] - \mathbb{E}[c]^2$$
$$+ 2\gamma^2 (y_1 \frac{\binom{|E|-8}{k-8}}{\binom{|E|}{k}} + y_2 \frac{\binom{|E|-7}{k-7}}{\binom{|E|}{k}} + y_3 \frac{\binom{|E|-6}{k-6}}{\binom{|E|}{k}})$$

where $p_{i,j}$ is the probability that the graph sample $\mathcal{S}$ contains both the butterfly B_i and the B_j, the y_1, y_2, y_3 indicate how many pairs of butterflies that share 0, 1, *and* 2 *edges, respectively, exist in the graph at the current time t (omitted for simplicity), c is the butterfly count estimation of* ABACUS*, whose expected value equals the ground truth butterfly count, $|E|$ is the number of valid edges in the graph stream that have not yet been deleted, k is the memory budget of $\mathcal{S}$, and $\gamma = \binom{E}{k} / \binom{E-4}{k-4}$. A tight upper bound of the variance is:*

$$Var[c] \leq \gamma\mathbb{E}[c] + 2\gamma^2 \binom{E[c]}{2} \times \frac{\binom{|E|-6}{k-6}}{\binom{|E|}{k}} - \mathbb{E}[c]^2$$

Proof. See Appendix C. □

Corollary 1 (Concentration)**.** ABACUS *provides estimates that concentrate around the expected value at any point in time. Specifically, for Algorithm 1 and any constant $\lambda > 0$ it holds:*

$$Pr[|c - \mathbb{E}[c]| \geq \lambda \times \sqrt{Var[c]}] \leq \frac{1}{\lambda^2}$$

where c is the butterfly estimate of ABACUS.

Proof. See Appendix D. □

B. Complexity Analysis

Here, we analyze ABACUS's complexity with respect to both time and space when processing a bipartite graph stream.

Time Complexity. We provide the worst-case analysis in Theorem 3. Essentially, we claim that given a fixed memory budget k, ABACUS scales linearly with the number of elements in the input bipartite graph stream. The time complexity theorem is as follows:

Theorem 3 (Time Complexity of ABACUS)**.** *Algorithm 1 takes $O(k^2 t)$ time to process the first t elements in the input bipartite graph stream, where k is the maximum number of edges maintained in the sample.*

Proof. The most expensive operation in Algorithm 1, is the per-edge butterfly counting for spotting the butterflies that each incoming edge $e^{(t)} = (\{u^{(t)}, v^{(t)}\}, \delta)$ forms with the edges in the graph sample. The per-edge counting process takes

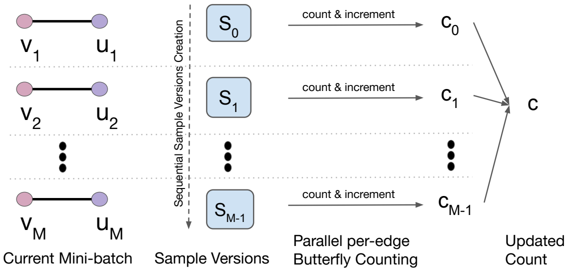


Fig. 2: Overview of ParAbacus.

$\Lambda = O(\min\{\sum_{x \in N_u^{\mathcal{S}}} \min\{d_x, d_v\}, \sum_{x \in N_v^{\mathcal{S}}} \min\{d_w, d_u\}\})$ time for an incoming edge $\{u, v\}$. All the vertex degrees are upper-bounded by k, which is the maximum number of edges in the sample. Therefore, $\Lambda = O(k^2)$, and the time complexity for processing the first t elements is $O(k^2 t)$. □

Space Complexity. We provide the space analysis in Theorem 4. In a nutshell, given a fixed memory budget k, Abacus needs to store at most k elements as a sample of the input bipartite graph stream, and a counter for the butterfly count estimate. The theorem for the space complexity is as follows:

Theorem 4 (Space Complexity of Abacus)**.** *Algorithm 1 has a space complexity of* $O(k)$.

Proof. While Abacus processes the first t elements of the input bipartite graph stream maintains a single estimate for the butterfly count. Furthermore, Abacus maintains up to k edges that consist of the graph sample, where k is the memory budget and a parameter of our algorithm. Therefore, the space complexity for processing the first t elements is $O(k)$. □

V. ParAbacus: Abacus goes Parallel

We now present ParAbacus, which is the parallel variant of Abacus that processes the bipartite graph stream in mini-batches. The primary challenge is to process the edges in a mini-batch simultaneously while attaining the same accuracy as Abacus. To achieve this, we revert Abacus's workflow, namely, we first conduct the sample updates that correspond to each edge in the mini-batch and create versions of the sample. Subsequently, we conduct the per-edge butterfly counting between an edge in the mini-batch and its corresponding version of the sample in parallel. Next, we first describe how ParAbacus processes each mini-batch and, then, provide correctness proof and its time and space complexities.

A. ParAbacus *Algorithm*

Sampling and Versioned Samples. ParAbacus processes the bipartite graph stream in mini-batches of size M. Figure 2 shows the overview of ParAbacus' workflow. In specific, we process the graph stream in mini-batches that contain M edges each, namely, $\{u_1, v_1\}$, $\{u_2, v_2\}$,…, $\{u_M, v_M\}$. Recall that in Abacus, for each incoming edge, we first conduct the per-edge butterfly counting to refine the butterfly count estimates and then trigger the sample update procedure. Since the per-edge butterfly counting is the most time-consuming operation in Abacus's workflow, we ought to effectively parallelize it when processing whole mini-batches. To achieve this, ParAbacus sequentially processes the edges in the mini-batch once to calculate all the sample states, which would be created as if the edges were processed by Abacus. Each different state of the sample S is a distinct sample version. For example, as shown in Figure 2, $\mathcal{S}_0$ indicates the state of the sample immediately after the arrival of the mini-batch. Assuming the edges arrive in the following sequence $\{u_1, v_1\}$, $\{u_2, v_2\}$,…,$\{u_M, v_M\}$, the edge $\{u_1, v_1\}$ will observe the $\mathcal{S}_0$ version of the sample. Subsequently, the edge $\{u_2, v_2\}$ would observe the version $\mathcal{S}_1$, which corresponds to the state of the sample after updating $\mathcal{S}_0$ with incorporating the edge $\{u_1, v_1\}$. Similarly, the edge $\{u_M, v_M\}$ would observe the $\mathcal{S}_{M-1}$ version, which corresponds to the state of the sample after incorporating the updates due to all the edges in the mini-batch except the M-th one. ParAbacus maintains all the calculated versions of the samples $S_0, S_1, \ldots, S_{M-1}$ in a single *versioned sample* data structure. In particular, we use adjacency lists to store the edges that we sample. More precisely, in the versioned sample, each vertex of the sample stores its neighbors in its adjacency list that might change between versions. However, from one version to another, we store only the discrepancies between the neighboring sets of each vertex to save space. Furthermore, along with each sample version we cache a triplet containing the following information: $\{s, c_g, c_b\}$, where s is the number of edges in the graph stream and c_g, c_b are the good and bad edge deletions that need compensation at the point of creation of the sample version. ParAbacus utilizes a triplet to calculate the increment using which it refines the butterfly count estimates, as we described in Section III.

Parallel Per-edge Butterfly Counting. After assembling the versioned sample, we have every sample state readily available. This allows ParAbacus to conduct the butterfly counting operations for all the M edges in the mini-batch in parallel. Specifically, we have to conduct per-edge butterfly counting between each edge in the mini-batch and its corresponding sample version using a separate thread among the available ones. For example, in Figure 2, we have to count the butterflies formed between edge $\{v_1, u_1\}$ and S_0, the ones formed between $\{v_1, u_1\}$ and S_1, all the way to the ones formed between $\{v_{M-1}, u_{M-1}\}$ and S_{M-1}. Assuming that there are p threads available and that we process mini-batches of M edges where $p \leq M$, ParAbacus groups the edges into p equal-sized sets. Therefore, each thread receives a subset of edges from the mini-batch and has to count the butterflies that its corresponding edges form with their corresponding sample versions. Subsequently, every edge, e.g., $\{u_1, v_1\}$, extrapolates its calculated butterfly count by multiplying it with an appropriate increment. We compute the increment for each edge using the information in its corresponding cached triplet $\{s, c_b, c_g\}$. In specific, we use Equation 1 as in Abacus to compute each increment and produce the partial counts, $c_0, \ldots, c_{M-1}$, as shown in Figure 2. Recall that depending on whether an edge is an insertion or a deletion its partial count can be positive or negative, respectively. Finally, all the calculated partial counts $c_0, c_1, \ldots, c_{M-1}$ are added to the old butterfly count so that the final refined count c is computed. Note that two different versions of the sample differ slightly from each other, i.e., up to M edges. Therefore, the vertex degrees among different sample versions are similar, and subsequently, the per-edge butterfly computations are balanced across threads, as we show in the experiments.

Version Consolidation. The final step when processing a mini-batch is to consolidate the distinct sample versions $S_0, S_1, \ldots, S_{M-1}$ into one. Specifically, PARABACUS creates and keeps a final version of the sample that integrates all the M edges in the mini-batch. In Figure 2, the final sample version also incorporates the sample update due to the edge $\{u_M, v_M\}$ into the version S_{M-1}, which will serve as the 0-th version for the next mini-batch.

B. Correctness and Complexity Analysis

Here, we analyze PARABACUS' correctness with respect to the butterfly counts it delivers, and its complexity with respect to both time and space when processing each mini-batch.

Theorem 5 (Correctness)**.** *PARABACUS correctly counts the butterflies in a fully dynamic bipartite stream and provides the same counts as ABACUS after processing each mini-batch.*

Proof Sketch. PARABACUS first sequentially creates all the versions of the sample that an edge would observe in the order of its arrival as in ABACUS. Subsequently, PARABACUS exactly counts the per-edge butterflies formed between each edge in the mini-batch and its corresponding sample version. The partial counts for each edge are the same as in ABACUS. The associativity property of the sum of the counts guarantees that the final refined butterfly count is the same as in ABACUS. Therefore, PARABACUS achieves the same accuracy as ABACUS after processing each mini-batch. □

Consequently, since PARABACUS provides the same butterfly count estimates as ABACUS, its estimates are unbiased as well.

Time Complexity. We now analyze the time complexity of PARABACUS in the worst case as follows:

Theorem 6 (Time Complexity of PARABACUS)**.** *Butterfly counting per mini-batch is performed in $O(M + \frac{Mk^2}{p})$ time, where k is the maximum number of edges maintained in the sample, M is the number of edges in each mini-batch, and p is the number of threads.*

Proof. When processing each mini-batch, PARABACUS sequentially processes all M edges to create and maintain a versioned sample, which takes $O(M)$ time (O(1) for each edge). After constructing the versioned sample, PARABACUS utilizes all p available threads to conduct the per-edge butterfly counting, which takes $O(\frac{Mk^2}{p})$ time. Note that per-edge counting process takes $\Lambda = O(\min\{\sum_{x \in N_u^S} \min\{d_x, d_v\}, \sum_{x \in N_v^S} \min\{d_w, d_u\}\})$ time for an edge $\{u, v\}$. All the vertex degrees are upper-bounded by k, which is the maximum number of edges in the sample, and thus, $\Lambda = O(k^2)$. Therefore, the time complexity for processing each mini-batch is $O(M + \frac{Mk^2}{p})$. □

TABLE II: Datasets Statistics.

Graph	$\lvert E \rvert$	$\lvert L \rvert$	$\lvert R \rvert$	B	**Butterfly Density**
MovieLens	10M	69.8K	10.6K	1.1T	$1.1 * 10^{-16}$
LiveJournal	112M	3.2M	10.7M	3.3T	$2.1 * 10^{-20}$
Trackers	140.6M	27.6M	12.7M	20.0T	$5.1 * 10^{-20}$
Orkut	327M	2.7M	8.73M	22.1T	$1.9 * 10^{-21}$

Comparison with ABACUS. Note that ABACUS takes $O(M + Mk^2) = O(Mk^2)$ time to process a mini-batch with M edges.

Space Complexity. We provide the space analysis in Theorem 7. In a nutshell, PARABACUS needs to store the k elements as a sample of the input graph stream and up to M elements more for the sample versions.

Theorem 7 (Space Complexity of PARABACUS)**.** *Butterfly counting is performed in $O(k + M)$ space, where k is the number of edges maintained in the sample, and M is the number of edges in each mini-batch.*

Proof. The sample has k edges. Since we maintain a versioned sample that stores only the deltas between versions, we have to maintain up to M more edges. Also, we maintain a separate partial count for each edge in the mini-batch; M in total. Therefore, PARABACUS requires $O(k + M)$ space. □

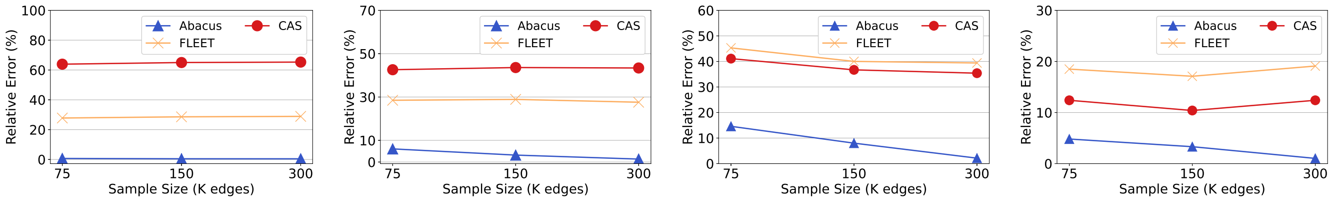
VI. EXPERIMENTAL EVALUATION

We evaluate ABACUS/PARABACUS using four large-scale real-world bipartite graphs and investigate: how effective it is in terms of the error in butterfly estimation; how efficient it is in terms of throughput; how it is affected by the amount of edge deletions; how it scales to large graph streams; and, how much the speedup of its parallel version, PARABACUS, is affected by the mini-batch size and number of threads.

Overall, our major findings include that ABACUS/PARABACUS: (i) achieves significantly higher (up to 148× better) accuracy than the baselines, which inherently cannot handle deletions majorly affecting accuracy, (ii) it has similar throughput with its competitors when processing edge insertions using one thread and much higher throughput in its parallel version when using multiple threads, (iii) it is consistently accurate irrespective of the ratio of deleted edges, (iv) it scales linearly to the number of edges in a graph, (v) PARABACUS accomplishes considerable speedup that depends on the graph characteristics. Finally, our solution delivers unbiased estimates as proved in Section IV.

A. Experimental Setup

Hardware. We ran our experiments on a server with a 10-core Intel(R) Xeon(R) Gold 5115 CPU @ 2.40GHz with 4-way hyper-threading and 188GB of main memory.



(a) *Movielens*

(b) *LiveJournal*

(c) *Trackers*

(d) *Orkut*

Fig. 3: Relative Error of ABACUS with 20% of deletions while varying the sample size of edges. Discarding deletions, as in FLEET and CAS, negatively impacts the accuracy.

Implementation. We implemented ABACUS and our baselines in Java. For the parallel version of ABACUS we used the Callable Java Interface. Note that we store the sampled edges using the adjacency list format.

Datasets. We used four real-world bipartite graphs from the Koblenz Network Collection [1] (KONECT) [34], whose characteristics we show in Table II. ***MovieLens*** contains movie ratings by users. ***LiveJournal*** is a bipartite graph of the LiveJournal social network with users and their group memberships. ***Trackers*** is a bipartite graph of internet domains and trackers, where each edge represents that a tracker is identified by its domains. ***Orkut*** consists of group-user relationships where edges represent the group memberships of users. We preprocessed and converted the graphs to be undirected and unweighted. Also, we removed duplicate edges, self-loops, and zero-degree vertices. In all our datasets, we simulate the stream assuming that an edge arrives at each discrete time $t \geq 1$. All edges arrive in their natural order as in the datasets. *Deletions.* Our real datasets are insertion-only by default, and thus, we generate fully dynamic graph streams by generating the edge deletions from the graphs listed in Table II. In specific, we (a) create the insertions of each edge in the input bipartite graphs using their natural order, (b) create the deletions by selecting α% of the edges from the input bipartite graphs, (c) place each created deletion in a random position after its corresponding insertion. We use $\alpha = 20\%$ as our default value and assess the impact of varying α in Section VI-E. These values stem from [35] which reports up to 30% edge deletions in real-world Twitter graphs.

Baselines. To our knowledge, no existing solution considers estimating butterfly counts on fully dynamic graph streams, entailing both insertions and deletions. Yet, we compare ABACUS/PARABACUS with FLEET [26] and CAS [15], which are the state-of-the-art approaches for insertion-only bipartite graph streams and are the most relevant techniques to our problem. We do so, first, to quantify the effect of disregarding edge deletions on accuracy, and, second, to compare the throughput of ABACUS and its parallel variant, PARABACUS, with that of the best available solutions designed for insertion-only streams. In specific, we use FLEET3, the best method of [26], with a reservoir resizing parameter $\gamma = 0.75$ as proposed, and CAS-R, the best method of [15], with the ratio of memory usage of AMS sketch to total memory equal to $\lambda = 0.33$ as proposed. For PARABACUS, we use a mini-batch size of 500 and 40 threads unless indicated otherwise.

Evaluation Metrics. Let x be the true butterfly count and let $\hat{x}$ be the corresponding estimate obtained by the evaluated algorithm. For evaluating the accuracy of a method, we use the *relative error* metric (the lower the better), which is defined as $\frac{|x-\hat{x}|}{x}$, for a true butterfly count x that is greater than zero.

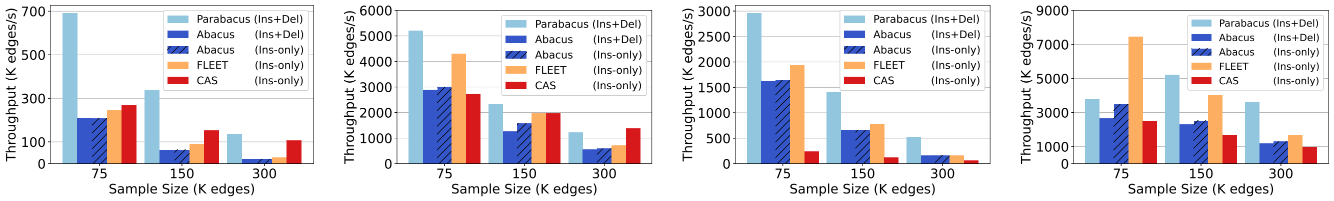
B. Accuracy

We first investigate the accuracy of estimating butterfly counts in the presence of edge deletions. ABACUS and PARABACUS demonstrate the same accuracy and, thus, we denote our solution as ABACUS for simplicity.

We vary the sample size from $75K$ to $300K$ edges. We run each experiment 10 times and show the average relative error values in Figure 3. ABACUS provides $40.6-65.7\times$ more accurate counts than FLEET in *Movielens*, $4.7 - 20.3\times$ in *Livejournal*, $3.8-18.5\times$ in *Trackers*, and $3.2-18.4\times$ in *Orkut*. Also, ABACUS provides $93.4 - 148.4\times$ more accurate counts than CAS in *Movielens*, $7.1 - 31.9\times$ in *Livejournal*, $2.81 - 16.54\times$ in *Trackers*, and $2.57 - 12.03\times$ in *Orkut*.

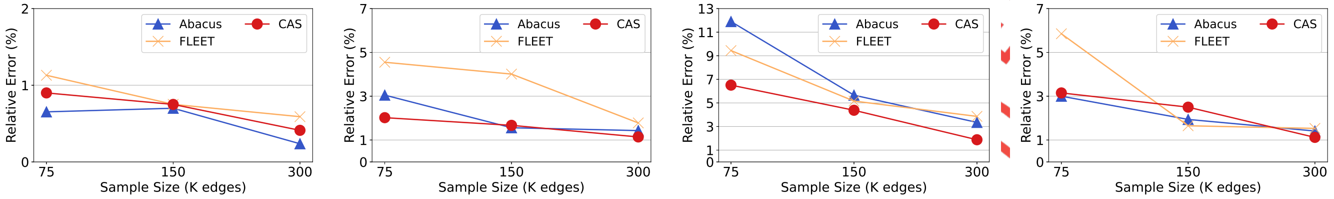
This clearly indicates that edge deletions have a significant impact on butterfly counts if they are ignored, and therefore, it is essential to handle them carefully in order to achieve optimal performance. Furthermore, ABACUS is capable of maintaining quite accurate counts on all datasets and achieves on average 0.52% relative error on *Movielens*, 3.54% on *Livejournal*, 8.25% on *Trackers*, and 3.05% on *Orkut*. Additionally, we observe that the relative error decreases as the sample size increases. For instance, in *Livejournal* ABACUS achieves 6.06% error for a sample size of $75K$ edges, and only 1.36% error when maintaining $300K$ edges. We observe a similar pattern across the remaining datasets. As the sample size grows, ABACUS can more precisely calculate the number of butterflies that each incoming edge forms with the edges in the sample, allowing for a more accurate estimation. This is not the case for the baselines, because they discard the edge deletions. Consequently, their samples are not representative of the fully dynamic graph streams irrespective of the sample size. *Therefore, we conclude that (i) it is imperative to account for edge deletions when estimating butterfly counts, and (ii) ABACUS accurately estimates the butterfly counts in bipartite graph streams containing both edge insertions and deletions.*

[1] http://konect.uni-koblenz.de/



(a) *Movielens* (b) *LiveJournal* (c) *Trackers* (d) *Orkut*

Fig. 4: Throughput for all datasets with 20% of deletions, while varying the sample size of edges. Notably Abacus achieves a similar throughput with the baselines when processing either insertions only (Ins-only) or both insertions and deletions (Ins+Del). ParAbacus achieves a significantly higher throughput for a small mini-batch size of 500 edges.



(a) *Movielens* (b) *LiveJournal* (c) *Trackers* (d) *Orkut*

Fig. 5: Relative Error of Abacus on insertion-only streams ($\alpha = 0\%$), while varying the sample size k of edges maintained.

C. Throughput

We now compare Abacus/ParAbacus with FLEET and CAS in terms of throughput, i.e., the number of edges processed per second. For both FLEET and CAS, we set the reservoir size equal to the sample size in Abacus/ParAbacus. For calculating the throughput, we measure the running time of each method independently of the ingestion rate of the input graph stream ignoring the waiting time for each edge's arrival.

Figure 4 illustrates the throughput that Abacus, ParAbacus, FLEET, and CAS achieve when processing input graph streams with insertions and deletions ($\alpha = 20\%$). For Abacus/ParAbacus, we show the throughput it achieves for processing both insertions and deletions. For a fair comparison with the baselines that do not support deletions, we also show the throughput of Abacus for processing the insertions only (Ins-only). In general, we observe that Abacus achieves a throughput close to that of FLEET and CAS, not only in the case of insertions-only but also in the case where Abacus handles deletions. In addition, we see that ParAbacus significantly enhances the throughput and by far surpasses the baselines in the majority of cases, even with a relatively small mini-batch size of 500 edges. In specific, ParAbacus achieves up to $4.85\times$ higher throughput than FLEET, and up to $12.26\times$ higher throughput than CAS, without sacrificing the accuracy. The throughput enhancement increases when using a larger mini-batch size, as we show later in Section VI-G.

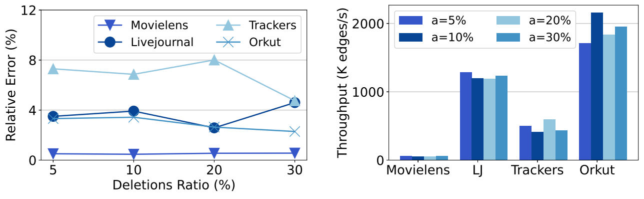
In more detail, we observe that Abacus' throughput when processing insertions is similar to the throughput of FLEET for sample sizes of $150K$ and $300K$ edges. However, for smaller sample sizes such as $75K$ edges, FLEET attains approximately up to $1.5\times$ higher throughput than Abacus. This happens because FLEET always maintains a non-full sample as it resizes its sample and keeps only the 75% of it every time it reaches its maximum capacity. Therefore, the per-edge butterfly counting after each edge's arrival is conducted using a consistently smaller sample than in Abacus. Another reason for this corner case is the low density of *Orkut*, which leads to even less number of butterlies to be formed in the maintained sample. In addition, we see that Abacus achieves a similar throughput to CAS, except in *Trackers* graph where CAS attains a lower throughput. We found out that around half of the time in CAS is attributed to the update of the sketch it reserves. Finally, for all approaches, we see that, in general, the more edges their sample has the lower the achieved throughput. This is reasonable since more work related to butterfly counting is required for processing the same bipartite graph stream.

We conclude that Abacus achieves a throughput very close to that of FLEET and CAS and ParAbacus can achieve an order of magnitude higher throughput even when using a small mini-batch size of 500 edges. Consequently, performance is not sacrificed when processing edge deletions.

D. Accuracy for Insertion-only Streams

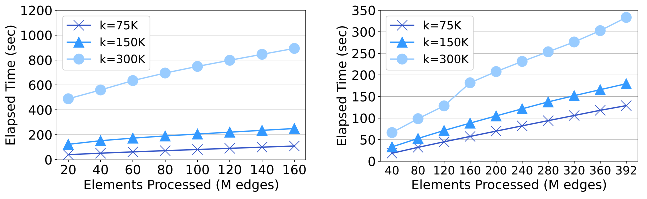
We now compare Abacus with FLEET and CAS in terms of accuracy (i.e., relative error) when processing insertion-only bipartite graph streams, i.e., $\alpha = 0\%$.

Figure 5 shows the accuracy in terms of relative error that Abacus, FLEET, and CAS achieve over bipartite graph streams that contain no deletions. We vary the sample size from $75K$ to $300K$ edges. We run each experiment 10 times and show the average relative error values in Figure 5. We observe that Abacus maintains accuracy comparable to that



(a) Relative Error (b) Throughput

Fig. 6: Impact of deletions α on accuracy and throughput.



(a) *Trackers* (b) *Orkut*

Fig. 7: ABACUS scales linearly with the input stream size.

of FLEET, and is even more accurate in *Movielens* and *Livejournal* bipartite graph streams. We attribute this to the fact that ABACUS maintains a sample that has a maximum size equal to its memory budget and always strives to keep its sample full, whereas FLEET resizes its sample every time it becomes full and keeps only 75% of the edges it contains. Furthermore, ABACUS achieves similar accuracy to CAS indicating that it does not exhibit deficiencies in the absence of deletions. In addition, we observe that the relative error decreases as the sample size increases. For instance, ABACUS achieves 11.9% relative error in *Trackers* for a sample size of $75K$ edges, and only 3.35% error when maintaining $300K$ edges in its sample. We observe a similar trend in the rest of the datasets. This holds for ABACUS, FLEET, and CAS because the more edges stored in their sample, the more precisely they estimate the butterfly counts. *We conclude that* ABACUS *provides butterfly count estimations on insertion-only streams that are at least as accurate as the methods designed specifically for processing insertion-only streams.*

E. Impact of Deletions

We proceed in exploring the impact of deletions ratio, α, on the accuracy (i.e., relative error) and throughput (i.e., number of edges processed per second) of our approach. We use a sample size of $150K$ edges and vary α from 5% up to 30%.

Figure 6a illustrates the relative error for the butterfly count that ABACUS entails when varying the actual ratio of deletions α. We observe that ABACUS produces relatively accurate butterfly counts by maintaining a sample of only $150K$ edges irrespective of the size of the graph stream. Specifically, the relative error in all of our datasets is less than 8%. Furthermore, we observe that the relative error of ABACUS is consistent across datasets and is unaffected by the ratio α, indicating that ABACUS maintains a small error in the presence of deletions regardless of their number. In addition, Figure 6b shows the effect of α on the overall throughput of ABACUS when processing an input bipartite stream with deletions. Note that the bigger the deletions ratio α the more edges exist in a graph stream in total. Despite this, ABACUS maintains a constant throughput for a given dataset, regardless of the deletions ratio α. Also, note that the throughput of ABACUS varies based on the dataset. The graph characteristics, such as the butterfly density of a graph affect the number of butterflies observed after the arrival of each edge and, thus, lead to different throughput for each distinct graph stream. *Therefore, we conclude that* ABACUS *provides consistently accurate butterfly count estimations and maintains steady throughput regardless of the ratio of deletions in the bipartite graph stream.*

F. Scalability

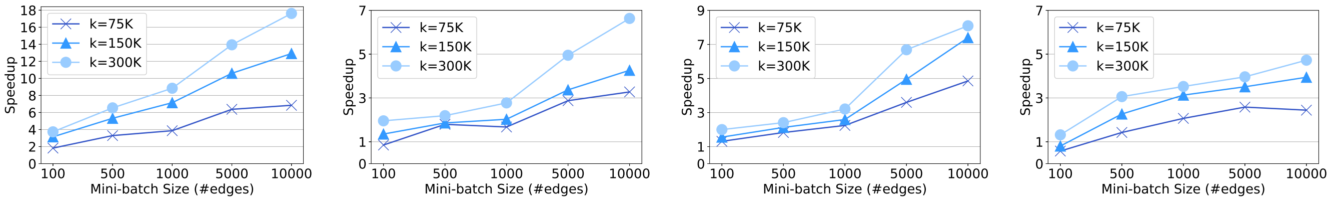
We now demonstrate the scalability of ABACUS with respect to the input graph size. Specifically, we measure the elapsed time that ABACUS needs to fully process bipartite graph streams with varying numbers of edges. Note that we do not consider the waiting time for the arrival of each edge, but we only measure the actual time that ABACUS needs to process the edges of a stream (with a default deletions ratio $\alpha = 20\%$). We use different values for the sample size k, i.e., $75K$, $150K$, and $300K$ edges. We measure the elapsed times each time we process another 10% of edges from the entire graph stream.

Figure 7 illustrates the elapsed time to process the whole graph stream. Specifically, Figure 7a shows that ABACUS scales linearly to the input graph size for the *Trackers* graph. As expected, a larger sample size leads to increased elapsed times; however, the linearity effect is preserved. In addition, we observe a similar scalability trend in the *Orkut* graph stream as shown in Figure 7b. Note that we received linear scalability trends on the other real bipartite graph streams, yet we omit the results for the sake of space. *Therefore, we conclude that* ABACUS *scales linearly to the graph input size, which is in accordance with the Theorem 3.*

G. Parallelization In-depth

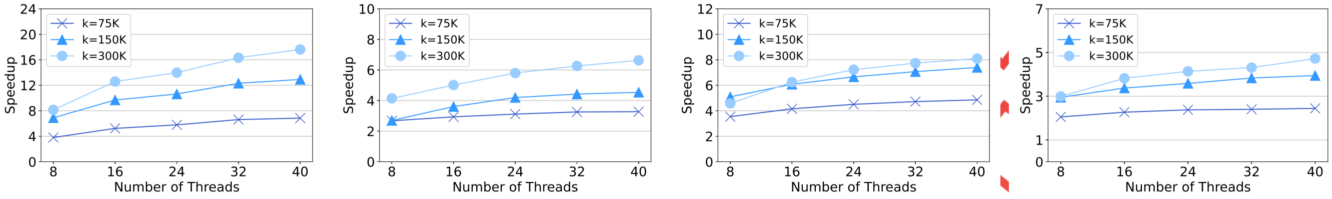
We now analyse in depth the performance of PARABACUS. To this end, we consider the impact of the mini-batch size and the number of threads when processing of the entire bipartite graph stream. Specifically, we measure the speedup in runtime that PARABACUS achieves over ABACUS.

Mini-batch Size. Figure 8 illustrates the speedup that our algorithm achieves when we vary the mini-batch size. We illustrate the speedup for three different sample sizes k, namely, $75K$, $150K$, and $300K$ edges for each dataset. Note that we use all 40 available threads in this experiment. We see that the larger the mini-batch size, the greater the speedup we achieve. When the mini-batch size increases, the work assigned to each thread also increases, and consequently, parallelism is more beneficial. For instance, we see that for a mini-batch size of $10K$ edges, in *Movielens* in Figure 8a we achieve up to $17.6\times$ speedup when the sample size is



(a) *MovieLens* (b) *LiveJournal* (c) *Trackers* (d) *Orkut*

Fig. 8: Speedup of PARABACUS when varying the mini-batch size and using all 40 threads and with a fixed sample size.

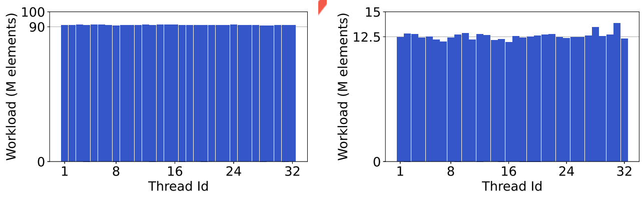


(a) *MovieLens* (b) *LiveJournal* (c) *Trackers* (d) *Orkut*

Fig. 9: Speedup of PARABACUS when varying the number of threads and using mini-batches of $10K$ edges.

$300K$ edges, $12.9\times$ speedup when the sample size is $150K$ edges, and $6.84\times$ speedup when the sample size is $75K$ edges. In Figure 8c on *Trackers* we achieve up to $8.1\times$ speedup when the sample size is $300K$ edges, $7.4\times$ speedup when the sample size is $150K$ edges, and $4.85\times$ speedup when the sample size is $75K$ edges. Interestingly, we observe that the speedup ranges that we achieve differ across different datasets. To validate this empirically, we additionally counted the number of vertices examined due to the set intersection operations for a sample size of 150K for each dataset. We found that the total number of vertices examined was $2.21B$ in *Movielens*, $0.45B$ in *Livejournal*, $0.84B$ in *Trackers*, and $0.30B$ in *Orkut*. This also correlates with the density of butterflies in each dataset, with *Movielens* having the highest density and *Orkut* having the lowest (as shown in Table II). As we maintain uniform random samples, the denser the graph in terms of butterfly containment, the denser the sample, and consequently, more work is done for every set intersection to identify butterflies. Therefore, in graphs with higher density, such as *Movielens*, we observe a relatively higher speedup due to the larger workload assigned to each thread. Conversely, in sparser graphs like *Orkut*, the speedup achieved is still significant but comparatively lower. Also, note that the larger the sample size, the more significant the overall speedup PARABACUS achieves. Parallelism is more beneficial in this case because the set intersection operations related to the per-edge butterfly counting are performed between neighboring sets of a bigger size.

Number of Threads. Figure 9 shows the speedup that PARABACUS achieves when we vary the number of threads for a mini-batch size fixed to $10K$ edges. For each dataset, we illustrate the speedup for three different sample sizes, namely, $75K$, $150K$, and $300K$ edges. We observe that the more threads we utilize, the greater the speedup that PARABACUS attains. Furthermore, as the sample size increases from $75K$ to $300K$ edges, we see the workload increase, and having many threads working in parallel pays off. As shown in Figures 9a-9d, we achieve up to $18\times$ speedup in *Movielens*, up to $6.65\times$ in *Livejournal*, up to $8.1\times$ in *Trackers*, and up to $5\times$ in *Orkut*. Similar to the mini-batch experiments, the observed speedup in our experiment varies across different datasets due to their individual density characteristics. Consequently, the overall computation performed by the set intersections also varies across datasets. Additionally, the larger the sample size the bigger the performance gains are as we increase the number of threads. For instance, in *Movielens* for a sample size of $75K$ edges as we increase the threads from 8 to 40 the speedup we achieve ranges from $3.8 - 6.85\times$, for a sample of $150K$ edges it ranges from $6.9 - 12.91\times$, and for sample of size $300K$ edges it ranges from $8.13 - 17.91\times$.



(a) *Movielens* (b) *Orkut*

Fig. 10: Workload per thread.

Balanced Load. Figure 10 illustrates the workload per thread, i.e., the number of checks that happened within the set intersection operations that take place during the butterfly counting. In this experiment, we use a sample size of 150K edges, a mini-batch size of 10K edges, and set the number of threads to 32. Specifically, in Figure 10a we illustrate the workload per thread for the densest graph, *Movielens*, and in Figure 10b we show the workload per thread for the sparsest graph, *Orkut*, in terms of butterfly density. We observe that all threads are assigned similar workloads, which indicates that the computations of ParAbacus are load-balanced. Additionally, we see that in *Movielens* the average per-thread load is 90M element comparisons, whereas in *Orkut* it is 12.5M element checks. This is in accordance with our previous observation that the work that is needed to process dense graphs is more than that for sparse ones. Same observations hold for the other datasets; however, we do not show the results for the sake of space.

We conclude that ParAbacus *achieves significant speedup using multi-threading, which allows for load-balanced processing of highly volatile bipartite graph streams that may receive thousands of updates per time unit.*

VII. Related Work

A. Triangles in Fully Dynamic Graph Streams

Kutzkov et al. [36] present the first method for counting triangles in fully dynamic graph streams, which adapts colorful triangle sampling [37] to obtain a sparsified graph on which the ratio of two-paths that form triangles is estimated and is afterward scaled to the whole graph. However, [36] is not a real-time streaming algorithm as it only computes an estimate once at the end of the stream, and requires more memory than that for storing the whole input graph in the worst case. Han et al. [38] present ESD, which maintains the current snapshot of a fully dynamic input graph stream. For every incoming edge, ESD tosses a biased coin, and *iff* it lands on heads, it updates the triangle counts by approximating the estimate changes, rather than calculating them precisely. Yet, ESD is not scalable as it has to maintain the whole graph in memory.

Triest$_{FD}$ [33] maintains a uniform sample given a specific memory budget, and derives its estimates by multiplying the triangle counts it obtains from the sampled graph and the reciprocal of the probability that each triangle is sampled. While Triest$_{FD}$ plainly discards the edges that are not sampled without using them for updating its count estimates, ThinkD [13], [14] also leverages the non-sampled edges to update its triangle estimates before discarding them.

B. Butterflies in Static Graphs

Wang et al. [24] present the first technique for exact butterfly counting in static bipartite graphs through wedge enumeration. Sanei-Mehri et al. [25] improve [24] by selecting the cheapest bipartition to traverse when computing exact butterfly counts. Also, the authors proposed randomized algorithms based on sampling and sparsification for computing approximate butterfly counts. Wang et al. [1] propose a vertex ordering-based method, which considers the vertex degrees such that it enumerates fewer wedges throughout the process of butterfly counting. PARBUTTERFLY [39] is a framework that conducts parallel butterfly counting with work-efficient guarantees in static bipartite graphs by exploring various vertex priority functions. Besides vertex ordering, PARBUTTERFLY also offers other type of orderings (or rankings) such as *side*, *approximate degree*, *log-degree*, *degeneracy*, *complement degeneracy*, and *approximate complement degeneracy* orderings. However, adapting the vertex-ordering technique to the streaming setting is infeasible as the priorities continuously change after each incoming edge and sorting becomes a huge overhead. Zhou et al. [40], [41] build on the techniques of [1] and design methods for counting butterflies in uncertain bipartite graphs. Xu et al. [42] propose a GPU-based butterfly counting algorithm that uses an adaptive strategy, which balances the workload among GPU threads for maximizing efficiency. All the aforementioned methods are tailored to static graphs and, thus, are unsuitable for the streaming setting.

C. Butterflies in Insert-Only Graph Streams

Wang et al. [19] extend [1] for dynamic graphs. Similarly, adapting the vertex-ordering approach to the streaming setting is infeasible as the priorities continuously change after each incoming edge and sorting becomes a huge overhead – besides [19] must store the whole graph in main memory, which is prohibitive for streaming algorithms that maintain only a sample of the graph in main memory. Sanei-Mehri et al. [26] propose FLEET, which utilizes adaptive sampling for counting butterflies in bipartite streams using a fixed memory. Li et al [15] present a Co-Affiliation Sampling (CAS) approach, which uses sampling and sketching to provide accurate estimates of butterfly counts in bipartite graph streams. Sheshbolouki et al. [5] propose sGrapp, an approximate adaptive window-based algorithm for counting butterflies after conducting a data-driven empirical analysis to reveal the temporal organizing principles of butterflies in real-time streams. All the above-mentioned streaming methods are tailored to insert-only bipartite graph streams. To the best of our knowledge, Abacus is the first method that provides accurate butterfly counts in fully dynamic bipartite graph streams.

VIII. Conclusions

We proposed Abacus, the first algorithm that estimates butterfly counts in fully dynamic graph streams, which entail both insertions and deletions of edges. We showed that Abacus is: (a) accurate in estimating butterfly counts as it achieves up to $148\times$ smaller error than the baselines; (b), efficient as it performs butterfly counting with similar throughput as the competitors while also attaining linear scalability; and (c) theoretically sound as it consistently and provably provides unbiased estimates of low variance at any time as the input bipartite graph evolves. Additionally, we presented ParAbacus, the parallel version of Abacus, which processes a graph stream in mini-batches and counts butterflies in a load-balanced manner using versioned samples. We showed that ParAbacus achieves considerable speedup and is thus suitable for applications in the streaming setting.

References

[1] K. Wang, X. Lin, L. Qin, W. Zhang, and Y. Zhang, "Vertex priority based butterfly counting for large-scale bipartite networks," *Proc. VLDB Endow.*, vol. 12, no. 10, pp. 1139–1152, 2019.

[2] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou, "Real-time constrained cycle detection in large dynamic graphs," *Proc. VLDB Endow.*, vol. 11, no. 12, pp. 1876–1888, 2018.

[3] Jingren Zhou (Alibaba Group), "Managing, analyzing, and learning heterogeneous graph data: Challenges and opportunities," 2019.

[4] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler, "Practice of streaming processing of dynamic graphs: Concepts, models, and systems," *IEEE Trans. Parallel Distributed Syst.*, vol. 34, no. 6, pp. 1860–1876, 2023.

[5] A. Sheshbolouki and M. T. Özsu, "sgrapp: Butterfly approximation in streaming graphs," *ACM Trans. Knowl. Discov. Data*, vol. 16, no. 4, pp. 76:1–76:43, 2022.

[6] P. G. Lind, M. C. González, and H. J. Herrmann, "Cycles and clustering in bipartite networks," *Physical review E*, vol. 72, no. 5, 2005.

[7] S. Aksoy, T. G. Kolda, and A. Pinar, "Measuring and modeling bipartite graphs with community structure," *J. Complex Networks*, vol. 5, no. 4, pp. 581–603, 2017.

[8] T. Opsahl, "Triadic closure in two-mode networks: Redefining the global and local clustering coefficients," *Soc. Networks*, vol. 35, no. 2, pp. 159–167, 2013.

[9] G. Robins and M. Alexander, "Small worlds among interlocking directors: Network structure and distance in bipartite graphs," *Comput. Math. Organ. Theory*, vol. 10, no. 1, pp. 69–94, 2004.

[10] N. K. Ahmed, N. G. Duffield, and L. Xia, "Sampling for approximate bipartite network projection," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, J. Lang, Ed. ijcai.org, 2018, pp. 3286–3292. [Online]. Available: https://doi.org/10.24963/ijcai.2018/456

[11] P. Jia, P. Wang, J. Tao, and X. Guan, "A fast sketch method for mining user similarities over fully dynamic graph streams," in *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 2019, pp. 1682–1685. [Online]. Available: https://doi.org/10.1109/ICDE.2019.00172

[12] D. Militaru and C. Zaharia, "A survey of collaborative filtering-based systems for online recommendation," in *Proceedings of the 12th International Conference on Electronic Commerce - Roadmap for the Future of Electronic Business, ICEC 2010, Honolulu, Hawaii, USA, August 2-4, 2010*, T. Bui, M. Jarke, V. Dhar, K. L. Hui, and H. Krcmar, Eds. ACM, 2010, pp. 43–47. [Online]. Available: https://doi.org/10.1145/2389376.2389383

[13] K. Shin, J. Kim, B. Hooi, and C. Faloutsos, "Think before you discard: Accurate triangle counting in graph streams with deletions," in *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2018, Dublin, Ireland, September 10-14, 2018, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 11052. Springer, 2018, pp. 141–157.

[14] K. Shin, S. Oh, J. Kim, B. Hooi, and C. Faloutsos, "Fast, accurate and provable triangle counting in fully dynamic graph streams," *ACM Trans. Knowl. Discov. Data*, vol. 14, no. 2, pp. 12:1–12:39, 2020.

[15] R. Li, P. Wang, P. Jia, X. Zhang, J. Zhao, J. Tao, Y. Yuan, and X. Guan, "Approximately counting butterflies in large bipartite graph streams," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 12, pp. 5621–5635, 2022.

[16] S. Bhatia, M. Wadhwa, K. Kawaguchi, N. Shah, P. S. Yu, and B. Hooi, "Sketch-based anomaly detection in streaming graphs," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023, Long Beach, CA, USA, August 6-10, 2023*, A. K. Singh, Y. Sun, L. Akoglu, D. Gunopulos, X. Yan, R. Kumar, F. Ozcan, and J. Ye, Eds. ACM, 2023, pp. 93–104. [Online]. Available: https://doi.org/10.1145/3580305.3599504

[17] A. E. Sariyüce and A. Pinar, "Peeling bipartite networks for dense subgraph discovery," in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining, WSDM 2018, Marina Del Rey, CA, USA, February 5-9, 2018*. ACM, 2018, pp. 504–512.

[18] K. Wang, X. Lin, L. Qin, W. Zhang, and Y. Zhang, "Efficient bitruss decomposition for large-scale bipartite graphs," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 661–672.

[19] K. Wang, X. Lin, L. Qin, W. Zhang, Y. Zhang, and et. al., "Towards efficient solutions of bitruss decomposition for large-scale bipartite graphs," *VLDB J.*, vol. 31, no. 2, pp. 203–226, 2022.

[20] K. Wang, W. Zhang, X. Lin, Y. Zhang, L. Qin, and Y. Zhang, "Efficient and effective community search on large-scale bipartite graphs," in *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 2021, pp. 85–96.

[21] Y. Fang, K. Wang, X. Lin, and W. Zhang, "Cohesive subgraph search over big heterogeneous information networks: Applications, challenges, and solutions," in *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 2021, pp. 2829–2838.

[22] Y. Fang, X. Huang, L. Qin, Y. Zhang, W. Zhang, R. Cheng, and X. Lin, "A survey of community search over big graphs," *VLDB J.*, vol. 29, no. 1, pp. 353–392, 2020.

[23] D. Gibson, R. Kumar, and A. Tomkins, "Discovering large dense subgraphs in massive graphs," in *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. ACM, 2005, pp. 721–732.

[24] J. Wang, A. W. Fu, and J. Cheng, "Rectangle counting in large bipartite graphs," in *2014 IEEE International Congress on Big Data, Anchorage, AK, USA, June 27 - July 2, 2014*. IEEE Computer Society, 2014, pp. 17–24.

[25] S. Sanei-Mehri, A. E. Sariyüce, and S. Tirthapura, "Butterfly counting in bipartite networks," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*. ACM, 2018, pp. 2150–2159.

[26] S. Sanei-Mehri, Y. Zhang, A. E. Sariyüce, and S. Tirthapura, "FLEET: butterfly estimation from a bipartite graph stream," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM 2019, Beijing, China, November 3-7, 2019*. ACM, 2019, pp. 1201–1210.

[27] S. Ranshous, S. Shen, D. Koutra, S. Harenberg, C. Faloutsos, and N. F. Samatova, "Anomaly detection in dynamic networks: a survey," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 7, no. 3, pp. 223–247, 2015.

[28] R. Gemulla, W. Lehner, and P. J. Haas, "Maintaining bounded-size sample synopses of evolving datasets," *VLDB J.*, vol. 17, no. 2, pp. 173–202, 2008.

[29] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010.

[30] A. Koliousis, M. Weidlich, R. C. Fernandez, A. L. Wolf, P. Costa, and P. R. Pietzuch, "SABER: window-based hybrid stream processing for heterogeneous architectures," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 2016, pp. 555–569.

[31] M. N. Garofalakis, J. Gehrke, and R. Rastogi, Eds., *Data Stream Management - Processing High-Speed Data Streams*, ser. Data-Centric Systems and Applications. Springer, 2016.

[32] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, pp. 37–57, 1985.

[33] L. D. Stefani, A. Epasto, M. Riondato, and E. Upfal, "Trièst: Counting local and global triangles in fully dynamic streams with fixed memory size," *ACM Trans. Knowl. Discov. Data*, vol. 11, no. 4, pp. 43:1–43:50, 2017.

[34] J. Kunegis, "KONECT: the koblenz network collection," in *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013, Companion Volume*. International World Wide Web Conferences Steering Committee / ACM, 2013, pp. 1343–1350.

[35] H. Almuhimedi, S. Wilson, B. Liu, N. M. Sadeh, and A. Acquisti, "Tweets are forever: a large-scale quantitative analysis of deleted tweets," in *Computer Supported Cooperative Work, CSCW 2013, San Antonio, TX, USA, February 23-27, 2013*, A. S. Bruckman, S. Counts, C. Lampe, and L. G. Terveen, Eds. ACM, 2013, pp. 897–908. [Online]. Available: https://doi.org/10.1145/2441776.2441878

[36] K. Kutzkov and R. Pagh, "Triangle counting in dynamic graph streams," in *Algorithm Theory - SWAT 2014 - 14th Scandinavian Symposium and Workshops, Copenhagen, Denmark, July 2-4, 2014. Proceedings*, ser. Lecture Notes in Computer Science, vol. 8503. Springer, 2014, pp. 306–318.

[37] R. Pagh and C. E. Tsourakakis, "Colorful triangle counting and a mapreduce implementation," *Inf. Process. Lett.*, vol. 112, no. 7, pp. 277–281, 2012.

[38] G. Han and H. Sethu, "Edge sample and discard: A new algorithm for counting triangles in large dynamic graphs," in *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017, Sydney, Australia, July 31 - August 03, 2017*. ACM, 2017, pp. 44–49.

[39] J. Shi and J. Shun, "Parallel algorithms for butterfly computations," in *1st Symposium on Algorithmic Principles of Computer Systems, APOCS 2020, Salt Lake City, UT, USA, January 8, 2020*, B. M. Maggs, Ed. SIAM, 2020, pp. 16–30.

[40] A. Zhou, Y. Wang, and L. Chen, "Butterfly counting on uncertain bipartite graphs," *Proc. VLDB Endow.*, vol. 15, no. 2, pp. 211–223, 2021.

[41] ——, "Butterfly counting and bitruss decomposition on uncertain bipartite graphs," *VLDB J.*, vol. 32, no. 5, pp. 1013–1036, 2023. [Online]. Available: https://doi.org/10.1007/s00778-023-00782-4

[42] Q. Xu, F. Zhang, Z. Yao, L. Lu, X. Du, D. Deng, and B. He, "Efficient load-balanced butterfly counting on GPU," *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 2450–2462, 2022.

Appendix A
Proof for Lemma 1 (Butt. Discovery Probability)

Proof. The probability $p^{(t)}$ that a specific set of three edges, namely, $e_1 = \{u, v\}$, $e_2 = \{w, x\}$, and $e_3 = \{y, z\}$ being selected in the sample $\mathcal{S}^{(t)}$ can be written using conditional probabilities as $P(e_1) * P(e_2|e_1) * P(e_3|e_1, e_2)$, where $P(e_2|e_1)$ is the probability that edge e_2 exists in the sample given that edge e_1 already exists in the sample. Similarly, $P(e_3|e_1, e_2)$ is the probability that edge e_3 exists in the sample given that the edges e_1 and e_2 exist in the sample. As the sample is uniform, the probability of containing edge e_1 is $P(e_1) = \frac{y}{T}$, where $y = \min(k, |E| + c_g + c_g)$ is the size of the sample, and $T = |E| + c_g + c_g$ is the size of the stream with all the edges in the stream that are not yet deleted. Given that edge e_1 exists in the sample, the probability of edge e_2 existing in the sample is $P(e_2|e_1) = \frac{y-1}{T-1}$ since there are only $T - 1$ edges remaining after edge e_1 is selected, and $y-1$ of them can be selected to form the sample. Similarly, given that edges e_1 and e_2 exist, the probability of edge e_3 existing in the sample is $P(e_3|e_1, e_2) = \frac{y-2}{T-2}$. Therefore, Equation 1 represents the probability of selecting a uniform random sample of y edges from the graph stream containing the edges e_1, e_2, and e_3. □

Appendix B
Proof of Theorem 1 (Unbiasedness)

Proof. Let us assume a butterfly $(\{u, v, w, x\}, s) \in \mathcal{C}^{(t)}$, for an insertion $e^{(s)} = (\{u, v\}, +)$ without loss of generality, where $\mathcal{C}^{(t)}$ is the set of created butterflies at time t or earlier. Furthermore, the increment amount of change, $c^{(s)}_{uvwx}$, in the butterfly count c that is attributed to the creation of the butterfly $(\{u, v, w, x\}, s)$ (Algorithm 1, lines 6,11) is as follows:

$$c^{(s)}_{uvwx} = \begin{cases} \frac{1}{p^{(s-1)}} = \frac{1}{Pr(\{u,x\} \in \mathcal{S}^{(s-1)} \cap \{v,w\} \in \mathcal{S}^{(s-1)} \cap \{w,x\} \in \mathcal{S}^{(s-1)})} \\ \text{, if } \{u, x\}, \{v, w\}, \text{ and } \{w, x\} \in \mathcal{S}^{(s-1)} \\ 0, \text{ otherwise} \end{cases}$$

which holds from Equation 2. Therefore, it trivially holds that $\mathbb{E}(c^{(s)}_{uvwx}) = 1$. Similarly, assume a butterfly $(\{u, v, w, x\}, s) \in \mathcal{D}^{(t)}$, for an edge deletion $e^{(s)} = (\{u, v\}, -)$ without loss of generality, where $\mathcal{D}^{(t)}$ is the set of deleted butterflies at time t or earlier. Furthermore, the amount of change, $d^{(s)}_{uvwx}$, in the butterfly count c that is attributed to the deletion of the butterfly $(\{u, v, w, x\}, s)$ (Algorithm 1, lines 6,11) is as follows:

$$d^{(s)}_{uvwx} = \begin{cases} \frac{-1}{p^{(s-1)}} = \frac{-1}{Pr(\{u,x\} \in \mathcal{S}^{(s-1)} \cap \{v,w\} \in \mathcal{S}^{(s-1)} \cap \{w,x\} \in \mathcal{S}^{(s-1)})} \\ \text{, if } \{u, x\}, \{v, w\}, \text{ and } \{w, x\} \in \mathcal{S}^{(s-1)} \\ 0, \text{ otherwise} \end{cases}$$

which holds from Equation 2. Thus, it holds that $\mathbb{E}(d^{(s)}_{uvwx}) = -1$. Therefore, for the butterfly count, c, it holds that:

$$c^{(t)} = \sum_{(\{u,v,w,x\},s) \in \mathcal{C}^{(t)}} c^{(s)}_{uvwx} + \sum_{(\{u,v,w,x\},s) \in \mathcal{D}^{(t)}} d^{(s)}_{uvwx}$$

By linearity of expectation the following equality holds:

$$\begin{aligned} \mathbb{E}(c^{(t)}) &= \sum_{(\{u,v,w,x\},s) \in \mathcal{C}^{(t)}} \mathbb{E}(c^{(s)}_{uvwx}) + \sum_{(\{u,v,w,x\},s) \in \mathcal{D}^{(t)}} \mathbb{E}(d^{(s)}_{uvwx}) \\ &= \sum_{(\{u,v,w,x\},s) \in \mathcal{C}^{(t)}} (+1) + \sum_{(\{u,v,w,x\},s) \in \mathcal{D}^{(t)}} (-1) \\ &= |\mathcal{C}^{(t)}| - |\mathcal{D}^{(t)}| = |B^{(t)}| \end{aligned} \tag{4}$$

□

Appendix C
Proof of Theorem 2 (Variance)

Proof. As we described in Algorithm 1, ABACUS maintains a uniform random sample at all times, which has size $0 \leq |\mathcal{S}| \leq k$, where k is our memory budget or the maximum sample size. In the beginning of the bipartite graph stream ABACUS's sample is empty and ABACUS keeps the first k edges (attaining zero variance and conducting exact estimation). Later on, ABACUS always maintains a sample that has size k. The only case where the sample size will get smaller than k is when an edge deletion arrives and that edge is specifically stored in $\mathcal{S}$, which is later compensated directly by the subsequent insertions such that the sample size becomes k again. Therefore, we assume that the sample $\mathcal{S}$ has k edges. It shall be clear from the proof that our analysis is robust and would give nearly the same bounds even if $\mathcal{S}$ has size close to k but not exactly k.

Consider a particular moment in time denoted by t, where the sample $\mathcal{S}$ contains k edges. At the given time t, let $|B^{(t)}|$ denotes the true count of butterflies (the ground truth) observed in the graph stream so far, while $|B^{(t)}_{\mathcal{S}}|$ represents the number of butterflies present in the sample. In the sequel, we omit the time t notations from our formulas for the sake of clarify. Figure 11 illustrates an instance of sample $\mathcal{S}$, where the edges that appear in the graph stream are depicted with black color, while the edges that are present in the sample $\mathcal{S}$ are highlighted in red. We see that at time t the true number of butterflies is 3, i.e., B_1, B_2, and B_3, but using the red edges in the sample we count only 1, namely, B_1. Therefore, we observe that through the sample $\mathcal{S}$, ABACUS can either spot a formed butterfly pattern or not. To facilitate our variance analysis, we introduce *indicator random variables* X_i, which equals to 1 if the butterfly $B_i \in \mathcal{S}$, and 0 if $B_i \notin \mathcal{S}$. Subsequently, the total number of butterflies that ABACUS spots in the sample $\mathcal{S}$ is $|B_{\mathcal{S}}| = \sum_i X_i$, whereas after extrapolating (as we described in Algorithm 1) the actual butterfly count that ABACUS returns is $c = \gamma \sum_i X_i$, where γ is equal to $\frac{groundTruth}{groundTruth \times \binom{E-4}{k-4} / \binom{E}{k}} = \binom{E}{k} / \binom{E-4}{k-4}$ since we already proved in Theorem 1 that ABACUS is an unbiased estimator. From the definition of variance, the following equality holds:

$$\begin{aligned} Var[c] &= \mathbb{E}[(c - \mathbb{E}[c])^2] \\ &= \mathbb{E}[c^2] - \mathbb{E}[c]^2 \end{aligned} \tag{5}$$

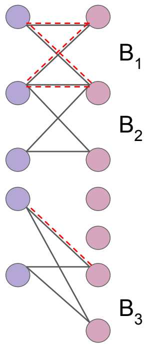


Fig. 11: Instance of the graph stream (black edges) and of the graph sample (red edges) ABACUS maintains at timestamp t.

In Theorem 3, we showed that $\mathbb{E}[c^{(t)}] = \mathbb{E}[c]$ is unbiased. Therefore, we must calculate the second portion of Equation 5. In specific, we unfold the $\mathbb{E}[c]^2$ as follows:

$$\begin{aligned}\mathbb{E}[c^2] &= \gamma^2\mathbb{E}[(\sum_i X_i)^2] \\ &= \gamma(\mathbb{E}[\gamma\sum_i X_i]) + 2\gamma^2(\mathbb{E}[\sum_{i,j} X_iX_j]) \\ &= \gamma(\mathbb{E}[c]) + 2\gamma^2(\mathbb{E}[\sum_{i,j} X_iX_j]) \\ &= \gamma(\mathbb{E}[c]) + 2\gamma^2\sum_{i,j}\mathbb{E}[X_iX_j] \\ &= \gamma(\mathbb{E}[c]) + 2\gamma^2\sum_{i,j} p_{i,j}\end{aligned}$$

where $p_{i,j} = \mathbb{E}[X_iX_j]$ is the probability that the graph sample $\mathcal{S}$ contains both the butterfly B_i and the B_j. In order to calculate the probability $p_{i,j}$ we distinguish the following three cases as shown in Figure 12. Specifically, Figure 12i describes the case where the butterflies B_i and B_j exist in the sample $\mathcal{S}$, but they do not share any edge. Figure 12ii describes the case where the butterflies B_i and B_j exist in the sample $\mathcal{S}$, and they share one edge. Figure 12i describes the case where the butterflies B_i and B_j exist in the sample $\mathcal{S}$, and they share two edges. Notice that it is not possible for two distinct butterflies to share three edges, because then they would be the same butterfly. The probability that ABACUS encounters case i is $\frac{\binom{|E|-8}{k-8}}{\binom{|E|}{k}}$, where $|E|$ is the number of edges that are still valid in the graph stream (have not been deleted), and k is the memory budget of ABACUS that indicates the maximum number of edges in the maintained graph sample $\mathcal{S}$. Similarly, the probability that ABACUS encounters case ii is $\frac{\binom{|E|-7}{k-7}}{\binom{|E|}{k}}$, and the probability that it encounters case iii is $\frac{\binom{|E|-6}{k-6}}{\binom{|E|}{k}}$. The above probabilities stem from the fact that ABACUS always maintains a uniform random sample. Consequently, the probability of the equation above is as follows:

$$\sum_{i,j} p_{i,j} = y_1\frac{\binom{|E|-8}{k-8}}{\binom{|E|}{k}} + y_2\frac{\binom{|E|-7}{k-7}}{\binom{|E|}{k}} + y_3\frac{\binom{|E|-6}{k-6}}{\binom{|E|}{k}}$$

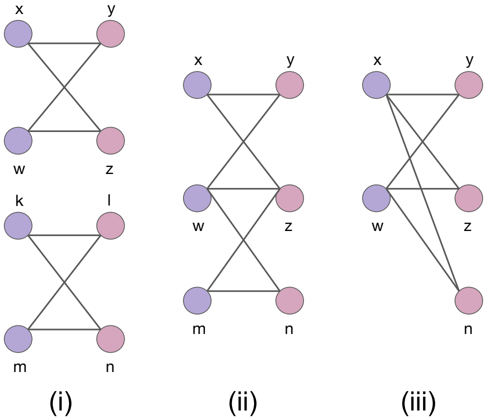


Fig. 12: Cases where the probability $p_{i,j}$ is non-zero.

where y_1, y_2, y_3 indicate how many pairs of butterflies of case i, ii, iii exist in the graph at the current time t (which we omit for simplicity). By combining the two equations above, we get the following closed form for the variance of ABACUS:

$$\begin{aligned}Var[c] &= \mathbb{E}[c^2] - \mathbb{E}[c]^2 \\ &= \gamma\mathbb{E}[c] + 2\gamma^2\sum_{i,j} p_{i,j} - \mathbb{E}[c]^2 \\ &= \gamma\mathbb{E}[c] - \mathbb{E}[c]^2 \\ &+ 2\gamma^2(y_1\frac{\binom{|E|-8}{k-8}}{\binom{|E|}{k}} + y_2\frac{\binom{|E|-7}{k-7}}{\binom{|E|}{k}} + y_3\frac{\binom{|E|-6}{k-6}}{\binom{|E|}{k}})\end{aligned}$$

where by upper bounding $\sum_{i,j} p_{i,j}$ as follows:

$$\sum_{i,j} p_{i,j} \leq (y_1 + y_2 + y_3) \times \frac{\binom{|E|-6}{k-6}}{\binom{|E|}{k}} = \binom{E[c]}{2} \times \frac{\binom{|E|-6}{k-6}}{\binom{|E|}{k}}$$

we can upper bound the the variance as follows:

$$Var[c] \leq \gamma\mathbb{E}[c] + 2\gamma^2\binom{E[c]}{2} \times \frac{\binom{|E|-6}{k-6}}{\binom{|E|}{k}} - \mathbb{E}[c]^2$$

where c is the butterfly count estimation of ABACUS, whose expected value equals the ground truth, $|E|$ is the number of valid edges in the graph stream that have not yet been deleted, and k is the memory budget or equivalently the maximum number of edges in the sample $\mathcal{S}$ that ABACUS maintains. We observe that ABACUS' variance is bounded, and it provably gives accurate estimates. *Tightness.* The upper bound we derived on the variance of ABACUS is *tight*. In specific, consider the 2, 3-bipartite graph that is a clique. There, the variance satisfies the above inequality on the equality condition. □

APPENDIX D
PROOF OF COROLLARY 1 (CONCENTRATION)

Proof. The proof stems directly from applying the Chebysev's inequality since we know the variance (we proved a closed form) and the expected value of ABACUS's estimation. □