ASAP: Fast, Approximate Graph Pattern Mining at Scale

Anand Padmanabha Iyer, *UC Berkeley;* **Zaoxing Liu and Xin Jin,** *Johns Hopkins University;*
Shivaram Venkataraman, *Microsoft Research / University of Wisconsin;*
Vladimir Braverman, *Johns Hopkins University;* **Ion Stoica,** *UC Berkeley*

https://www.usenix.org/conference/osdi18/presentation/iyer

This paper is included in the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-931971-47-8

Open access to the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

ASAP: Fast, Approximate Graph Pattern Mining at Scale

Anand Padmanabha Iyer[⋆*] Zaoxing Liu[†*] Xin Jin[†]
Shivaram Venkataraman[•] Vladimir Braverman[†] Ion Stoica[⋆]
[⋆]*UC Berkeley* [†]*Johns Hopkins University* [•]*Microsoft Research / University of Wisconsin*

Abstract

While there has been a tremendous interest in processing data that has an underlying graph structure, existing distributed graph processing systems take several minutes or even hours to mine simple patterns on graphs. This paper presents ASAP, a fast, approximate computation engine for graph pattern mining. ASAP leverages state-of-the-art results in graph approximation theory, and extends it to general graph patterns in distributed settings. To enable the users to navigate the tradeoff between the result accuracy and latency, we propose a novel approach to build the Error-Latency Profile (ELP) for a given computation. We have implemented ASAP on a general-purpose distributed dataflow platform and evaluated it extensively on several graph patterns. Our experimental results show that ASAP outperforms existing exact pattern mining solutions by up to 77×. Further, ASAP can scale to graphs with billions of edges without the need for large clusters.

1 Introduction

The recent past has seen a resurgence in storing and processing massive amounts of graph-structured data [1, 3]. Algorithms for graph processing can broadly be classified into two categories. The first, *graph analysis* algorithms, compute properties of a graph typically using neighborhood information. Examples of such algorithms include PageRank [46], community detection [31] and label propagation [65]. The second, *graph pattern mining* algorithms, discover structural patterns in a graph. Examples of graph pattern mining algorithms include motif finding [44], frequent sub-graph mining (FSM) [60] and clique mining [19]. Graph mining algorithms are used in applications like detecting similarity between graphlets [49] in social networking and for counting pattern frequencies to do credit card fraud detection.

Today, a deluge of graph processing frameworks exist, both in academia and open-source [20, 24, 25, 34–36, 40, 42, 43, 45, 50, 53, 54, 58, 64]. These frameworks typically provide high-level abstractions that make it easy for developers to implement many graph algorithms. A vast majority of the existing graph processing frameworks however have focused on graph analysis algorithms. These frameworks are fast and can scale out to handle very large graph analysis settings: for instance, GraM [59] can run one iteration of page rank on a trillion-edge graph in 140 seconds in a cluster. In contrast, systems that support graph pattern mining fail to scale to even moderately sized graphs, and are slow, taking several hours to mine simple patterns [29, 55].

The main reason for the lack of the scalability in pattern mining is the underlying complexity of these algorithms—mining patterns requires complex computations and storing exponentially large intermediate candidate sets. For example, a graph with a million vertices may possibly contain 10^{17} triangles. While distributed graph-processing solutions are good candidates for processing such massive intermediate data, the need to do expensive joins to create candidates severely degrades performance. To overcome this, Arabesque [55] proposes new abstractions for graph mining in distributed settings that can significantly optimize how intermediate candidates are stored. However, even with these methods, Arabesque takes over 10 hours to *count* motifs in a graph with less than 1 billion edges.

In this paper, we present ASAP[1], a system that enables both *fast* and *scalable* pattern mining. ASAP is motivated by one key observation: *in many pattern mining tasks, it is often not necessary to output the exact answer*. For instance, in FSM the task is to find the *frequency* of subgraphs with an end-goal of ordering them by occurrences. Similarly, motif counting determines the number of occurrences of a given motif. In these scenarios, it is sufficient to provide an *almost* correct answer. Indeed, our conversations with a social network firm [4] revealed that their application for social graph similarity uses a count of similar graphlets [49]. Another company's [4] fraud detection system similarly counts the frequency of pattern occurrences. In both cases, an approximate count is good enough. Furthermore, it is not necessary to materialize *all* occurrences of a pattern[2]. Based on these use cases, we build a system for *approximate* graph pattern mining.

*Equal contribution.

[1] for *A Swift Approximate Pattern-miner*

[2] In fact, it may even be infeasible to output all embeddings of a pattern in a large graph.

Approximate analytics is an area that has gathered attention in big data analytics [6, 13, 32], where the goal is to let the user trade-off accuracy for much faster results. The basic idea in approximation systems is to execute the *exact* algorithm on a small portion of the data, referred to as *samples*, and then rely on the statistical properties of these samples to compose partial results and/or error characteristics. The fundamental assumption underlying these systems is that there exists a relationship between the input size and the accuracy of the results which can be inferred. However, this assumption falls apart when applied to graph pattern mining. In particular, running the exact algorithm on a sampled graph may not result in a reduction of runtime or good estimation of error (§ 2.2).

Instead, in ASAP, we leverage graph approximation theory, which has a rich history of proposing approximation algorithms for mining specific patterns such as triangles. ASAP exploits a key idea that approximate pattern mining can be viewed as equivalent to probabilistically sampling random instances of the pattern. Using this as a foundation, ASAP extends the state-of-the-art probabilistic approximation techniques to *general patterns* in a *distributed* setting. This lets ASAP massively parallelize instance sampling and provide a drastic reduction in runtimes while sacrificing a small amount of accuracy. ASAP captures this technique in a simple API that allows users to plugin code to detect a single instance of the pattern and then automatically orchestrates computation while adjusting the error bounds based on the parallelism.

Further, ASAP makes pattern mining practical by supporting predicate matching and introducing caching techniques. In particular, ASAP allows mining for patterns where edges in the pattern satisfy a user-specified property. To further reduce the computation time, ASAP leverages the fact that in several mining tasks, such as motif finding, it is possible to cache partial patterns that are building blocks for many other patterns. Finally, an important problem in any approximation system is in allowing users to navigate the tradeoff between the result accuracy and latency. For this, ASAP presents a novel approach to build the Error-Latency Profile (ELP) for graph mining: it uses a small sample of the graph to obtain necessary information and applies Chernoff bound analysis to estimate the worst-case error profile for the original graph.

The combination of these techniques allows ASAP to outperform Arabesque [55], a state-of-the-art exact pattern mining solution by up to 77× on the LiveJournal graph while incurring less than 5% error. In addition, ASAP can scale to graphs with billions of edges—for instance, ASAP can count all the 6 patterns in 4-motifs on the Twitter (1.5B edges) and UK graph (3.7B edges) in 22 and 47 minutes, respectively, in a 16 machine cluster.

We make the following contributions in this paper:

- We present ASAP, the first system to our knowledge, that does fast, scalable approximate graph pattern mining on large graphs. (§3)
- We develop a general API that allows users to mine any graph pattern and present techniques to automatically distribute executions on a cluster. (§4)
- We propose techniques that quickly infer the relationship between approximation error and latency, and show that it is accurate across many real-world graphs. (§5)
- We show that ASAP handles graphs with billions of edges, a scale that existing systems failed to reach. (§6)

2 Background & Motivation

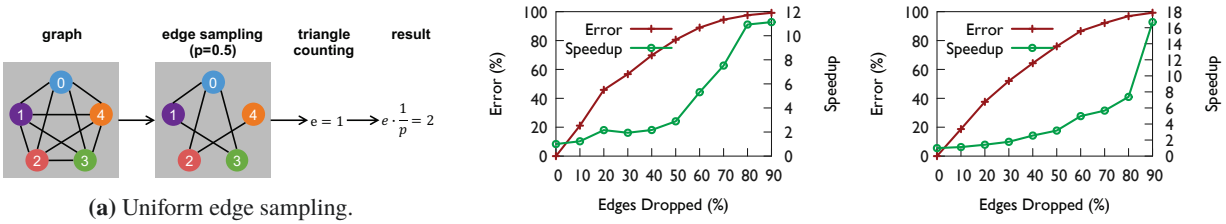
We begin by discussing graph pattern mining algorithms and then motivate the need for a new approach to approximate pattern mining. We then describe recent advancements in graph pattern mining theory that we leverage.

2.1 Graph Pattern Mining

Mining patterns in a graph represent an important class of graph processing problems. Here, the objective is to find instances of a given pattern in a graph or graphs. The common way of representing graph data is in the form of a *property graph* [52], where user-defined properties are attached to the vertices and edges of the graph. A *pattern* is an arbitrary subgraph, and pattern mining algorithms aim to output all subgraphs, commonly referred to as *embeddings*, that match the input pattern. Matching is done via sub-graph isomorphism, which is known to be NP-complete. Several varieties of graph pattern mining problems exist, ranging from finding cliques to mining frequent subgraphs. We refer the reader to [7, 55] for an excellent, in-depth overview of graph mining algorithms.

A common approach to implement pattern mining algorithms is to iterate over all possible embeddings in the graph starting with the simplest pattern (e.g., a vertex or an edge). We can then check all *candidate* embeddings, and prune those that cannot be a part of the final answer. The resulting candidates are then expanded by adding one more vertex/edge, and the process is repeated until it is not possible to explore further. The obvious challenge in graph pattern mining, as opposed to graph analysis, is the exponentially large candidate set that needs to be checked.

Distributed graph processing frameworks are built to process large graphs, and thus seem like an ideal candidate for this problem. Unfortunately when applied to graph mining problems, they face several challenges in managing the candidate set. Arabesque [55], a recently proposed distributed graph mining system, discusses these challenges in detail, and proposes solutions to tackle several of them. However, even Arabesque is unable to scale to large graphs due to the need to materialize candidates and exchange them between machines. As an example, Arabesque takes over 10 hours to count motifs of size 3



(a) Uniform edge sampling.

(b) 3-chains in Twitter graph

(c) Triangles in UK graph

Figure 1: Simply extending approximate processing techniques to graph pattern mining does not work.

in a graph with less than a billion edges on a cluster of 20 machines, each having 256GB of memory.

2.2 Approximate Pattern Mining

Approximate processing is an approach that has been used with tremendous success in solving similar problems in both the big data analytics [6, 32] and databases [22, 26, 27], and thus it is natural to explore similar techniques for graph pattern mining. However, simply extending existing approaches to graphs is insufficient.

The common underlying idea in approximate processing systems is to *sample the input* that a query or an algorithm works on. Several techniques for sampling the input exists, for instance, BlinkDB [6] leverages stratified sampling. To estimate the error, approximation systems rely on the assumption that the sample size relates to the error in the output (e.g., if we sample K items from the original input, then the error in aggregate queries, such as `SUM`, is inversely proportional to $\sqrt{K}$). It is straightforward to envision extending this approach to graph pattern mining—given a graph and a pattern to mine in the graph, we first sample the graph, and run the pattern mining algorithm on the sampled graph.

Figure 1a depicts the idea as applied to triangle counting. In this example, the input graph consists of 10 triangles. Using uniform sampling on the edges we obtain a graph with 50% of the edges. We can then apply triangle counting on this sample to get an answer 1. To scale this number to the actual graph, we can use several ways. One naive way is to double it, since we reduced the input by half. To verify the validity of the approach, we evaluated it on the Twitter graph [39] for finding 3-chains and the UK webgraph [17] graph for triangle counting. The relation between the sample size, error and the speedup compared to running on the original graph ($\frac{T_{orig}}{T_{sample}}$) is shown in figs. 1b and 1c respectively.

These results show the fundamental limitations of the approach. We see that there is no relation between the size of the graph (sample) and the error or the speedup. Even very small samples do not provide noticeable speedups, and conversely, even very large samples end up with significant errors. We conclude that the existing approximation approach of *running the exact algorithm on one or more samples of the input is incompatible with graph pattern mining*. Thus, in this paper, we propose a new approach.

2.3 Graph Pattern Mining Theory

Graph theory community has spent significant efforts in studying various approximation techniques for *specific patterns*. The key idea in these approaches is to model the edges in the graph as a *stream* and *sample instances of a pattern* from the edge stream. Then the *probability of sampling* is used to bound the number of occurrences of the pattern. There has been a large body of theoretical work on various algorithms to sample specific patterns and analysis to prove their bounds [8, 11, 21, 38, 47, 48, 56].

While the intuition of using such sampling to approximate pattern counts is straightforward, the technical details and the analysis are quite subtle. Since sampling once results in a large variance in the estimate, multiple rounds are required to bound the variance. Consider triangle counting as an example. Naively, one would design an technique that uniformly samples three edges from the graph without replacement. Since the probability of sampling one edge is $1/m$ in a graph of m edges, the probability of sampling three edges is $1/m^3$. If the sampled three edges form a triangle, we estimate the number of triangles to be m^3 (the expectation); otherwise, the estimation is 0. While such a sampling technique is unbiased, since m is large in practice, the probability that the sampling would find a triangle is very low and the variance of the result is very large. Obtaining an approximated count with high accuracy, would require a large number of trials, which not only consumes time but also memory.

Neighborhood sampling [48] is a recently proposed approach that provides a solution to this problem in the context of a specific graph pattern, triangle counting. The basic idea is to sample one edge and then gradually add more edges until the edges form a triangle or it becomes impossible to form the pattern. This can be analyzed by Bayesian probability [48]. Let's denote E as the event that a pattern is formed, $E_1, E_2, \ldots, E_k$ are the events that edges $e_1, e_2, \ldots, e_k$ are sampled and stored. Thus the probability of a pattern is actually sampled can be calculated as $Pr(E) = Pr(E_1 \cap E_2 \cdots \cap E_k) = Pr(E_1) \times Pr(E_2|E_1) \cdots \times Pr(E_k|E_1, \ldots, E_{k-1})$. Intuitively, compared to the naive sampling, neighborhood sampling

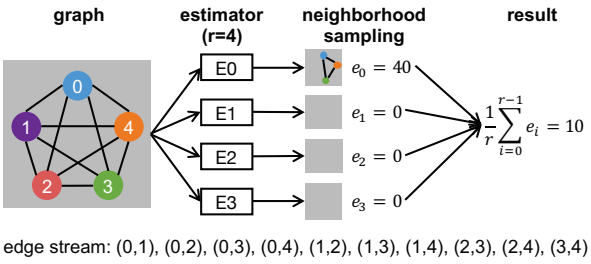


Figure 2: Triangle count by neighborhood sampling

increases the probability that each trial would find an instance of the given pattern, and thus requires fewer estimations to achieve the same accuracy.

2.3.1 Example: Triangle Counting

To illustrate neighborhood sampling, we will revisit the triangle counting example discussed earlier. To sample a triangle from a graph with m edges, we need three edges:

- **First edge l_0.** Uniformly sample one edge from the graph as l_0. The sampling probability $Pr(l_0) = 1/m$.
- **Second edge l_1.** Given that l_0 is already sampled, we uniformly sample one of l_0's adjacent edges (neighbors) from the graph, which we call l_1. Note that neighborhood sampling depends on the ordering of edges in the stream and l_1 appears after l_0 here. The sampling probability $Pr(l_1|l_0) = 1/c$, where c is the number l_0's neighbors appearing after l_0.
- **Third edge l_2.** Find l_2 to finish if edges l_2, l_1, l_0 form a triangle and l_2 appears after l_1 in the stream. If such a triangle is sampled, the sampling probability is $Pr(l_0 \cap l_1 \cap l_2) = Pr(l_0) \times Pr(l_1|l_0) \times Pr(l_2|l_0, l_1) = 1/mc$.

The above technique describes the behaviors of one sampling trial. For each trial, if it successfully samples a triangle, converting probabilities to expectation, $e_i = mc$ will be the estimate of the triangles in the graph. For a total of r trials, $\frac{1}{r}\sum_r e_i$ is output as the approximate result. Figure 2 presents an example of a graph with five nodes.

2.4 Challenges

While the neighborhood sampling algorithm described above has good theoretical properties, there are a number of challenges in building a general system for large-scale approximate graph mining. First, neighborhood sampling was proposed in the context of a specific graph pattern (triangle counting). Therefore, to be of practical use, ASAP needs to generalize neighborhood sampling to other patterns. Second, neighborhood sampling and its analysis assume that the graph is stored in a single machine. ASAP focuses on large-scale, distributed graph processing, and for this it needs to extend neighborhood sampling to computer clusters. Third, neighborhood sampling assumes homogeneous vertices and edges. Real-world graphs are *property graphs*, and in practice pattern mining queries require *predicate matching* which needs the technique to

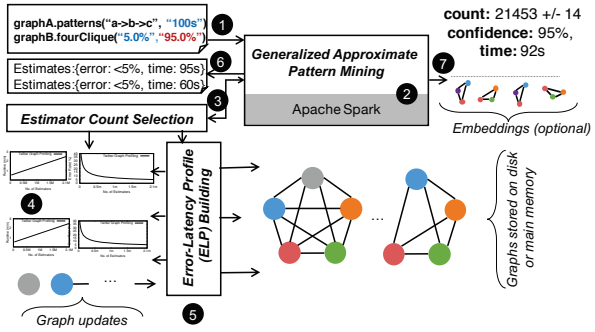


Figure 3: ASAP architecture

be aware of vertex and edge types and properties. Finally, as in any approximate processing system, ASAP needs to allow the end user to trade-off accuracy for latency and hence needs to understand the relation between run-time and error in a distributed setting.

3 ASAP Overview

In this work, we design ASAP, a system that facilitates fast and scalable approximate pattern mining. Figure 3 shows the overall architecture of ASAP. We provide a brief overview of the different components, and how users leverage ASAP to do approximate pattern mining in this section to aid the reader in following the rest of this paper.

User interface. ASAP allows the users to tradeoff accuracy for result latency. Specifically, a user can perform pattern mining tasks using the following two modes ❶:

- **Time budget T.** The user specifies a time budget T, and ASAP returns the most accurate answer within T with a error rate guarantee e and a configurable confidence level (default of 95%).
- **Error budget ϵ.** The user gives an error budget ϵ and confidence level, and ASAP returns an answer within ϵ in the shortest time possible.

Before running the algorithm, ASAP first returns to the user its estimates on the time or error bounds it can achieve ❻. After user approves the estimates, the algorithm is run and the result presented to the user consists of the count, confidence level and the actual run time ❼. Users can also optionally ask to output actual (potentially large number of) embeddings of the pattern found.

Development framework. All pattern mining programs in ASAP are versions of generalized approximate pattern mining ❷ we describe in detail in §4. ASAP provides a standard library of implementations for several common patterns such as triangles, cliques and chains. To allow developers to write program to mine *any* pattern, ASAP further provides a simple API that lets them utilize our approximate mining technique (§ 4.1.2). Using the API, developers simply need to write a program that finds a *single instance* of the pattern they are interested in, which we refer to as *estimator* in the rest of this paper. In a

nutshell, our approximate mining approach depends on running multiple such estimators in parallel.

Error-Latency Profile (ELP). In order to run a user program, ASAP first must find out how many estimators it needs to run for the given bounds ❸. To do this, ASAP builds an ELP. If the ELP is available for a graph, it simply queries the ELP to find the number of estimators ❹. Otherwise, the system builds a new ELP ❺ using a novel technique that is extremely fast and can be done online. We detail our ELP building technique in §5. Since this phase is fast, ASAP can also accommodate graph updates; on large changes, we simply rebuild the ELP.

System runtime. Once ASAP determines the number of estimators necessary to achieve the required error or time bounds, it executes the approximate mining program using a distributed runtime built on Apache Spark [62, 63].

4 Approximate Pattern Mining in ASAP

We now present how ASAP enables large-scale graph pattern mining using neighborhood sampling as a foundation. We first describe our programming abstraction(§ 4.1) that generalizes neighborhood sampling. Then, we describe how ASAP handles errors that arise in distributed processing(§ 4.2). Finally, we show how ASAP can handle queries with predicates on edges or vertices(§ 4.3).

4.1 Extending to General Patterns

To extend the neighborhood sampling technique to general patterns, we leverage one simple observation: at a high level, neighborhood sampling can be viewed as consisting of two phases, *sampling* phase and *closing* phase. In the *sampling* phase, we select an edge in one of two ways by treating the graph as an ordered stream of edges: (a) sample an edge randomly; (b) sample an edge that is adjacent to any previously sampled edges, from the remainder of the stream. In the *closing* phase, we wait for one or more specific edges to complete the pattern.

The probability of sampling a pattern can be computed from these two phases. The closing phase always has a probability of 1 or 0, depending on whether it finds the edges it is waiting for. The probability of the sampling phase depends on how the initial pattern is formed and is a choice made by the developer. For a general graph pattern with multiple nodes, there can be multiple ways to form the pattern. For example, there are two ways to sample a four-clique with different probabilities, as shown in Figure 4. (*i*) In the first case, the sampling phase finds three adjacent edges, and the closing phase waits for rest three edges to come, in order to form the pattern. The sampling probability is $\frac{1}{mc_1c_2}$, where c_1 is the number of the first edge's neighbors and c_2 represents the neighbor count of the first and the second edges. (*ii*) In the second case, the sampling phase finds two disjoint edges, and

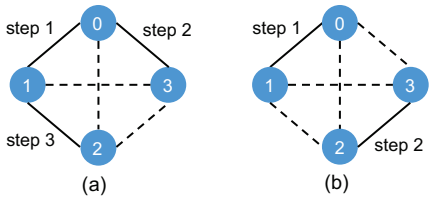


Figure 4: Two ways to sample four cliques. (a) Sample two adjacent edges $(0,1)$ and $(0,3)$, sample another adjacent edge $(1,2)$, and wait for the other three edges. (b) Sample two disjoint edges $(0,1)$ and $(2,3)$, and wait for the other four edges.

the closing phase waits for other four edges to form the pattern. The sampling probability in this case is $\frac{1}{m^2}$.

4.1.1 Analysis of General Patterns

We now show how neighborhood sampling, when captured using the two phases, can extend to general patterns.

Definition 4.1 (General Pattern)**.** *We define a "general pattern" as a set of k connected vertices that form a subgraph in a given graph.*

First, let's consider how an estimator can (possibly) find any general patterns. We show how to sample one general pattern from the graph uniformly with a certain success probability, taking 2 to 5-node patterns as examples. Then, we turn to the problem of maintaining $r \leq 1$ pattern(s) sampled with replacement from the graph. We sample r patterns and a reasonably large r will yield a count estimate with good accuracy. For the convenience of the analysis, we define the following notations: input graph $G = (V, E)$ has m edges and n vertices, and we denote the occurrence of a given pattern in G as $f(G)$. A pattern $p = \{e_i, e_j, \dots\}$ contains a set of ordered edges, i.e., e_i arrives before e_j when $i < j$. When describing the operation of an estimator, $c(e)$ denotes the number of edges adjacent to e and appearing after e, and c_i is $c(e_1, \dots, e_i)$ for any $i \geq 1$. For a given a pattern p^* with k^* vertices, the technique of neighborhood sampling produces p^* with probability $Pr[p = p^*, k = k^*]$. The goal of one estimator is to fix all the vertices that form the pattern, and complete the pattern if possible.

Lemma 4.2. *Let p^* be a k-node pattern in the graph. The probability of detecting the pattern $p = p^*$ depends on k and the different ways to sample using neighborhood sampling technique.*
(1) When $k = 2$, the probability that $p = p^*$ after processing all edges in the graph by all possible neighborhood sampling ways is

$$Pr[p = p^*, k = 2] = \frac{1}{m}$$

(2) When $k = 3$, the probability that $p = p^*$ is

$$Pr[p = p^*, k = 3] = \frac{1}{m \cdot c_1}$$

(3) When $k = 4$, the probability that $p = p^*$ is

$$Pr[p = p^*, k = 4] = \frac{1}{m^2} \text{ (Type-I) or } \frac{1}{m \cdot c_1 \cdot c_2} \text{ (Type-II)}$$

(4) When $k = 5$, the probability that $p = p^*$ is

$$Pr[p = p^*, k = 5] = \frac{1}{m^2 \cdot c_1} \text{ (Type-I)}$$
$$or = \frac{1}{m^2 \cdot c_2} \text{ (Type-II.a)}$$
$$or = \frac{1}{m \cdot c_1 \cdot c_2 \cdot c_3} \text{ (Type-II.b)}$$

Proof. Since a pattern is connected, the operations in the sampling phase are able to reach all nodes in a sampled pattern. To fix such a pattern, the neighborhood sampling needs to confirm all the vertices that form the pattern. Once the vertices are found, the probability of completing such a pattern is fixed.

When $k = 2$, let $p^* = \{e_1\}$ be an edge in the graph. Let $\mathcal{E}_1$ be the event that e_1 is found by neighborhood sampling. There is only one way to fix two vertices of the pattern—uniformly sampling an edge from the graph. By reservoir sampling, we claim that

$$Pr[p = p^*, k = 2] = Pr[\mathcal{E}_1] = \frac{1}{m}$$

When $k = 3$, we need to fix one more vertex beyond the case of $k = 2$. As shown in [48], we need to sample an edge e_2 from e_1's neighbors that occur in the stream after e_1. Let $\mathcal{E}_2$ be the event that e_2 is found. Since $Pr[\mathcal{E}_2|\mathcal{E}_1] = \frac{1}{c(e_1)}$,

$$Pr[p = p^*, k = 3] = Pr[\mathcal{E}_1] \cdot Pr[\mathcal{E}_2|\mathcal{E}_1] = \frac{1}{m \cdot c(e_1)}$$

When $k = 4$, we require one more step from the case of $k = 2$ or the case of $k = 3$, from extending neighborhood sampling. By extending from the case of $k = 2$ (denoted as Type-I), two more vertices are needed to fix a 4-node pattern. In Type-I, we independently find another edge e_2^* that is not adjacent to the sampled edge e_1. Let $\mathcal{E}_2^*$ be the event that e_2^* is found. Since $Pr[\mathcal{E}_2^*|\mathcal{E}_1] = \frac{1}{m}$,

$$Pr[p = p^*, k = 4] = Pr[p = p^*, k = 2] * Pr[\mathcal{E}_2^*|\mathcal{E}_1]$$
$$= \frac{1}{m^2} \text{ (Type-I)}$$

When extending from the case $k = 2$ (denoted as Type-II), one more vertex is needed to fix a 4-node pattern. In Type-II, we sample a "neighbor" e_3 that comes after $e_1 and e_2$. Let $\mathcal{E}_3$ be the event that e_3 is found. Since e_3 is sampled uniformly from the neighbors of e_1 and e_2 and is appearing after e_1, e_2, $Pr[\mathcal{E}_3|\mathcal{E}_1, \mathcal{E}_2] = \frac{1}{c(e_1,e_2)}$. Thus,

$$Pr[p = p^*, k = 4] = Pr[p = p^*, k = 3] \cdot Pr[\mathcal{E}_3|\mathcal{E}_1, \mathcal{E}_2]$$
$$= \frac{1}{m \cdot c(e_1) \cdot c(e_1, e_2)} \text{ (Type-II)}$$

When $k = 5$, we again need one more step from the case $k = 3$ or the case $k = 4$. By extending from $k = 3$ (denoted as Type-I), we require two separate vertices to fix a 5-node pattern. In Type-I, we independently sample another edge e_3^* that is not adjacent to e_1, e_2. Let $\mathcal{E}_3^*$ be the event that e_3^* is found. $Pr[\mathcal{E}_3^*|\mathcal{E}_1, \mathcal{E}_2] = \frac{1}{m}$. Therefore,

$$Pr[p = p^*, k = 5] = Pr[p = p^*, k = 3] * Pr[\mathcal{E}_3^*|\mathcal{E}_1, \mathcal{E}_2]$$
$$= \frac{1}{m^2 \cdot c(e_1)} \text{ (Type-I)}$$

When extending from the case $k = 4$, we need to consider the two types separately. By extending Type-I of case $k = 4$ (denoted as Type-II.a), we need one more vertex to construct a 5-node pattern and thus we sample a neighboring edge e_4. Let $\mathcal{E}_4$ be the event that e_4 is found. Since e_4 is sampled from the neighbors of e_1, e_2,

$$Pr[p = p^*, k = 5] = Pr[p = p^*, k = 4] * Pr[\mathcal{E}_4|\mathcal{E}_1, \mathcal{E}_2^*]$$
$$= \frac{1}{m^2 \cdot c(e_1, e_2)} \text{ (Type-II.a)}$$

Similarly, by extending Type-II of case $k = 4$ (denoted as Type-II.b),

$$Pr[p = p^*, k = 5] = \frac{1}{m \cdot c(e_1) \cdot c(e_1, e_2) \cdot c(e_1, e_2, e_3)}$$

□

Lemma 4.3. *For pattern p^* with k^* nodes, let's define*

$$\tilde{t} = \begin{cases} \frac{1}{Pr[p=p^*, k=k^*]} & \text{if } p \neq \emptyset \\ 0 & \text{if } p = \emptyset \end{cases}$$

Thus, $E[\tilde{t}] = f(G)$.

Proof. By Lemma 4.2, we know that one estimator samples a particular pattern p^* with probability $Pr[p = p^*, k = k^*]$. Let $p(G)$ be the set of a given pattern in the graph,

$$E[\tilde{t}] = \sum_{p^* \in p(G)} \tilde{t}(p \neq \emptyset) \cdot Pr[p = p^*, k = k^*] = |p(G)| = f(G)$$

□

The estimated count is the average of the input of all estimators. Now, we consider how many estimators are needed to maintain an ϵ error guarantee.

Theorem 4.4. *Let $r \geq 1$, $0 < \epsilon \leq 1$, and $0 < \delta \leq 1$. There is an $O(r)$-space bounded algorithm that return an ϵ-approximation to the count of a k-node pattern, with probability at least $1 - \delta$. For a certain ϵ, when $k = 4$, we need $r \geq \frac{C_1 m^2}{f(G)}$ Type-I estimators, or $r \geq \frac{C_2 m \Delta^2}{f(G)}$ Type-II estimators for some constants C_1 and C_2, to achieve ϵ-approximation in the worst case; When $k = 5$, we need $r \geq \frac{C_3 m^2 \Delta}{f(G)}$ Type-I estimators, or $r \geq \frac{C_4 m^2 \Delta}{f(G)}$ Type-II.a estimators, or $r \geq \frac{C_5 m \Delta^3}{f(G)}$ Type-II.b estimators, for some constants C_3, C_4, C_5 in the worst case.*

API	Description
`sampleVertex`: ()→(v,p)	Uniformly sample one vertex from the graph.
`SampleEdge`: ()→(e,p)	Uniformly sample one edge from the graph.
`ConditionalSampleVertex`: (`subgraph`)→(v,p)	Uniformly sample a vertex that appears after a sampled subgraph.
`ConditionalSampleEdge`: (`subgraph`)→(e,p)	Uniformly sample an edge that is adjacent to the given subgraph and comes after the subgraph in the order.
`ConditionalClose`: (`subgraph`, `subgraph`)→$boolean$	Given a sampled subgraph, check if another subgraph that appears later in the order can be formed.

Table 1: ASAP's Approximate Pattern Mining API.

Proof. Let's first consider the case $k = 4$. Let X_i for $i = 1,\ldots,r$ be the output value of i-th estimator. Let $\bar{X} = \frac{1}{r}\sum_{i=1}^{r} X_i$ be the average of r estimators. By Lemma 4.3, we know that $E[X_i] = f(G)$ and $E[\bar{X}] = f(G)$. From the properties of graph G, we have $c(e) \leq \Delta$ for $\forall e \in E$, where Δ is the maximum degree (note that in practice Δ isn't a tight bound for the edge neighbor information). In Type-I, $X_i \leq m^2$ and we construct random variables $Y_i = \frac{X_i}{m^2}$ such that $Y_i = [0,1]$. Let $Y = \sum_{i=1}^{r} Y_i$ and $E[Y] = \frac{f(G)r}{m^2}$. Thus the probability that the estimated number of patterns has a more than ϵ relative error off its expectation $f(G)$ is $Pr[\bar{X} > (1+\epsilon)f(G)] \leq \frac{\delta}{2}$, which is at most

$$Pr[\sum_{i=1}^{r} Y_i > (1+\epsilon)E[Y]] \leq e^{-\frac{\epsilon^2}{2+\epsilon}E[Y]} \leq e^{-\frac{\epsilon^2}{3}E[Y]} \leq \frac{\delta}{2}$$

by Chernoff bound. Thus $r \geq \frac{3m^2}{\epsilon^2 f(G)} \cdot \ln\frac{2}{\delta}$. Similarly, this lower bound of r holds for $Pr[\bar{X} < (1-\epsilon)f(G)]$.

In Type-II, $X_i \leq 6m\Delta^2$. Let $Y_i = \frac{X_i}{6m\Delta^2}$ such that $Y_i = [0,1]$. Let $Y = \sum_{i=1}^{r} Y_i$ and $E[Y] = \frac{f(G)r}{6m\Delta^2}$. By Chernoff bound, $r \geq \frac{18m\Delta^2}{\epsilon^2 f(G)} \cdot \ln(\frac{2}{\delta})$. Similarly, when $k = 5$, we (theoretically) need $\frac{6m^2\Delta}{\epsilon^2 f(G)} \cdot \ln(\frac{2}{\delta})$ Type-I estimators, $\frac{12m^2\Delta}{\epsilon^2 f(G)} \cdot \ln(\frac{2}{\delta})$ Type-II.a estimators, and $\frac{24m\Delta^3}{\epsilon^2 f(G)} \cdot \ln(\frac{2}{\delta})$ Type-II.b estimators. Since each estimator stores $O(1)$ edges, the total memory is $O(r)$. □

4.1.2 Programming API

ASAP automates the process of computing the probability of finding a pattern, and derives an expectation from it by providing a simple API that captures two phases. The API, shown in Table 1, consists of the following five functions:

- `SampleVertex` uniformly samples one vertex from the graph. It takes no input, and outputs v and p, where v is the sampled vertex, and p is the probability that sampled v, which is the inverse of the number of vertices.
- `SampleEdge` uniformly samples one edge from the graph. It also takes no input, and outputs e and p, where e is the sampled edge, and p is the sampling probability, which is the inverse of the number of edges of the graph.
- `ConditionalSampleVertex` conditionally samples one vertex from the graph, given *subgraph* as input. It outputs v and p, where v is the sampled vertex and p is the probability to sample v given that *subgraph* is already sampled.
- `ConditionalSampleEdge`(`subgraph`) conditionally samples one edge adjacent to *subgraph* from the graph, given that *subgraph* is already sampled. It outputs e and p, where e is the sampled edge and p is the probability to sample e given *subgraph*.
- `ConditionalClose`(`subgraph`, `subgraph`) waits for edges that appear after the first *subgraph* to form the second *subgraph*. It takes the two subgraphs as input and outputs *yes*/*no*, which is a boolean value indicating whether the second *subgraph* can be formed. This function is usually used as the final step to sample a pattern where all nodes of a possible instance have been fixed (thereby fixing the edges needed to complete that instance of the pattern) and the sampling process only awaits the additional edges to form the pattern.

These five APIs capture the two phases in neighborhood sampling and can be used to develop pattern mining algorithms. To illustrate the use of these APIs, we describe how they can be used to write two representative graph patterns, shown in Figure 5.

Chain. Using our API to write a sampling function for counting three-node chains is straightforward. It only includes two steps. In the first step, we use `SampleEdge()` to uniformly sample one edge from the graph (line 1). In the second step, given the first sampled edge, we use `ConditionalSampleEdge (subgraph)` to find the second edge of the three-node chain, where `subgraph` is set to be the first sampled edge (line 2). Finally, if the algorithm cannot find e_2 to form a chain with e_1 (line 3), it estimates the number of three-node chains to be 0; otherwise, since the probability to get e_1 and e_2 is $p_1 \cdot p_2$, it estimates the number of chains to be $1/(p_1 \cdot p_2)$.

Four clique. Similarly, we can extend the algorithm of sampling three node chains to sample four cliques. We first sample a three-node chain (line 1-2). Then we sample an adjacent edge of this chain to find the fourth node (line 4). Again, during the three steps, if any edges were not

SampleThreeNodeChain

```
(e1, p1) = SampleEdge()
(e2, p2) = ConditionalSampleEdge(Subgraph(e1))
if (!e2)
  return 0
else
  return 1/(p1.p2)
```

SampleFourCliqueType1

```
(e1, p1) = SampleEdge()
(e2, p2) = ConditionalSampleEdge(Subgraph(e1))
if (!e2) return 0
(e3, p3) = ConditionalSampleEdge(Subgraph(e1, e2))
if (!e3) return 0
subgraph1 = Subgraph(e1,e2,e3)
subgraph2 = FourClique(e1,e2,e3)-subgraph1
if (ConditionalClose(subgraph1,subgraph2))
  return 1/(p1.p2.p3)
else return 0
```

Figure 5: Example approximate pattern mining programs written using ASAP API

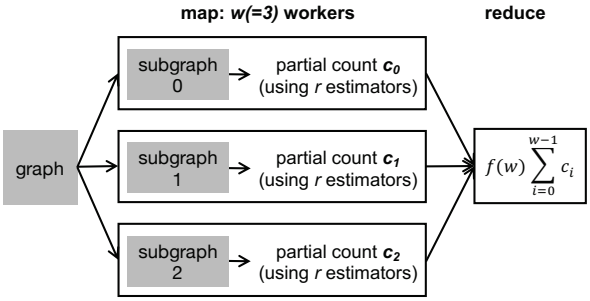


Figure 6: Runtime with graph partition.

sampled, the function would return 0 as no cliques would be found (line 3 and 5). Given e_1, e_2 and e_3, all the four nodes are fixed. Therefore, the function only needs to wait for all edges to form a clique (line 8-9). If the clique is formed, it estimates the number of cliques to be $1/(p_1 \cdot p_2 \cdot p_3)$; otherwise, it returns 0 (line 10). Figure 4(a) illustrates this sampling procedure (CliqueType1).

4.2 Applying to Distributed Settings

Capturing general graph pattern mining using the simple two phase API allows ASAP to extend pattern mining to distributed settings in a seamless fashion. Intuitively, each execution of the user program can be viewed as an instance of the sampling process. To scale this up, ASAP needs to do two things. First, it needs to parallelize the sampling processes, and second, it needs to combine the outputs in a meaningful fashion that preserves the approximation theory.

For parallelizing the pattern mining tasks, ASAP's runtime takes the pattern mining program and wraps it into an *estimator*[3] task. ASAP first partitions the vertices in the graph across machines and executes many copies of the estimator task using standard dataflow operations: *map* and *reduce*. In the map phase, ASAP schedules several copies of the estimator task on each of the machines. Each estimator task operates on the local subgraph in each machine and produces an output, which is a partial count. ASAP's runtime ensures that each estimator in a machine sees the graph's edges and vertices in the *same order*, which is important for the sampling process to produce correct results. Note that although every estimator in each partition sees the graph in the same order, there is *no restriction* on what the order might be (e.g., there is no sorting requirement), thus ASAP uses a random ordering which is fast and requires no pre-processing of the graph. Once this is completed, ASAP runs a reduce task to combine the partial counts and obtain the final answer. This is depicted in fig. 6. This massively parallel execution is one of the reasons for huge latency reduction in ASAP. Since the input to the reduce phase is simply an array of numbers, ASAP's shuffle is extremely light-weight, compared to a system that produces exact answers (and needs to exchange intermediate patterns).

Handling Underestimation. Only summing up the partial counts in the reduce phase underestimates the total number of instances, because when vertices are partitioned to the workers, the instances that span across the partitions are not counted. This results in our technique underestimating the results, and makes the theoretical bounds in neighborhood sampling invalid. Thus, ASAP needs to estimate the error incurred due to distributed execution and incorporate that in the total error analysis.

We use probability theory to do this estimation. We enforce that the vertices in the graph are uniformly randomly distributed across the machines. ASAP is not affected by the normal shortcomings of random vertex partitioning [35] as the amount of data communication is independent of partitioning scheme used. In this case random vertex partitioning is in fact simple to implement, and allows us to theoretically analyze the underestimation.

The theoretical proof for handling the underestimation is outside the scope of this paper. Intuitively, we can think of the random vertex partitioning into w workers as uniform vertex coloring from w available colors. Vertices with the same color are at the same worker and each worker estimates patterns locally on its monochromatic vertices. By doing this coloring, the occurrence of a pattern has been reduced by a factor of $1/f(w)$, where f is a function of the number of workers and the pattern. For instance, a locally sampled triangle has three monochromatic vertices and the probability that this happens among all triangles is $1/w^2$. Thus by the linearity of expectation, each such triangle is scaled by $f(w) = w^2$. A rigorous proof on the maximum possible w with small errors (in practice

[3]Since each program is providing an estimate of the final answer.

w can be >> 100), can be shown using concentration bounds and Hajnal-Szemerédi Theorem [47]. Similarly, each monochromatic 4-clique is scaled by $f(w) = w^3$ and $f(w)$ can be computed for any given pattern.

4.3 Advanced Mining Patterns

Predicate Matching. In property graphs, the edges and vertices contain properties; and thus many real-world mining queries require that matching patterns satisfy some predicates. For example, a *predicate query* might ask for the count of all four cliques on the graph where every vertex in the clique is of a certain type. ASAP supports two types of predicates on the pattern's vertices and edges **all** and **atleast-one**.

For "all" predicate, queries specify a predicate that is applied to *every vertex or edge*. For example, such query may ask for "four cliques where all vertices have a weight of atleast 10". To execute such queries, ASAP introduces a *filtering* phase where the predicate condition is applied before the execution of the pattern mining task. This results in a new graph which consists only of vertices and edges that satisfy the predicate. On this new graph, ASAP runs the pattern mining algorithm. Thus, the "all" predicate query does not require any changes to ASAP's pattern mining algorithm.

The "atleast-one" predicate allows specifying a condition that *atleast* one of the vertices or edges in the pattern satisfies. An example of such a query is "four cliques where atleast one edge has a weight of 10". To execute such predicate queries, we modify the execution to take two passes on the edge list. In the first pass, edges that match the predicate are copied from the `original` edge list to a `matched` edge list. Every entry in the `matched` list is a tuple, `(edge, pos)`, where `pos` is the position in the original list where the matched edge appears. In the second pass, every estimator picks the first edge randomly from the `matched` list. This ensures that the pattern found by the estimator (if it finds one) satisfies the predicate. For the second edge onwards, the estimator uses the `original` list but starts the search from the position at which the first matched edge was found. This ensures that ASAP's probability analysis to estimate the error holds.

Motif mining. Another query used in many real-world workloads is to find all patterns with a certain number of vertices. We define these as *motif queries*; for example a 3-motif query will look for two patterns, triangles and 3-chains. Similarly a 4-motif query looks for six patterns [51]. For motif mining we notice that several patterns have the same underlying *building block*. For example, in 4-motifs, 3-chains are used in many of the constituent patterns. To improve performance, ASAP saves the *sampling* phase's state for the building block pattern. This state includes (i) the currently sampled edges, (ii) the probability of sampling at that point, and (iii) the position in the edge list up to which the estimator has traversed. All the patterns that use this building block are then executed starting from the saved state. This technique can significantly speedup the execution of motif mining queries and we evaluate this in Section 6.2.

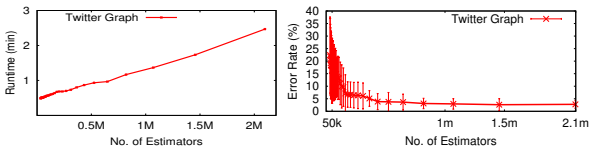


Figure 7: The actual relations between number of estimators and run-time or error rate.

Refining accuracy. In many mining tasks, it is common for the user to first ask for a low accuracy answer, followed by a higher accuracy. For example, users performing exploratory analysis on graph data often would like to iteratively refine the queries. In such settings, ASAP caches the state of the estimator from previous runs. For instance, if a query with an error bound of 10% was executed using 1 million estimators, ASAP saves the output from these estimators. Later, when the same pattern is being queried, but with an error bound of 5% that requires 3 million estimators, ASAP only needs to launch 2 million, and can reuse the first 1 million.

5 Building the Error-Latency Profile (ELP)

A key feature in any approximate processing system is allowing users to trade-off accuracy for result latency. To do this for graph mining, we need to understand the relation between running time and error.

In ASAP's general, distributed graph pattern mining technique described earlier, the only configurable parameter is the number of *estimator* processes used for a mining task. By using r estimators and making r sufficient large, ASAP is able to get results with bounded errors. Since an estimator takes computation and memory resource to sample a pattern, picking the number of estimators r provides a trade-off between result accuracy and resource consumption. In other words, setting a specific number of estimators, N_e, results in a fixed runtime and an error within a certain bound. As an example, fig. 7 depicts the relation between the number of estimators, runtime and error for triangle counting run on the Twitter graph [39]. To enable the user to traverse this trade-off, ASAP needs to determine the correct number of estimators given an error or time budget.

5.1 Building Estimator vs. Time Profile

The time complexity of our approximation algorithm is linearly related to the number of edges in the graph and the number of estimators. Given a graph and a particular pattern, we find the computation time is dominated by the number of estimators when the number of estimators is large enough. From fig. 7, we see that the estimator-time

Algorithm 1 BuildTimeProfile(T^*)

```
1: P ← ∅ // store points for the profile
2: T ← 0, t ← 0, α ← α* // α* can be a reasonable random start
3: while T + t <= T* do
4:     t ← run approximation algorithm with α estimators
5:     P.add((α, t))
6:     α ← 2α
7:     T ← T + t
```

curve is close to linear when the number of estimators is greater than 0.5M. Thus we propose using a linear model to relate the running time to the number of estimators.

When the number of estimators is small, the computation time is also affected by other factorsand thus the curve is not strictly linear. However, for these regions, it is not computationally expensive to profile more exhaustively. Therefore, to build the time profile, we exponentially space our data collection, gathering more points when the number of estimators is small and fewer points as the number of estimators grows. We use a profiling budget T^* to bound the total time spent on profiling. Algorithm 1 shows the pseudo code. ASAP starts from using a small number of estimators ($\alpha \leftarrow \alpha^*$), and doubles α each time until the total profiling time exceeds the profiling cost T^*. In practice, we have found that setting T^* in the minute granularity gives us good results.

5.2 Building Estimator vs. Error Profile

Since error profile is non-linear (fig. 7), techniques like extrapolating from a few data points is not directly applicable. Some recent work has leveraged sophisticated techniques, such as experiment design [57] or Bayesian optimization [12] for the purpose of building non-linear models in the context of instance selection in the cloud. However, these techniques also require the system to compute the error for a given setting for which we need to know the ground-truth, say, by running the exact algorithm on the graph. Not only is this infeasible in many cases, it also undermines the usefulness of an approximation system.

In ASAP, we design a new approach to determine the relationship between the number of estimators N_e and error ϵ. Our approach is based on two main insights: first, we observe that for every pattern based on the probability of sampling, a loose upper bound for the number of estimators required can be computed using Chernoff bounds. For instance for triangle counting, the sampling probability is $1/mc$ where m is the number of edges and c is the degree of first chosen edge(§ 2.3.1). This probability bound can be translated to an estimator of form $N_e > \frac{K*m*\Delta}{\epsilon^2 P}$ (Theorem 3.3 [48]) where K is a constant, m is the number of edges, Δ is the maximum degree and P is the ground truth or the exact number of triangles. At a high level, the bound is based on the fact that the maximum degree vertex leads to the worst case scenario where we have the minimum probability of sampling. Similar bounds exist for 4-cliques and other patterns [48]. These theoretical bounds provide a relation between the number of estimators (N_e), error bound (ϵ) and ground truth (P) in terms of the graph properties such as m and Δ.

The second insight we use is that for smaller graphs we can get a very close approximation to the ground truth by using a very large number of estimators. This is useful in practice as this avoids having to run the exact algorithm to get a good estimate of the ground truth. Based on these two insights, the steps we follow are:

(a) We first uniformly sample the graph by edges to reduce it to a size where we can obtain a nearly 100% accurate result. In our experiments, we find that 5 – 10% of the graph is appropriate according to the size of the graph.
(b) On the sampled graph, we run our algorithm with a large number of estimators (N_{gt}) to find $\hat{P_s}$, a value very close to the ground truth for the sampled graph.
(c) Using $\hat{P_s}$ as the ground truth value and the theoretical relationship described above, we compute the value of other variables on the sampled graph. For example, in the sampled graph, it is easy to compute m_s and Δ_s, and then infer K by running varying number of estimators.
(d) Finally we scale the values m_s, Δ_s and $\hat{P_s}$ to the larger graph to compute N_e. We note that the scaled $\hat{P}$ might not be close to P for the larger graph. But as we use the worst case bound to compute $\hat{P_s}$, the computed value of N_e offers a good bound in practice for the larger graph.

5.3 Handling Evolving Graphs

The ELP building process in ASAP is designed to be fast and scalable. Hence, it is possible to extend our pattern mining technique to evolving graphs [37] by simply rebuilding the ELP every time the graph is updated. However, in practice, we don't need to rebuild the ELP for every update. and that it is possible to reuse an ELP for a limited number of graph changes. Thus we use a simple heuristic where are a fixed number of changes, say 10% of edges, we rebuild the ELP. The general problem of accurately estimating when a profile is incorrect for approximate processing systems is hard [5] and in the future we plan to study if we can automatically determine when to rebuild the ELP by studying changes to the smaller sample graph we use in § 5.2.

6 Evaluation

We evaluate ASAP using a number of real-world graphs and compare it to Arabesque, a state-of-the-art distributed graph mining system. Overall, our evaluations show that:

- Compared to Arabesque, we find ASAP can improve performance by up to 77× with just 5% loss of accuracy for counting 3-motifs and 4-motifs.
- We find that ASAP can also scale to much larger graphs (up to 3.7B edges) whereas existing systems fail to complete execution.

Graph	Nodes	Edges	Degrees
CiteSeer [30]	3,312	4732	2.8
MiCo [30]	100,000	1,080,298	22
Youtube [41]	1,134,890	2,987,624	8
LiveJournal [41]	3,997,962	34,681,189	17
Twitter [39]	41.7 million	1.47 billion	36
Friendster [61]	65.5 million	1.80 billion	28
UK [16, 17]	105.9 million	3.73 billion	35

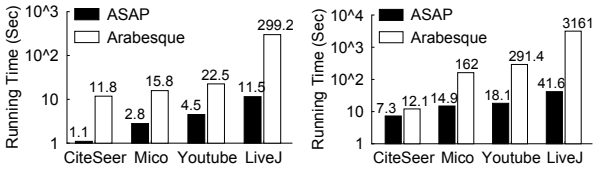
Table 2: Graph datasets used in evaluating ASAP.

- Our techniques to build error profile and time profile (ELP) are highly accurate across all the graphs while finishing within a few minutes.

Implementation. We built ASAP on Apache Spark [63], a general purpose dataflow engine. The implementation uses GraphX [34], the graph processing library of Spark to load and partition the graph. We do not use any other functionality from GraphX, and our techniques only use simple dataflow operators like `map` and `reduce`. As such, ASAP can be implemented on any dataflow engine.

Datasets and Comparisons. Table 2 lists the graphs we use in our experiments. We use 4 small and 3 large graphs and compare ASAP against Arabesque [55] (using its open-source release [2] built on Apache Giraph [14]) on four smaller graphs: CiteSeer [30], Mico [30], Youtube [41], and LiveJournal [41]. For all other evaluations, we use the large graphs. Our experiments were done on a cluster of 16 Amazon EC2 `r4.2xlarge` instances, each with 8 virtual CPUs and 61GiB of memory. While all of these graphs fit in the main memory of a single server, the intermediate state generated (§2) during pattern mining makes it challenging to execute them. Arabesque, despite being a highly optimized distributed solution, fails to scale to the larger graphs in our cluster. We note that Arabesque (or any exact mining system) needs to enumerate the edges significantly more number of times compared to ASAP which only needs to do it once or twice, depending on the query.

Patterns and Metrics. For evaluating ASAP, we use two types of patterns, *motif*s and *clique*s. For motifs, we consider 3-motifs (consisting of 2 individual patterns), and 4-motifs (consisting of 6 individual patterns) and for cliques, we consider 4-cliques. For our experiments, we run 10 trials for each point and report the median, and error bar in the ELP evaluation. We do not include the time to load the graph for any of the experiments for ASAP and Arabesque. We use total runtime as the metric when raw performance is evaluated. When evaluating ASAP on its ability to provide errors within the requested bound, we need to know the *actual* error so that it can be compared with ASAP's output. We compute actual error as $\frac{|t-t_{real}|}{t_{real}}$, where t_{real} is the ground truth number of a specific pattern in a given graph. Since this requires us to know the ground-truth, we use simpler, known patterns, such as triangles and chains, where the ground-truth can be obtained from verified sources for such experiments. Note that the *actual error* is only used for evaluation purposes. Unless otherwise stated, the ASAP evaluations were done with an error target of 5% at 95% confidence.



(a) 3-Motif Counting **(b)** 4-Motif Counting

Figure 8: ASAP is able to gain up to 77× improvement in performance against Arabesque. The gains increase with larger graphs and more complex patterns. Y-axis is in log-scale.

6.1 Overall Performance

We first present the overall performance numbers. To do so, we perform comparisons with Arabesque and evaluate ASAP's scalability on larger graphs. We do not include ELP building time in these numbers since it is a one-time effort for each graph/task and we measure this in § 6.3.

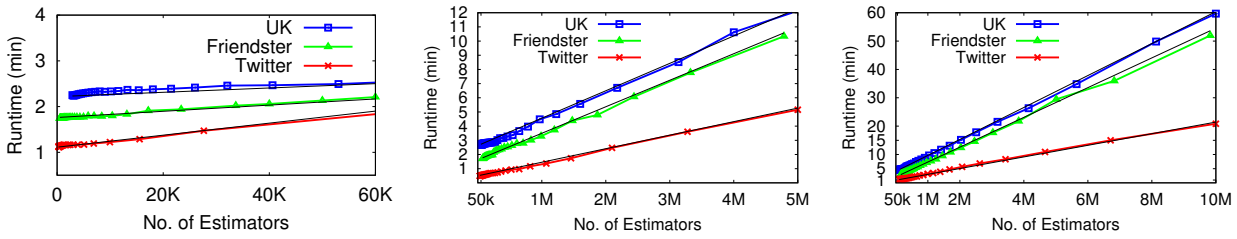
Comparison with Arabesque. In this experiment, we compare Arabesque and ASAP on the 4 smaller graphs (Table 2). In each of these systems, we load the graph first, and then warm up the JVM by running a few test patterns. Then we use each system to perform 3-motif and 4-motif mining, and measure the time taken to complete the task. In Arabesque, we do not consider the time to write the output. Similarly, for ASAP we do not output the patterns embeddings. The results are depicted in figs. 8a and 8b.

We see that ASAP significantly outperforms Arabesque on all the graphs on both the patterns, with performance improvements up to 77× with under 5% loss of accuracy. The performance improvements will increase if the user is able to afford a larger error (e.g., 10%). We also noticed that the performance gap between Arabesque and ASAP increases with larger graph and/or more complex patterns. In this experiment, mining the more complex pattern (4-motif) on the largest graph (LiveJournal) provides the highest gains for ASAP. This validates our choice of using approximation for large-scale pattern mining.

Scalability on Larger Graphs. We repeat the above experiment on the larger graphs. Since Arabesque fails to execute on these graphs on our cluster, we also provide performance numbers that were reported by its authors [55] as a rough comparison. The results are shown in Table 3.

When mining for 3-motif, ASAP performs vastly superior on the Twitter, the Friendster, and the UK graphs. Arabesque's authors report a run time of approximately 11 hours on a graph with a similar number of edges. This translates to a 258× improvement for ASAP. In the case

[4]These graph datasets in Arabesque are not publicly available.



(a) Chain Counting **(b)** Triangle Counting **(c)** Clique Counting

Figure 9: Runtime vs. number of estimators for Twitter, Friendster, and UK graphs. The black solid lines are ASAP's fitted lines.

3-Motif	System	Graph	|V|	|E|	Runtime
ASAP (5%)	16 x 8	Twitter	42M	1.5B	2.5m
	16 x 8	Friendster	66M	1.8B	5.0m
	16 x 8	UK	106M	3.7B	5.9m
Arabesque	20x32	Inst[4]	180M	0.9B	10h45m

4-Motif	System	Graph	|V|	|E|	Runtime
ASAP (5%)	16 x 8	Twitter	42M	1.5B	22m
	16 x 8	UK	106M	3.7B	47m
	16 x 8	LiveJ	**4M**	**34M**	0.7m
Arabesque	16 x 8	LiveJ	**4M**	**34M**	53m
	20x32	SN[4]	**5M**	**199M**	6h18m

Table 3: Comparing the performance of ASAP and Arabesque on large graphs. The System column indicates the number of machines used and the number of cores per machine.

of 4-motifs, ASAP is easily able to scale to the more complex pattern on larger graphs. In comparison, Arabesque is only able to handle a much smaller graph with less than 200 million edges. Even then, it takes over 6 hours to mine all the 4-motif patterns. These results indicate that ASAP is able to not only outperform state-of-the-art solutions significantly, but do so in a much smaller cluster. ASAP is able to effortlessly scale to large graphs.

6.2 Advanced Pattern Mining

We next evaluate the advanced pattern mining capabilities in ASAP described in § 4.3.

Motif mining. We first evaluate the impact of ASAP's optimization when handling motif queries for multiple patterns. We use the Twitter graph and study a 4-motif query that looks for 6 different patterns. In this case ASAP caches the 3-node chain that is shared by multiple patterns. As shown in Table 4, we see a 32% performance improvement from this.

Predicate Matching. To study how well predicate matching queries work, we annotate every edge in the Twitter graph with a randomly chosen property. We then consider a 3-motif query which matches 10% of the edges. With ASAP's filtering based technique, the "all" query completes in 27 seconds, compared to 2.5 minutes when running without pre-filtering.

Accuracy Refinement. We study a scenario where the user first launches a 3-motif query on the Twitter graph with 10% error guarantee and then refines the results with another query that has a 5% error bound. We find that the running time goes from 2.5min to 1.5min (40% improvement) when our caching technique is enabled.

Pattern	Baseline	ASAP	Improv.
Motif Mining	32.2min	22min	32%
Predicate Matching	2.5min	27s	82%
Accuracy Refinement	2.5min	1.5min	40%

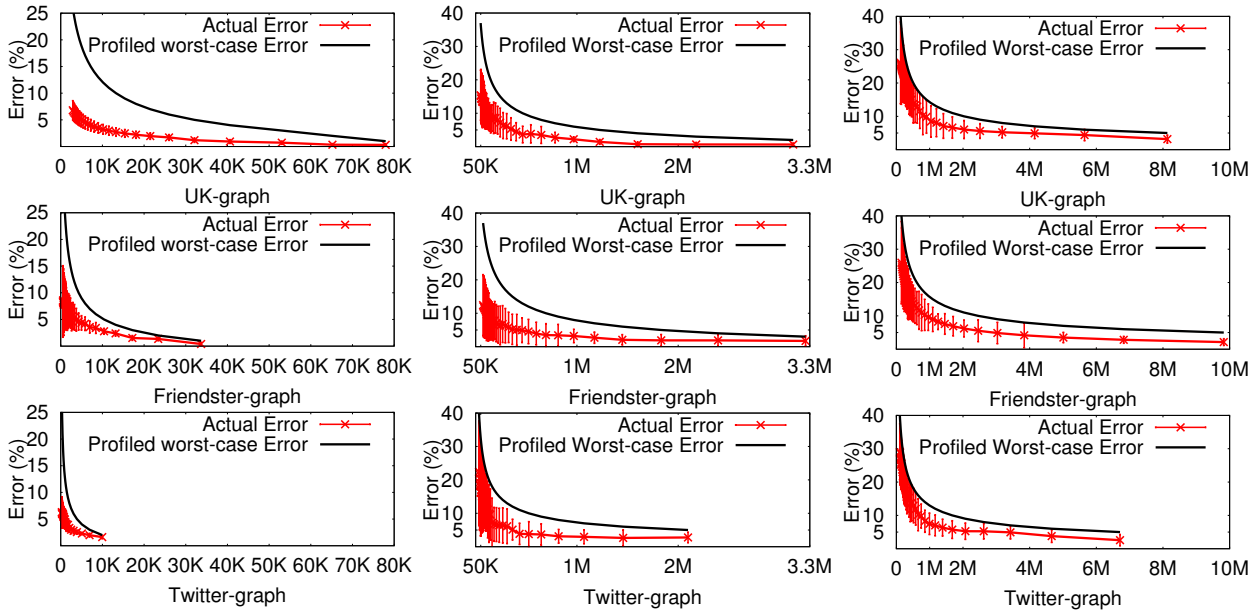
Table 4: Improvements from techniques in ASAP that handle advanced pattern mining queries.

6.3 Effectiveness of ELP Techniques

Here, we evaluate the effectiveness of the ELP building techniques in ASAP, described in §5.

Time Profile. To evaluate how well our time profiling technique (§ 5.1) works, we run three patterns—3-chains, triangles, and 4-cliques—on the three large graphs. In each graph, we obtain the time vs. estimator curve by exhaustively running the mining task with varying number of estimators and noting the time taken to complete the task. We then use our time profiling technique which uses a small number of points instead of exhaustive profiling to obtain ASAP's estimate. We plot both the curves in fig. 9 for each of the three graphs. In these figures, the colored lines represent the actual (exhaustively profiled) curve, and the black line shows ASAP's estimate. From the figure we can see that the time profile estimated by ASAP very closely tracks the actual time taken, thereby showing the effectiveness of our technique.

Error Profile. We repeat the experiment for evaluating ASAP's error profile building technique. Here, we exhaustively build the error profile by running a different number of estimators on each graph, and note the error. Then we use ASAP's technique of using a small portion of the graph to build the profile. We show both in fig. 10. We see that the actual errors are always within the estimated profile. This means that ASAP is able to guarantee that the answer it returns is within the requested error bound. We also note that in real-world graphs, the worst-case bounds are never really reached. In edge cases, where the number of patterns in the graphs are high like the chains in UK graph, the overestimation may be large, and one concern might be that we run more estimators than



(a) Chain Counting **(b)** Triangle Counting **(c)** Clique Counting

Figure 10: Error vs. number of estimators for Twitter, Friendster, and UK graphs.

Graph	Task	Time Profile	Error Profile
UK-2007-05	3-Chain	5.2m	2.1m
	3-Motif	6.1m	2.7m
	4-Clique	9.5m	4.8m
	4-Motif	11.2m	5.9m

Table 5: ELP building time for different tasks on UK graph

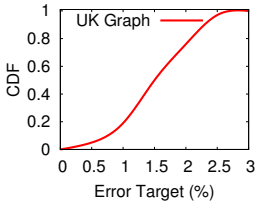


Figure 11: CDF of 100 runs with 3% error target.

required. We are working on techniques that can help us determine a tighter bound for the number of estimators in the future but as discussed in § 6.1, even with this over-estimation we get significant speedups in practice. This experiment confirms that ASAP's heuristic of using a very small portion of the graph and leveraging the Chernoff bound analysis (§ 5.2) is a viable approach.

Error rate Confidence. In Figure 11, we evaluate the cumulative distribution function (CDF) of 100 independent runs on the UK graph with 3% error target and 99% confidence. We can see that 100/100 actual results are not worse than 3% error and 74/100 results are within 2% error. Thus the actual results are even better than the theoretical analysis for 99% confidence.

ELP Building Time. Finally, we evaluate the time taken for building the profiling curves. For this, we use the UK graph and configure ASAP to use 1% of the graph to build the error profile. The results are shown in table 5 for different patterns, which shows that the time to build the profiles is relatively small, even for the largest graph.

6.4 Scaling ASAP on a Cluster

ASAP partitions the graph into different subgraphs based on random vertex partition, and aggregates scaled results in the final reduce phase. In this section we evaluate how configurations with different numbers of machines impact the accuracy. In Fig. 12, we consider two scenarios: *strong-scaling*, where we fix the total number of estimators used for the entire graph, and increase the number of machines used; and *weak-scaling* where we fix the number of estimators used per-machine and thus correspondingly scale the number of estimators as we add more machines. We run the triangle counting task with the Twitter graph on different cluster sizes of 4, 8, 12, and 16 machines. From the figure we see that in the strong-scaling regime, adding more machines has no impact on the accuracy of ASAP and that we are able to correctly adjust the accuracy as more graph partitions are created. In the weak-scaling case we see that the accuracy improves as we increase more machines, which is the expected behavior when we have more estimators.

6.5 More Complex Patterns

Finally, we evaluate the generality of ASAP's techniques by applying to mine 5-motifs, consisting of 21 individual patterns. This choice was influenced by our conversations with industry partners, who use similar patterns in their production systems. Due to the complexity of the patterns, we used a larger cluster for this experiment, consisting

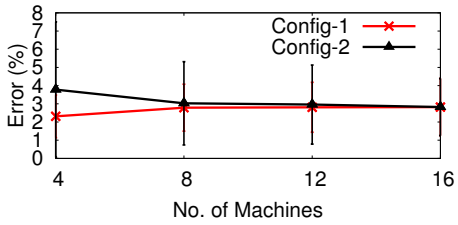


Figure 12: The errors from two cluster scenarios with different number of nodes. Config-1:*strong-scaling* to fix the total number of estimators as 2M × 128; Config-2: *weak-scaling* to fix the number of estimators per executor as 2M.

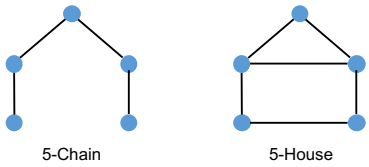


Figure 13: Two representative (from 21) patterns in 5-Motif.

of 16 machines, each with 16 cores and 128GB memory. Due to space constraints, and also because of the absence of a comparison, we only provide ASAP's performance on two representative patterns in table 6. As we see, ASAP is able to handle complex patterns on large graphs easily.

7 Related Work

A large number of systems have been proposed in the literature for **graph processing** [20, 23, 34, 35, 40, 42, 50, 53, 54, 58, 64]. Of these, some [40, 42, 54] are single machine systems, while the rest supports distributed processing. By using careful and optimized operations, these systems can process huge graphs, in the order of a trillion edges. However, these systems have focused their attention mainly on *graph analysis*, and do not support efficient graph pattern mining. Some systems implement very specific versions of simple pattern mining (e.g., triangle count). They do not support general pattern mining.

Similar to graph processing systems, a number of **graph mining** systems have also been proposed. Here too, the proposals contain a mix of centralized systems and distributed systems. These proposals can be classified into two categories. The first category focuses on mining patterns in an input consisting of multiple small graphs. This problem is significantly easier, since the system only finds one instance of the pattern in the graph, and is trivially incorporated in ASAP. Since this approach can be massively parallelized, several distributed systems exist that focus specifically on this problem. The state-of-the-art in distributed, general purpose pattern mining systems is Arabesque [55]. While it supports efficient pattern mining, the system still requires a significant amount of time to process even moderately sized graphs. A few distributed systems have focused on providing approximate pattern mining. However, these systems focus on a specific algorithm, and hence are not general-purpose.

5-Chain	System	Graph	|V|	|E|	Runtime
ASAP (5%)	16 x 16	Twitter	42M	1.5B	9.2m
	16 x 16	UK	106M	3.7B	17.3m
ASAP (10%)	16 x 16	Twitter	42M	1.5B	3.2m
	16 x 16	UK	106M	3.7B	6.5m
5-House	System	Graph	|V|	|E|	Runtime
ASAP (5%)	16 x 16	Twitter	42M	1.5B	12.3m
	16 x 16	UK	106M	3.7B	22.1m
ASAP (10%)	16 x 16	Twitter	42M	1.5B	5.6m
	16 x 16	UK	106M	3.7B	14.2m

Table 6: Approximating 5-Motif patterns in ASAP.

In distributed data processing, **approximate analysis systems** [6, 13, 32] have recently gained popularity due to the time requirements in processing large datasets. Following the approximate query processing theory in the database community, these systems focus on reducing the amount of data used in the analysis process in the hope that the analysis time is also reduced. However, as we show in this work, applying the exact algorithm on a sampled graph does not yield desired results. In addition, doing so complicates, or even makes it infeasible to provide good time or error guarantees.

Theory community has invested a significant amount of time in analyzing and proposing **approximate graph algorithms** for several graph analysis tasks [9, 10, 15, 18, 28, 33]. None of these are aimed at distributed processing, nor do they propose ways to understand the performance profile of the algorithms when deployed in the real world. We leverage this rich theoretical foundation in our work by extending these algorithms to mine general patterns in a distributed setting. We further devise a strategy to build accurate profiles to make the approach practical.

8 Conclusion

We present ASAP, a distributed, sampling-based approximate computation engine for graph pattern mining. ASAP leverages graph approximation theory and extends it to general patterns in a distributed setting. It further employs a novel ELP building technique to allow users to trade-off accuracy for result latency. Our evaluation shows that not only does ASAP outperform state-of-the-art exact solutions by more than a magnitude, but it also scales to large graphs while being low on resource demands.

Acknowledgments We thank our shepherd Roxana Geambasu and the reviewers for their valuable feedback. In addition to NSF CISE Expeditions Award CCF-1730628, this research is supported by gifts from Alibaba, Amazon Web Services, Ant Financial, Arm, CapitalOne, Ericsson, Facebook, Google, Huawei, Intel, Microsoft, Scotiabank, Splunk and VMware. Liu, Braverman, and Jin are supported in part by NSF grants No. 1447639, 1650041, 1652257, 1813487, and CRII-NeTS-1755646, Cisco faculty award, ONR Award N00014-18-1-2364, and a Facebook Communications & Networking Research Award.

References

[1] Enterprise DBMS, Q1 2014. https://www.forrester.com/report/TechRadar+Enterprise+DBMS+Q1+2014/-/E-RES106801.

[2] Graph data mining with arabesque. http://arabesque.io.

[3] Graph DBMS increased their popularity by 500% within the last 2 years. http://db-engines.com/en/blog_post//43.

[4] RISELab Sponsors. https://rise.cs.berkeley.edu/sponsors/, 2018.

[5] Agarwal, S., Milner, H., Kleiner, A., Talwalkar, A., Jordan, M., Madden, S., Mozafari, B., and Stoica, I. Knowing when you're wrong: Building fast and reliable approximate query processing systems. In *Proceedings of SIGMOD '14* (New York, NY, USA), ACM, pp. 481–492.

[6] Agarwal, S., Mozafari, B., Panda, A., Milner, H., Madden, S., and Stoica, I. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 29–42.

[7] Aggarwal, C. C., and Wang, H., Eds. *Managing and Mining Graph Data*, vol. 40 of *Advances in Database Systems*. Springer, 2010.

[8] Ahmed, N. K., Duffield, N., Neville, J., and Kompella, R. Graph sample and hold: A framework for big-graph analytics. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2014), KDD '14, ACM, pp. 1446–1455.

[9] Ahn, K. J., Guha, S., and McGregor, A. Analyzing graph structure via linear measurements. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 2012), SODA '12, Society for Industrial and Applied Mathematics, pp. 459–467.

[10] Ahn, K. J., Guha, S., and McGregor, A. Graph sketches: Sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (New York, NY, USA, 2012), PODS '12, ACM, pp. 5–14.

[11] Al Hasan, M., and Zaki, M. J. Output space sampling for graph patterns. *Proc. VLDB Endow. 2*, 1 (Aug. 2009), 730–741.

[12] Alipourfard, O., Liu, H. H., Chen, J., Venkataraman, S., Yu, M., and Zhang, M. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 469–482.

[13] Ananthanarayanan, G., Hung, M. C.-C., Ren, X., Stoica, I., Wierman, A., and Yu, M. Grass: Trimming stragglers in approximation analytics. In *NSDI* (2014), pp. 289–302.

[14] Apache Giraph. http://giraph.apache.org.

[15] Assadi, S., Khanna, S., and Li, Y. On estimating maximum matching size in graph streams. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 2017), SODA '17, Society for Industrial and Applied Mathematics, pp. 1723–1742.

[16] Boldi, P., Rosa, M., Santini, M., and Vigna, S. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web* (2011), S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, Eds., ACM Press, pp. 587–596.

[17] Boldi, P., and Vigna, S. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)* (Manhattan, USA, 2004), ACM Press, pp. 595–601.

[18] Braverman, V., Ostrovsky, R., and Vilenchik, D. *How Hard Is Counting Triangles in the Streaming Model?* Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 244–254.

[19] Bron, C., and Kerbosch, J. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM 16*, 9 (1973), 575–577.

[20] Buluç, A., and Gilbert, J. R. The combinatorial BLAS: design, implementation, and applications. *IJHPCA 25*, 4 (2011), 496–509.

[21] Buriol, L. S., Frahling, G., Leonardi, S., Marchetti-Spaccamela, A., and Sohler, C. Counting triangles in data streams. In *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (New York, NY, USA, 2006), PODS '06, ACM, pp. 253–262.

[22] Chaudhuri, S., Das, G., and Narasayya, V. Optimized stratified sampling for approximate query processing. *ACM Trans. Database Syst. 32*, 2 (June 2007).

[23] Chen, R., Shi, J., Chen, Y., and Chen, H. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA), EuroSys '15, ACM, pp. 1:1–1:15.

[24] Cheng, R., Hong, J., Kyrola, A., Miao, Y., Weng, X., Wu, M., Yang, F., Zhou, L., Zhao, F., and Chen, E. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 85–98.

[25] Ching, A., Edunov, S., Kabiljo, M., Logothetis, D., and Muthukrishnan, S. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow. 8*, 12 (Aug. 2015), 1804–1815.

[26] Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Gerth, J., Talbot, J., Elmeleegy, K., and Sears, R. Online aggregation and continuous query support in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 1115–1118.

[27] Cormode, G., Garofalakis, M. N., Haas, P. J., and Jermaine, C. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases 4*, 1-3 (2012), 1–294.

[28] Das Sarma, A., Gollapudi, S., and Panigrahy, R. Estimating pagerank on graph streams. In *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (New York, NY, USA, 2008), PODS '08, ACM, pp. 69–78.

[29] Elseidy, M., Abdelhamid, E., Skiadopoulos, S., and Kalnis, P. Grami: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow. 7*, 7 (Mar. 2014), 517–528.

[30] Elseidy, M., Abdelhamid, E., Skiadopoulos, S., and Kalnis, P. Grami: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.* (2014).

[31] Fortunato, S. Community detection in graphs. *Physics reports 486*, 3 (2010), 75–174.

[32] Goiri, I., Bianchini, R., Nagarakatte, S., and Nguyen, T. D. Approxhadoop: Bringing approximations to mapreduce frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015), ASPLOS '15, ACM, pp. 383–397.

[33] Gong, N. Z., Xu, W., Huang, L., Mittal, P., Stefanov, E., Sekar, V., and Song, D. Evolution of social-attribute networks: Measurements, modeling, and implications using google+. In *Proceedings of the 2012 Internet Measurement Conference* (New York, NY, USA, 2012), IMC '12, ACM, pp. 131–144.

[34] Gonzalez, J., Xin, R., Dave, A., Crankshaw, D., and Franklin, Stoica, I. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association.

[35] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 17–30.

[36] Han, W., Miao, Y., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, W., and Chen, E. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 1:1–1:14.

[37] Iyer, A. P., Li, L. E., Das, T., and Stoica, I. Time-evolving graph processing at scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems* (New York, NY, USA, 2016), GRADES '16, ACM, pp. 5:1–5:6.

[38] Jha, M., Seshadhri, C., and Pinar, A. A space-efficient streaming algorithm for estimating transitivity and triangle counts using the birthday paradox. *ACM Trans. Knowl. Discov. Data 9*, 3 (Feb. 2015), 15:1–15:21.

[39] Kwak, H., Lee, C., Park, H., and Moon, S. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web* (New York, NY, USA, 2010), ACM, pp. 591–600.

[40] Kyrola, A., Blelloch, G., and Guestrin, C. Graphchi: Large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, 2012), USENIX, pp. 31–46.

[41] Leskovec, J., and Krevl, A. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[42] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. Graphlab: A new framework for parallel machine learning. In *UAI* (2010), P. Grünwald and P. Spirtes, Eds., AUAI Press, pp. 340–349.

[43] Macko, P., Marathe, V. J., Margo, D. W., and Seltzer, M. I. Llama: Efficient graph analytics using large multiversioned arrays. In *2015 IEEE 31st International Conference on Data Engineering* (April 2015), pp. 363–374.

[44] Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., and Alon, U. Network motifs: simple building blocks of complex networks. *Science 298*, 5594 (2002), 824–827.

[45] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 439–455.

[46] Page, L., Brin, S., Motwani, R., and Winograd, T. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[47] Pagh, R., and Tsourakakis, C. E. Colorful triangle counting and a mapreduce implementation. *CoRR abs/1103.6073* (2011).

[48] Pavan, A., Tangwongsan, K., Tirthapura, S., and Wu, K.-L. Counting and sampling triangles from a graph stream. *Proc. VLDB Endow. 6*, 14 (Sept. 2013), 1870–1881.

[49] Pržulj, N., Corneil, D. G., and Jurisica, I. Modeling interactome: Scale-free or geometric? *Bioinformatics 20*, 18 (Dec. 2004), 3508–3515.

[50] Quamar, A., Deshpande, A., and Lin, J. Nscale: Neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal 25*, 2 (Apr. 2016), 125–150.

[51] Ribeiro, P., and Silva, F. G-tries: A data structure for storing and finding subgraphs. *Data Min. Knowl. Discov. 28*, 2 (Mar. 2014), 337–377.

[52] Robinson, I., Webber, J., and Eifrem, E. *Graph Databases*. O'Reilly Media, Inc., 2013.

[53] Roy, A., Bindschaedler, L., Malicevic, J., and Zwaenepoel, W. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 410–424.

[54] Roy, A., Mihailovic, I., and Zwaenepoel, W. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 472–488.

[55] Teixeira, C. H. C., Fonseca, A. J., Serafini, M., Siganos, G., Zaki, M. J., and Aboulnaga, A. Arabesque: A system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 425–440.

[56] Tsourakakis, C. E., Kang, U., Miller, G. L., and Faloutsos, C. Doulion: Counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2009), KDD '09, ACM, pp. 837–846.

[57] Venkataraman, S., Yang, Z., Franklin, M., Recht, B., and Stoica, I. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 363–378.

[58] Wang, G., Xie, W., Demers, A. J., and Gehrke, J. Asynchronous large-scale graph processing made easy. In *CIDR* (2013).

[59] Wu, M., Yang, F., Xue, J., Xiao, W., Miao, Y., Wei, L., Lin, H., Dai, Y., and Zhou, L. Gram: Scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, ACM, pp. 408–421.

[60] Yan, X., and Han, J. gspan: Graph-based substructure pattern mining. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on* (2002), IEEE, pp. 721–724.

[61] Yang, J., and Leskovec, J. Defining and evaluating network communities based on ground-truth. *CoRR abs/1205.6233* (2012).

[62] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 2–2.

[63] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2010), HotCloud'10, USENIX Association, pp. 10–10.

[64] Zhang, M., Wu, Y., Zhuo, Y., Qian, X., Huan, C., and Chen, K. Wonderland: A novel abstraction-based out-of-core graph processing system. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2018), ASPLOS '18, ACM, pp. 608–621.

[65] Zhu, X., and Ghahramani, Z. Learning from labeled and unlabeled data with label propagation.