Arya: Arbitrary Graph Pattern Mining with Decomposition-based Sampling

Zeying Zhu, *Boston University;* Kan Wu, *University of Wisconsin-Madison;*
Zaoxing Liu, *Boston University*

https://www.usenix.org/conference/nsdi23/presentation/zhu

This paper is included in the
Proceedings of the 20th USENIX Symposium on
Networked Systems Design and Implementation.

April 17–19, 2023 • Boston, MA, USA

978-1-939133-33-5

Open access to the Proceedings of the
20th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Arya: Arbitrary Graph Pattern Mining with Decomposition-based Sampling

Zeying Zhu[⋆*], *Kan Wu*[†*], *Zaoxing Liu*[⋆]
[⋆]*Boston University,* [†]*University of Wisconsin-Madison*

Abstract

Graph pattern mining is compute-intensive in processing massive amounts of graph-structured data. This paper presents Arya, an ultra-fast approximate graph pattern miner that can detect and count arbitrary patterns of a graph. Unlike all prior approximation systems, Arya combines novel graph decomposition theory with edge sampling-based approximation to reduce the complexity of mining complex patterns on graphs with up to tens of billions of edges, a scale that was only possible on supercomputers. Arya can run on a single machine or distributed machines with an Error-Latency Profile (ELP) for users to configure the running time of pattern mining tasks based on different error targets. Our evaluation demonstrates that Arya outperforms existing exact and approximate pattern mining solutions by up to five orders of magnitude. Arya supports graphs with 5 billion edges on a single machine and scales to 10-billion-edge graphs on a 32-server testbed.

1 Introduction

Graph-structured data have been used to represent relationships between entities in various domains, ranging from social networks [23, 39], financial transactions [46, 56], and knowledge bases [4]. There are two main categories of tasks in analyzing graphs: *graph computation* and *graph pattern mining*. Graph computation obtains various properties of a graph, such as PageRank [59] and connected components [41]. Graph pattern mining is more compute-intensive as it discovers structural patterns (i.e., subgraphs), such as motif finding [16, 57], frequent subgraph mining (FSM) [9, 73], and clique counting [21, 40]. These mining tasks are used in various applications, such as counting patterns of financial fraud [1] and detecting suspicious activities on social networks [8].

With graph data reaching multi-billion scales [7, 76], there is an increasing need to mine complex patterns to understand complicated internal relationships [22, 24, 29]. While many graph frameworks have been developed over the years based on various system and algorithmic optimizations, mining complex patterns (e.g., more than 5-vertex) in large graphs remains challenging. The fundamental reason is that pattern mining requires traversing and computing over large intermediate candidate sets, which grow exponentially with the graph size and pattern complexity. For instance, a recent high-performance mining engine, GraphPi [64], needs several hours to mine a six-vertex pattern in a graph of 1.2 billion edges, using the world's top-10 supercomputer with 1024 compute nodes (24,576 cores). Other general-purpose graph mining solutions, such as Peregrine [45] and Fractal [33], need more than a day to mine a six-vertex pattern even in small graphs (e.g., 1 million edges) on a 4-server testbed (see details in §7).

To reduce the underlying mining complexity, sampling-based approximation approaches have been proposed, e.g., ASAP [44] leverages a neighborhood sampling approach to approximate the pattern occurrences. Unfortunately, existing sampling-based approaches come with two common issues: (1) When pattern complexity (the numbers of vertices and edges) grows, the number of required sampling algorithm trials (called samplers) increases significantly (e.g., 10^{15} in mining 5-cliques in a billion-edge graph as shown in §2.1), making it infeasible to mine complex patterns in large graphs. (2) Systems like ASAP require developers to define distinct ways to sample a pattern to cover all possible occurrences. It might be easy for simple patterns like triangles: we can randomly pick the first edge, sample the second edge among the first edge's neighbors, and wait for the third edge to close the triangle. But it is challenging to figure out how to sample complex patterns as there are many distinct sampling ways.

In this paper, we present **Arya**, an approximate mining system that can scale to ultra-large graphs (e.g., 10 billion edges) and mine complex patterns (e.g., 11-vertex). In Arya, we tackle a general approximate mining problem: *Given an input graph, output the approximated occurrences of an arbitrary subgraph*. In many applications, an almost-correct result is sufficient, and the processing time is the key. For instance, a fintech company estimates the frequency of certain complex patterns to quantify the trends of online crime and fraud [24]. The chains of (money laundering) transactions form special patterns and estimating the occurrences of such patterns will help banks and companies to evaluate their operational risks.

Given that designing faster pattern sampling algorithms is theoretically difficult [35], Arya takes a new avenue to ***reduce the complexity of the pattern to be sampled***. Inspired by theory advances in graph sampling [14, 34, 36] and graph decomposition [18], our contribution is to bring theory into practice by an end-to-end system design that explores the algorithmic potential of approximate graph mining (e.g., systems

[*]Equal contribution.

design and optimizations, query accelerations) to meet user requirements (e.g. mining arbitrary patterns and error-latency profiles). Backed by decomposition theory, Arya significantly reduces the inherent approximation complexity of complex patterns in large dense graphs if the pattern can be properly decomposed. Arya has two main components: (1) a pattern decomposer that decomposes a complex pattern into a set of unique simple subpatterns and (2) a parallel estimation engine that generates a number of *samplers* for decomposed subpatterns and constructs an estimated frequency for the original complex pattern.

When designing our pattern decomposer, we need to determine an optimal decomposition of a complex pattern into simpler subgraphs such that it is sufficiently easier to sample the decomposed subpatterns and reconstruct the result for the original pattern. In Arya, we leverage the recent edge-cover-based graph decomposition [18]. The analysis shows that by computing the optimal fractional edge cover of a complex pattern (see §2.2), we can decompose the pattern into a unique collection of unique vertex-disjoint odd cycles and stars, which can be significantly easier to sample than the original pattern. With decomposition, Arya also has unique ways to sample patterns, alleviating the need to explore different sampling methods. Even if a pattern is too simple to be decomposed (e.g., 2-star), Arya performs no worse than existing sampling-based systems.

Once a pattern is decomposed, we build a parallel sampling engine to estimate the pattern occurrence by sampling cycles and stars separately. By extending the edge sampling theory from [15, 18], we build *odd cycle sampler* and *star sampler* with massive parallelism and construct the sampler for the original pattern. Each sampler is essentially a sampling trial aims to find *one instance* of the pattern with a fixed probability p: it merges the sampled odd cycles and stars and tests the remaining edges to find a potential pattern. If the sampled pattern can be formed, the sampler outputs $1/p$; otherwise zero. With a sufficient number of independent samplers, we can obtain an estimated pattern count by averaging the outputs from all samplers (linearity of expectation). To estimate the number of samplers required to achieve the desired accuracy, Arya introduces a heuristic inspired by ASAP [44] to build the Error-Latency Profile (ELP), which takes an error target as the input and infers relevant parameters (e.g., the number of samplers) to configure the graph miner based on bootstrapping (from a small sample of the graph).

With graph decomposition, edge sampling, and a series of system optimizations (e.g., probability-aware scheduling and sampler caching), Arya outperforms *any* existing graph mining systems in scalability. We implement Arya using Memcached to store graph data structures and optimize the most frequent queries to it (e.g., neighbor edges and vertices). We deploy Arya onto three computing scenarios: a single server, a single server with persistent memory, and a cluster with 4 to 32 servers. Our evaluation demonstrates that Arya outperforms GraphPi, a state-of-the-art supercomputer-based graph miner, by up to 5 orders of magnitude while incurring a less than 5% error. In addition, Arya outperforms the state-of-the-art approximate mining system [44] by up to 145× and scales to graphs with multi-billion edges. For example, Arya can mine a complex 7-vertex pattern in a 5-billion-edge synthetic graph in several seconds. We open-source Arya and datasets in https://github.com/Froot-NetSys/Arya.

In this paper, we make the following contributions.

- We present Arya, an approximate graph miner that scales to large graphs and complex patterns, leveraging advanced graph decomposition theory and edge-based sampling. (§3)
- We introduce techniques to quickly sample decomposed subpatterns and reconstruct the original pattern for approximation, based on the latest cycle/star sampling algorithms. (§4)
- We extend Arya to various distributed settings for heterogeneous graph processing scenarios. (§5)
- We show that Arya mines complex patterns in graphs with 5 billion edges using a single machine and 10 billion edges using multiple machines, a scale that even supercomputer-based systems failed to achieve. (§6,§7)

2 Background and Motivation

In this section, we discuss the background of graph pattern mining and approximate mining algorithms. We then describe the graph decomposition theory that we leverage.

2.1 Graph Pattern Mining

Problem Definition. Graph pattern mining is to find instances of a given pattern in a graph or set of graphs. A pattern is an arbitrary subgraph, which represents user-defined properties attached to the edges and vertices. Pattern mining algorithms aim to find all subgraph instances (called *embeddings*) that match a given pattern of interest. Such matching is usually done via iterating all subgraphs that are *isomorphic* to the input pattern, which is known to be NP-complete. At a high level, the compute complexity of an *exact* pattern mining algorithm is associated with the need to iterate over all possible embeddings in the graph.

Approximate Graph Pattern Mining. Given the search complexity of exact mining algorithms, approximation-based approaches become promising. Approximate analytics is widely used in solving complex big data [58], network telemetry [38, 63], and database problems [11], typically with significantly lower resource overheads. A common idea to perform approximation is to sample a *subset* of the input data uniformly at random and perform analytical tasks over the sampled data. For instance, uniform sampling (e.g., NetFlow and sFlow) has been widely used in monitoring network flows.

• **Advanced pattern sampling:** There is a large body of theoretical work on designing sampling algorithms [44, 47, 60] to mine specific patterns such as triangles and cliques. The

main difference between these algorithms is the way to sample a specific pattern. Intuitively, if a sampling approach can sample a pattern with a higher probability, a smaller number of samplers is required to achieve high accuracy (and thus a shorter completion time). For instance, *neighborhood sampling* is used in ASAP [44]. The main idea of neighborhood sampling is to continuously sample neighbor edges until the pattern can be formed. In mining triangles, each neighborhood sampler starts by sampling an edge uniformly at random and then sampling the second edge from the neighbor edges of the first edge. If there is a third edge in the remaining edges that can form a triangle with the existing two edges, this sampler successfully samples a triangle. Compared to the standard sampling approach that has a $1/m^3$ probability (sampling three edges uniformly at random), neighborhood sampling has a larger probability $1/m \cdot c$ to sample a triangle, where c is the number of neighbor edges of the first edge.

• **Limitations of existing sampling-based systems.** While sampling-based approximation is promising in reducing the computation in graph pattern mining compared to exact algorithms, two significant issues remain:

(1) Scalability remains an issue in mining complex patterns in (dense) large graphs for systems like ASAP [44]. In particular, the number of required samplers can be prohibitively large, leading to high computation and memory costs. Taking neighborhood sampling as an example, it requires $O(\frac{m^2}{f_p})$ to estimate 4-vertex patterns and $O(\frac{m^3\Delta}{f_p})$ for 5-vertex patterns, where Δ is the maximum degree in the graph and f_p is the occurrence of the pattern. From 4-vertex to 5-vertex, the computation complexity is increased by up to $O(m\Delta)$, where the number of edges m can be large (e.g., Twitter graph [50] has 1.2 billion edges). This complexity will be increased dramatically for more complex patterns, and this observation is confirmed by ASAP that they cannot scale to more than 5-vertex patterns in their large graphs evaluated in their 16-server testbed.

(2) Using ASAP, one needs to define how to sample a pattern using their neighborhood API [44]. While it is straightforward to sample simple patterns (e.g., triangle), sampling complex patterns is challenging (e.g., triangle-triangle). For instance, there is only one way to sample a triangle as described above, but there are multiple ways to sample a triangle-triangle (two triangles connected by an edge). It is challenging for developers to figure out all possible samplers when the pattern is even more complex. If samplers do not cover all possible ways to sample patterns, we can see severe underestimations in the final results.

2.2 Approximation with Graph Decomposition

The goal of our system, Arya, is to explore a scalable solution for mining complex patterns in large graphs. One potential way to improve the scalability is to continuously design and develop new sampling techniques that can sample patterns with higher probabilities. However, the fundamental limitation in this direction is that, we need to sample at least these many of edges and vertices to form a pattern, which implies an upper bound on how large the sampling probability can be on a specific pattern [35].

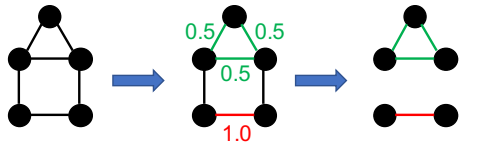


Figure 1: **An example of decomposing a pattern.**

Instead, Arya aims to take another road to improve the scalability of approximate pattern mining. The question we ask is that ***if we cannot improve the sampling technique further, can we instead reduce the complexity of the pattern to be sampled?*** We find that, graph decomposition theory [30], which is a powerful tool to reduce the complexity of graph matching and coloring problems, can also be applied jointly with existing sampling techniques to reduce the pattern mining complexity. Recent graph theory advances [18] made a contribution to proving that any subgraph can be decomposed as a collection of vertex-disjoint odd cycles and stars by solving an optimal fractional edge cover problem. This decomposition is a desired property for our purposes, as no matter how complex the patterns are, they can be decomposed into cycles and stars (tackling issue (2)), and these two subpatterns are "easier" cases to be sampled (tackling issue (1)).

Definition 1 ([18]). *Denote the fractional edge cover of a pattern P as $P(V_P, E_P)$, where V_P is the vertex set and E_P is the edge set. $P(V_P, E_P)$ is a mapping $\phi : E_P \rightarrow [0,1]$ such that for each v in V_P, $\sum_{e \in E_P, v \in e} \phi(e) \geq 1$. The fractional edge cover number is $\sum_{e \in E_P} \phi(e)$.*

The *optimal* fractional cover is to find a subset of edges in the pattern (covering all vertices) that minimum the fractional edge cover number $\rho(P)$ (i.e., min $\sum_{e \in E_P} \phi(e)$). Intuitively, the key insight (detailed proofs in [18]-A.2) is that for any pattern P, there always exists an optimal fractional cover that maps weights 0.5 to edges that can form odd cycles and maps weights 1.0 to edges that do not belong to any odd cycle (in the cover). This result is powerful because it ensures that we can decompose arbitrary patterns into a collection of odd cycles and stars. Moreover, the analysis further proves that this decomposition reaches optimal bounds for sampling arbitrary patterns, which strengthens our confidence in this decomposition. Thus, we need to calculate the following linear programming (LP), and construct odd cycles with edges of weights 0.5 and stars with edges of weights 1.0:

$$Minimize \sum_{e \in E_P} \phi(e)$$

$$s.t. \sum_{e \in E_P : v \in e} \phi(e) \geq 1, \forall v \in V_P$$

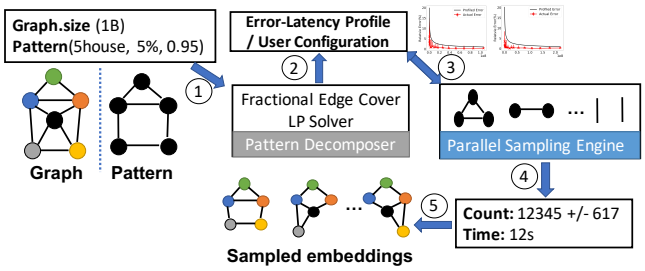


Figure 2: **Overall architecture of Arya.**

Figure 1 shows a 5-house example to find the optimal fraction cover as (b), and we then decompose 5-house into a three-cycle (if the weight ϕ of each edge is 0.5) and a 1-star (if the weight is 1.0).

Challenges. While the decomposition technique has provable theoretical guarantees, there are several challenges in building a general, distributed system for large-scale approximate graph mining. First, to be of practical use, once a pattern is decomposed into subpatterns, we need a distributed processing engine that optimizes the performance of running a large number of (subpattern) samplers, e.g., how to schedule the execution of the samplers and scale to distributed settings with optimized communication. Second, the graph theory we leverage assumes homogeneous edges and vertices while real-world graphs are often associated with properties. Therefore, the mining queries require *predicate matching* that envisions the technique to be property-aware. Finally, as an approximate processing system, we need to allow users to trade-off accuracy for running time. We need to understand the relationship between errors and actual running time in both single and distributed settings.

3 Arya Overview

We design Arya, an approximate graph pattern mining system leveraging decomposition-based graph sampling. Figure 2 demonstrates the overall architecture of Arya. Arya provides three operating modes to adjust to different compute scenarios: (1) *Single machine* mode that is optimized for local edge and vertex queries; (2) *distributed with replicated graphs* mode where the graph dataset is replicated entirely to multiple machines; (3) *distributed with partitioned graphs* mode that is developed with distributed KV-store (e.g., Memcached) to support arbitrarily partitioned graphs across machines. We provide an overview of different components in Arya and how users can leverage our system to perform approximate mining tasks for arbitrary patterns.

Arya workflow. Arya allows users to mine arbitrary patterns in a graph. As an approximate engine, a user can specify an input pattern and an error budget as follows:

- **Input pattern P:** The user defines an arbitrary subgraph P of the input graph as the pattern to mine in Arya. The user specifies P (in a text file) by adding a list of edges that form the pattern. This pattern will then be decomposed into a collection of odd cycles and stars via Arya decomposer. Unlike prior approximate mining systems, the user does not need to define the ways to sample the pattern as Arya will always sample stars and odd cycles for arbitrary patterns.
- **Error budget ε:** The user specifies an accuracy target by setting an error budget ε (e.g., 5%) with a confidence interval (e.g., 95%). Arya is expected to output an approximate result within ε error in time T.

After specifying the input pattern and the error budget, Arya first decomposes the pattern via the fractional edge cover LP solver as ①. For building an ELP, the ELP engine will return a required number of samplers (and time) for the error budget ε as ② using the parallel sampling engine ③; or the user directly specifies the number of samplers to run. Once the user approves the estimated time, the sampling engine will perform the approximation and return the estimated count with the actual run time as ④. The sampling engine also finds a set of sampled embeddings as ⑤.

Sampling vs. Enumeration. While Arya shows tremendous performance improvements on mining various complex patterns, we observe that Arya works best with the following two assumptions: (1) The graph is *dense* such that there are many pattern occurrences. (2) The decomposed subpatterns need only a few remaining edges to complete the pattern. For (1), it is a fundamental argument between deterministic enumeration-based approaches and sampling-based approaches. When the graph is sparse, it is challenging to find a pattern via random sampling (like "a needle in a haystack"), while it is a better case for enumeration. For (2), if the decomposition of a pattern breaks too many edges, each Arya's sampler spends extensive efforts on searching the remaining edges to complete the pattern, degrading the execution performance. If these two assumptions do not hold, Arya may experience many failed trials and thus requires more samplers. Therefore, while Arya supports arbitrary pattern mining, the actual runtime depends on the above two key conditions.

4 Basic Design

We now present how Arya enables ultra-fast graph pattern mining by combining pattern decomposition and edge-based sampling as a theoretical foundation. We focus on the single machine design in this section and extend it to distributed settings in the next section.

4.1 Pattern Sampling Algorithms

By leveraging graph decomposition theory (§2.2), a pattern will be represented as a set of vertex-disjoint odd cycles and stars. Our pattern sampler is to sample the relevant odd cycles and stars from the graph and check if there exist remaining edges that complete the pattern. Thus, we introduce two sampling algorithms based on [15, 18] to sample them separately, then use them to construct the pattern. In the algorithms, we denote an odd cycle with $2k+1$ edges as C_{2k+1} ($k \geq 1$) and a star with l petals as S_l.

Algorithm 1 Odd Cycle Sampler

1: **Input**:
2: Graph $G = (V, E)$ where $|E| = m$, an odd cycle C_{2k+1}
3: **Output**:
4: An instance of C_{2k+1} and sampling probability $Pr[C_{2k+1}]$ or 0
5: Sample an edge $e_1 = (u_1, v_1)$ from graph such that $d(u_1) \leq d(v_1)$. ▷ *sample first edge with an order*
6: Sample $k-1$ edges $e_2 = (v_2, u_2), \ldots, e_k = (v_k, u_k)$ with replacement from G. ▷ *sample rest edges as the cycle skeleton*
7: Sample a vertex n from the neighbors of u_1. ▷ *One more*
8: Check if there are remaining edges $(v_1, u_2), \ldots, (v_{k-1}, u_k), (v_k, n)$ in G. If succeeds, output C_{2k+1} and $Pr[C_{2k+1}]$; otherwise, 0.

Odd Cycle Sampler. At a high level, our goal here is to sample odd cycles and we can adopt *any* cycle sampling algorithms (e.g., neighborhood sampling [61] used in ASAP [44]). In Arya, we attempt to introduce the algorithm shown in [31] and [18]: first uniformly sample k edges (with the first edge having an order based on the degree), a neighbor edge of the first edge, and then test if there are remaining k edges in the graph to complete a cycle. Compared to ASAP, it is easier for Arya's sampler to sample an odd cycle using edge sampling since the probability of forming an odd cycle is higher. However, our empirical results show that the two algorithms are comparable for cycles (§7). We adopt this algorithm for two reasons: (1) *Easy to support longer odd cycles:* compared to neighborhood sampling, most of its sampling phase involves only random edge sampling, and thus no need to store nested neighborhood states (e.g. neighbors of neighbors); (2) *Easy to optimize performance* with the hash-based graph structures presented in §4.2 to accelerate queries.

Specially, we present the odd cycle sampler in Algorithm 1. First, we sample a special (directed) edge $e_1 = (u_1, v_1)$ whose first vertex does not have a larger degree than that of the second vertex (Line 5). Second, we sample another $k-1$ edges uniformly at random with replacement (Line 6). Third, we sample a neighbor edge (n, u_1) of u_1 as the last hoop of the cycle (Line 7). Finally, we need to test if there are remaining k edges in the graph to complete the odd cycle as $(u_1, v_1), (v_1, u_2), \ldots, (u_k, v_k), (v_k, n), (n, u_1)$. Since each sampling step is independent, the overall probability to sample this odd cycle is $Pr[C_{2k+1}] = Pr[e_1] \cdot Pr[e_2] \cdots Pr[e_k] \cdot Pr[n] = \frac{1}{m}(\frac{1}{2m})^{k-1}\frac{1}{d(u_1)}$.

Star Sampler. Intuitively, a star consists of a center vertex and a few petals, and sampling a star can be straightforward. There is a broad spectrum of theory work using star samplers as a main or subroutine in various applications (e.g., sparsification, clustering, and matching) [15, 18, 49]. Here, we adopt a common weighted star sampler as in Algorithm 2 (e.g., [15, 18]). We start by selecting a vertex v_1 with probability proportional to its degree $d(v_1)$ (i.e., $\frac{d(v_1)}{2m}$). This step is to sample centers that are more likely to form stars with multiple petals. In practice, we optimize the query performance by performing an edge sampling as the way to sample v_1. For instance, if a graph has 50 edges and a vertex v_1 has a degree of 10, randomly sampling an edge is equivalent to sampling a vertex that is v_1 with a probability of $1/10$ because there are 10 edges in the graph that are incident to v_1. This sampled vertex is used as the star center, and we will then sample l vertices from v_1's neighbors uniformly at random without replacement. We will either find such an l-star or return zero from this step. Overall, the probability to sample a star is $Pr[S_l] = Pr[v_1] \cdot Pr[petal\_vertices] = \frac{d_{v_1}}{2m}\binom{d_{v_1}}{l}$.

Algorithm 2 Star Sampler

1: **Input**:
2: Graph $G = (V, E)$ where $|E| = m$, a star S_l with l pedals
3: **Output**:
4: An instance of S_l and sampling probability $Pr[S_l]$ or 0
5: Sample a vertex $v_1 \in V$ with probability $d_{v_1}/2m$ for any $v_1 \in V$. ▷ *weighted center vertex sampling*
6: Sample l petal vertices uniformly at random from the neighbors of v_1 without replacement . ▷ *sample pedals to complete*
7: Output S_l and $Pr[S_l]$ if succeeds; otherwise, 0.

Approximation for the Original Pattern. We can sample an embedding of the original input pattern if and only if all associated odd cycle samplers and star samplers find their instances. If any sampler does not successfully form their cycle/star embedding, we will terminate this sampler for the original pattern and return zero. Once all the decomposed pattern samplers finish, we need to *reconstruct* the original pattern P by *merging* the cycles/stars and *testing* if the remaining edges between the cycles and the stars do exist in the graph to form the pattern. This is the last step of the whole pattern sampler. During the testing, we list all the possible remaining edges and check if they exist in the graph until there are enough checked edges to complete the pattern. If a complete pattern can be formed with o odd cycles and s stars, the probability of the sampler is $Pr[P] = Pr[C^1] \cdots Pr[C^o] Pr[S^1] \cdots Pr[S^s]$. A sampler outputs $R[P] = \frac{1}{Pr[P]}$ if a pattern instance is found; otherwise, it outputs $R[P] = 0$.

In summary, Arya runs a number of such pattern samplers in parallel based on ELP. In the final "reduce" phase, if there are n samplers and sampler i returns result $R_i[P]$, Arya returns $\frac{\sum_{i=1}^{n} R_i[P]}{n}$ as the final result. This is because Arya's pattern sampler finds any possible embedding P_i with $Pr[P_i]$. Suppose there are $\#P$ embeddings of the pattern, the expected output of a sampler is $E = \sum_{i=1}^{\#P} \frac{1}{Pr[P_i]} \cdot Pr[P_i] + 0 \cdot (1 - Pr[P_i]) = \#P$. With more samplers, the average of the sampler outputs will be closer to the expected value $\#P$. Therefore, Arya trades more samplers for better accuracy.

4.2 Sampler-Friendly Graph Structures

We observe that both odd cycle and star samplers involve a number of specific queries to the graph data, which are the major computation bottlenecks in the sampler runtime. To improve sampler performance, we summarize the most frequent runtime queries and provide simple, yet effective data structures to accelerate them.

- **Edge sampling**: sample an edge e uniformly at random from the graph.
- **Neighbor sampling** (v,i): perform a neighborhood sampling on v to obtain the i-th neighbor edge ($i \leq d(v)$) and check what vertex is associated with this edge.
- **Degree checking** v: obtain the degree of vertex v.
- **Edge checking** (u,v): check if vertices u and v form an edge in the graph.

Given the nature of the queries above in randomized algorithms, we should optimize data stores to accelerate processing. For instance, an edge sampling query can be implemented unoptimized as drawing a random number from the edge list $[1 \ldots m]$ and taking a linear traversal to find the exact edge. Instead, we use two auxiliary data stores for performance improvements: (1) An *edge array* that is grouped by vertex with the requirement that all neighbor edges of a vertex are stored consecutively. We observe that many public graph datasets are already stored in this order [6]. (2) A *hash table* that maps vertices to their metadata. Specifically, each vertex has an integer as its ID and its metadata containing the vertex's degree and the starting index of the vertex in the edge array.

4.3 Advanced Pattern Mining Features

Beyond approximating the occurrences of a pattern in a graph, Arya provides users with several advanced features.

Predicate matching. A common way of representing the graph data is in the form of *property graph*, where user-defined properties are attached to the vertices and the edges. Thus, the real-world queries to a property graph may require to match patterns satisfying certain predicates. For instance, a predicate matching query can ask for the count of all 5-House patterns in the graph where all edges are associated with an organization or all vertices meet a certain type.

Arya supports three types of predicates—*at-least-one*, *at-least-percentage*, and *all*. For "at-least-one" or "at-least-percentage" predicates, users are asked to specify a predicate that is applied to at least one (or a percentage or all) of the edges or vertices. Arya can support these predicates since a new property "subgraph" can be maintained and the same sampling techniques can be applied. To perform a predicate matching task, we introduce a *conservative sampling* stage. We first create an auxiliary graph that contains only the edges or the vertices that satisfy the predicate. The odd cycle and star samplers will sample the first (or a percentage of) edges or vertices from the auxiliary graph and then perform the rest of the sampling in the original graph. Different from the non-predicate-case, we need to refine the sampling rates based on the number of matched edges or vertices stored in the auxiliary data store. In a simple example, the probability of sampling the first edge uniformly at random is not $1/m$ but $1/m^*$, where m^* is the number of edges that satisfy the predicate. More details can be seen in Appendix A.

Intermediate state caching (e.g., Motifs). We consider two scenarios when some intermediate states can be cached and reused. (a) First, when running multiple mining tasks on the same graph, different patterns may share one or more decomposed subpatterns (i.e., odd cycles and stars). For instance, the decomposed 5-house and triangle patterns share a 3-cycle subpattern. Thus, Arya automatically caches the previous subgraph samplers (3 cycles) to reuse across patterns. (b) Second, some patterns share the same sampling steps in their samplers to form the patterns, e.g., one can sample any 4-motif patterns (except for 3-star) using two 1-star samplers with different remaining edges to complete the patterns. Thus, Arya does not need to sample each 4-motif pattern separately.

Algorithm 3 Error-Latency Profile (δ,ε)

Input: Original graph G with M edges, sampled subgraph g with m edges (with probability r), pattern P with p edges, error target ε, and confidence $1-\delta$.
Output: Number of estimators N_e for G

1: $avg_{last} \leftarrow \inf$, $range_{last} \leftarrow \inf$, $N_c \leftarrow 10{,}000$
2: **while** `True` **do**
3: Run Arya 3 times with N_c samplers on subgraph g
4: $avg_{cur} \leftarrow$ the average count of the 3 trials.
5: $range_{cur} \leftarrow$ the range (max - min) of the 3 trials.
6: $\tilde{\varepsilon} \leftarrow |avg_{last} - avg_{cur}|/avg_{cur}$
7: **if** $\frac{range_{last}}{avg_{last}} < 10\%$ and $\tilde{\varepsilon} < \varepsilon$ and $\frac{range_{cur}}{avg_{cur}} < 10\%$ **then**
8: $C \leftarrow \frac{N\tilde{\varepsilon}^2 avg_{cur}}{m^{\rho(P)}}$, $h \leftarrow avg_{cur}$
9: `Break`
10: $N_c \leftarrow N_c \times 2$, $avg_{last} \leftarrow avg_{cur}$, $range_{last} \leftarrow range_{cur}$
11: $N_e \leftarrow \frac{C \cdot M^{\rho(P)}}{\frac{h\varepsilon^2}{r^p}\delta}$ ▷ $\rho(P)$ is known given P

4.4 Error Latency Profile (ELP)

Arya allows users to tradeoff accuracy for result latency. As an approximation system, Arya needs to determine the number of samplers for expected errors and running time. Arya uses a heuristic to build the ELP. Our experiments in §7.3 demonstrate the accuracy of the ELP.

According to the mathematical analysis in [18, 30] using Chebyshev's inequality, decomposition-based sampling requires $O(\frac{m^{\rho(P)}}{\#P})$ estimators to provide a $(1\pm\varepsilon)$-approximation to the ground truth ($\#P$) for any $\varepsilon > 0$, where $\rho(P)$ is the minimum fractional edge cover number of pattern P. Furthermore, the actual number of required samplers is lower bounded by $\frac{Cm^{\rho(P)}}{\#P \cdot \varepsilon^2 \delta}$ with probability $1-\delta$ for some constant C. Thus, our goal is to estimate C for a particular graph and a pattern. We

achieve this by using a "sparsified" input graph: we uniformly sample a subgraph from the original graph with probability r (e.g., 30%), and determine an approximate number of needed samplers by running varying numbers of samplers and converging to a stable pattern count. The pseudocode is shown in Algorithm 3. Line 1 gives an initial number of samplers N_c to start with. For a given N_c, the algorithm runs Arya 3 times to obtain the average and range of 3 trails in lines 3 to 5. If the last and current range difference is small enough and when using the current average result as the ground truth, the last average estimated result is within the error target (line 7), we can exit, calculate an estimated C , and treat the current average result as the estimated ground truth h (line 8). Otherwise, ELP exponentially increases N_c (line 10) and proceeds again. Line 11 calculates N_e, the number of samplers for G, based on C and δ, ε, M and scaled $\#P$.

5 Scaling Arya to Distributed Settings

In this section, we introduce how Arya is scaled to multiple machines to support larger graphs and more complex patterns. Naturally, a graph store can be distributed in the following three ways: (1) distributed replicated graphs, (2) randomly partitioned graphs, and (3) arbitrarily partitioned graphs. Arya is designed to support all these configurations and arbitrarily partitioned graph is the most challenging one. We introduce several optimizations to improve Arya's scalability by up to 4.7× over the basic design.

5.1 Distributed Replicated Graphs

Replicated graphs are a common approach for serving input data in distributed graph mining systems, such as Fractal [33] and GraphPi [64]. In Arya, the compute can be distributed directly across the machines if the graphs are replicated. This is because each sampler is independent and each machine will be assigned a subset of required samplers to run on its multiple CPU cores/threads. Each thread takes one sampler at a time. Once the samplers are assigned, there is no communication between machines or samplers until the final aggregation of the results from all the samplers.

5.2 Distributed Partitioned Graphs

The second scenario is when graphs are partitioned to multiple machines. G-thinker [72] and Kudu [52] are example mining systems that assume graphs are partitioned among compute nodes. Similarly, graphs can also be separated from compute nodes in real-world scenarios. Meta, for example, has its own cluster of graph store RIPQ [68]. Arya assumes an API (e.g., *getedge(edgeID)*, *getAdjList(vertexID))* to access partitioned graphs and can work with either locally partitioned graphs (as in G-thinker) or remote graph stores.

In practice, many partitioning strategies are possible. ASAP [44] requires to partition the graph edges uniformly at random. Graph partition strategies often depend on the workloads (e.g., PageRank [59]). In addition, the graphs may need to be partitioned based on strategies to be compliant with security and privacy requirements, such as GDPR [3] and GDPR-Neo4j [5]). Unlike ASAP, Arya makes no assumptions about partitioning strategies.

Arya extends its design from replicated graphs to partitioned graphs, with one major challenge to overcome. In contrast to the replicated graph scenario, a graph is partitioned into slices to compute nodes; each node's samplers will have a potentially large number of random accesses to the graph data stored in other nodes. This poses significant scaling challenges for Arya on partitioned graphs: Arya will be constrained by network communication overheads. A single triangle sampler, for example, entails six graph queries (1 edge sampling, 3 degree checkings, 1 neighbor sampling and 1 edge checking). Each triangle sampler in the Friendster graph [74] generates around 6KB network traffic. To count triangles in Friendster with a 5% error, we will need at least 4 million samplers, which translates to 20 million graph queries and 23GB of network traffic. The comptation-communication ratio is approximately $c\frac{p}{(p-1)}$, where c is a constant depending on the pattern and graph, and p is the number of partitions. The detailed analysis is defered to Appendix C. While the intermediate state caching technique can help reduce communication costs, Arya introduces two more techniques for communication reduction: (1) probability-aware sampler scheduling and (2) batched sampling and communication.

Technique 1: Probability-aware sampler scheduling. A key observation we have is that different decomposed subpattern samplers (e.g., triangle vs. 2-star) have different probabilities to fail (not finding one). According to our Mico graph profiling, a 2-star sampler has a 0.5% failure probability while a triangle sampler has a 92% probability to fail. This is because simpler structures are more likely to be sampled than complex structures in a graph. Based on this observation, we can save communication overheads if we sample these likely-to-fail subpatterns earlier. A lot of such samplers will fail early and we can prune them without running other subpattern samplers.We note that after decomposition, each subpattern sampling occurs independently, and thus the order of subpattern sampling has no effect on the original pattern sampling's success/failure probability and overall accuracy. Taking the triangle-2star pattern as an example: for each pattern sampler, if sampling the triangle first, it is more likely to fail (92%) and there is no need to sample 2-star in 92% of the cases. Hence, we schedule subpattern samplers in the order of their sampling failure probabilities to achieve better performances.

To do so, we must address an important question: how do we know which subpattern samplers are likely to fail? The answer depends on the pattern and graph. Given the static graph, Arya first offline profiles failure probabilities of popular subpatterns (such as 2-star, triangle) in a small number of trials. Then each pattern counting task can query the failure probability profile for any subpattern samplers. When the failure probability of a subpattern is not in the

profile, we perform an online profile by letting the first set (e.g., 10%) of the samplers collect the failure probabilities information without early pruning. These probabilities will be used to schedule the remaining samplers. This technique is applied to all Arya versions.

Now, we analyze the overheads of Arya's offline and online profiles. The cost of offline failure profiling is minimal because the overheads are amortized by all queries to the graph, and profiling is limited to simple common subpatterns. For example, profiling simple 2- to 5-stars and triangle subpatterns for the Friendster graph takes only 220ms even for less than 5% error results, whereas a single 5-node pattern query to Friendster can take tens of seconds as we will show in Figure 5(b).

Arya's online failure probability profiling trades off early pruning opportunities in the query's first 10% samplers for better subpattern sampling order in its remaining 90% samplers. This approach produces runtime comparable to Arya with perfect subpattern sampling order pre-knowledge and significant improvements over Arya with the worst subpattern sampling order. We use 10% as we found it to be adequate for accurate simple subpattern failure probability profiling.

If the profiled failure probability is inaccurate (which is uncommon as subpatterns are simple patterns that are easy to estimate), Arya may use suboptimal subpattern sampling orders. In this worst case, probability-aware sampling performs similarly to the case of no early pruning.

Technique 2: Batched sampling/communication. Arya reduces overheads from the network stack by using batched sampling and communication. One Arya thread advances a batch of samplers at the same time (vs. progress one sampler until it finishes). When a graph query is required in a sampler, the thread buffers the query and pauses the sampler before moving on to the next sampler in the batch. When all of the samplers in the batch are waiting for graph queries, the thread will begin its batch communication with the graph store (i.e., send out the queries we buffered, for example, with Memcached multi-get).

6 Implementation

We build Arya for both single-machine and distributed graph computing scenarios. The pattern decomposition logic is implemented with Python, and the core components of Arya are written in C++ with 11K LOC. We open-source Arya at [2].

Pattern Decomposition. Arya takes an arbitrary pattern as input and outputs a set of stars and odd-cycles via a pattern decomposition logic. As discussed in §2.2, Arya will find the optimal fractional cover of the input pattern. We use `scipy` linear programming package to find the optimal cover: it takes only 900ms on a single server to decompose complex 20-vertex patterns and less than 400ms for less complex patterns that have fewer than 10 vertices. This running time is negligible compared to the total mining time.

Graph Sampler. There are three versions of sampling logic written in C++: single-machine, distributed replicated graph, and distributed partitioned graph. The former two versions access in-process graph stores, while the distributed partitioned graph version accesses remote graph stores (e.g., Memcached) via TCP. To parallelize the samplers, all Arya versions employ multi-threading. Single-machine version and distributed replicated graph version use the work-stealing algorithm dynamically scheduling computations. A communication thread distributes tasks when the total number of samplers is smaller than required and uses asynchronous communication primitives for work stealing. Worker threads return the results from a batch of samplers to the communication thread when they finish a task. Worker threads then execute the next batch of samplers. We can configure the granularity of a sampling task. For distributed partitioned graph implementation, the master process will initiate samplers on each machine and collect results from each machine when the sampling phase is completed. For evaluating ASAP in a fair setting, we implement ASAP graph samplers using Arya's system API (which is faster than Spark used in ASAP), including accessing the graph structures and performing edge- and node-related queries.

7 Evaluation

We evaluate Arya on a variety of open-source and synthesized graphs and compare it to the state-of-the-art approximate mining system (ASAP [44]) and exact mining systems (*Single-machine:* Peregrine [45], DwarvesGraph [26], AutoMine [54]. *Distributed:* Fractal [33], GraphPi [64], G-Thinker [72], Kudu [52]). Our experiments demonstrate:

- Compared to single-machine exact mining systems, Arya is up to $105,365\times$ faster than Peregrine within a 5% loss of accuracy when counting complex patterns. To the best of our knowledge, Arya is the first system capable of mining complex patterns (>6 vertices) on giant graphs.
- Compared to distributed mining systems, Arya outperforms Fractal by $62\times$ to $56,842\times$ and GraphPi by up to $988\times$.
- Compared to ASAP, Arya can mine arbitrary patterns in both single-machine and distributed settings. Arya is up to $145\times$ times faster on a single machine and $55\times$ faster in distributed settings, with a 5% error target.
- Arya's probability-aware scheduling and batched sampling techniques are effective for speeding up Arya by up to $4.7\times$.

Datasets and baselines. We compare Arya to state-of-the-art systems using a set of representative graphs as in Table 1. We obtain the ground truth via running deterministic mining systems such as GraphPi [64] and Peregrine [45][2]. For datasets used in distributed partitioned graph experiments, the graph

[2]We are unable to get the mining results of P3 and P4 patterns [64] in the Twitter graph because all tested deterministic miners including GraphPi experienced system crashes or their running time exceeded 24 hours.

Size	Graph	Nodes	Edges	Degrees
Medium	Mico [37]	100,000	1,080,298	22
	Youtube [51]	1,134,890	2,987,624	8
Large	Twitter [50]	41.7 million	1.2 billion	36
	Friendster [75]	65.5 million	1.8 billion	28
Giant	RMAT-5B	500 million	5 billion	
	RMAT-10B	1 billion	10 billion	

Table 1: **Graph datasets used in the evaluation.** *We use the RMAT model [48] to generate small and giant synthetic graphs. In the RMAT model, we used default parameters (a,b,c,d) as (0.44,0.22,0.22,0.22).*

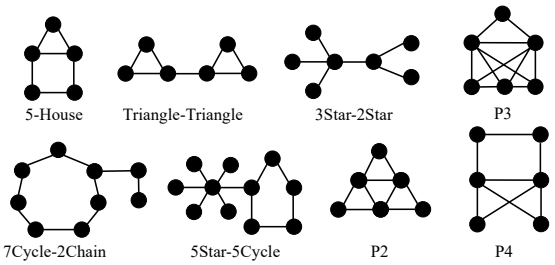


Figure 3: **Evaluated complex patterns.**

is partitioned based on node index hash into relatively similar sizes, and edges belonging to the same node are put into one partition. Since we do not have access to (single-machine) Automine, DwarvesGraph, and (distributed) Kudu, we refer to their performance numbers in a similar setup. Since ASAP is built on Spark, we reimplement its sampling approach using our API for a fair comparison.

Hardware testbed. Our experiments are carried out in three different hardware configurations: (1) Single-machine DRAM, which has 20 CPU cores and 188 GB of DRAM. (2) Single-machine PMEM (persistent memory) with additional 4 × 128GB Intel Optane DCPM. (3) Distributed settings with 4 to 32 machines in a cluster, each with the same configuration as the single-machine-DRAM. The testbed CPUs are Intel Xeon Silver 4114 clocked at 2.2Ghz per core.

Evaluated patterns. We evaluate both simple patterns (Triangle and 4-Motif) and complex patterns. Most prior systems did not evaluate patterns larger than 5-vertex while Arya can mine arbitrary patterns in large graphs. We describe the complex patterns we evaluate in Figure 3.

7.1 Single-Machine Performance

Overall Performance. We first compare Arya to Peregrine, Automine, Dwarvesgraph, and GraphPi. As shown in Table 2, we evaluate both simple patterns (triangle, 3-Motif) and complex patterns of up to 11 vertices on medium (Mico) and large graphs (Friendster). We also mine the extremely complex patterns such as 3Star-2Star. The results show that Arya significantly outperforms existing systems, particularly in complex patterns. On Mico, complex patterns (3Star-2Star, 7Cycle-2Star, 5Star-5Cycle) always take longer than 24 hours or crash. The long running time of Peregrine illustrates that

Pattern	Graph	System	Runtime	Error/Speedup
Triangle	Mico	**Arya**	**22ms**	0.74%
		Peregrine	46ms	2×
		GraphPi	3.5s	159×
	Friendster	**Arya**	**15ms**	1.24%
		Peregrine	11.3s	782×
		GraphPi	770.5s	51367×
3-Motif	Mico	**Arya**	**36ms**	0.09%
		Peregrine	67ms	1.8×
		DwarvesGraph	48ms	1.3×
		AutoMine	161ms	4.4×
		GraphPi	6.86s	190×
	Friendster	**Arya**	**59ms**	0.71%
		Peregrine	20.6s	349×
		GraphPi	804.4s	13634×
4-Motif	Mico	**Arya**	**1.0s**	0.42%
		Peregrine	5.2s	5.2×
		DwarvesGraph	1.3s	1.3×
		AutoMine	22s	22×
		GraphPi	21s	21×
	Friendster	**Arya**	**13248s**	0.76%
		Peregrine	2158s	**1/6**×
		DwarvesGraph	4369s	**1/3**×
		GraphPi	4399s	**1/3**×
3Star-2Star (7 vertices)	Mico	**Arya**	**0.8s**	N/A
		Peregrine	>24h	105365×
		GraphPi	2.33s	2.91×
	Friendster	**Arya**	**287s**	N/A
		Peregrine	Crashed	N/A
		GraphPi	924s	3.22×
7Cycle-2Chain (9 vertices)	Mico	**Arya**	**4s**	N/A
		Peregrine	Crashed	N/A
		GraphPi	Stuck	N/A
5Star-5Cycle (11 vertices)	Mico	**Arya**	**211s**	N/A
		Peregrine	>24h	409×
		GraphPi	Stuck	N/A
P3 [64]	Mico	**Arya**	**11s**	2.5%
		GraphPi	8.7s	**1/1.2**×
P4 [64]	Mico	**Arya**	**6.7s**	1.6%
		GraphPi	13.3s	2×

Table 2: **Single-machine DRAM: Arya vs. Peregrine, DwarvesGraph, Automine.** *This table summarizes runtime of Arya and other graph engines on various patterns (first column) and graphs (second column). Arya has a 5% error target.*

existing exact mining systems are fundamentally incapable of mining complex patterns. In contrast, Arya counts 3Star-2Star in Mico in 0.8s, outperforming Peregrine by 105,365×. We observe that while GraphPi completes mining star-related patterns, their results were incorrect, which prevents us to evaluate Arya's errors in some cases.

In this setting, we also explore an undesirable scenario for Arya (and any sampling-based approaches). In the Friendster graph, the occurrence of 4-Motif is relatively "sparse", making sampling-based approaches fundamentally more challenging to sample patterns. This is due to the "searching a needle in a haystack" effect and it is an ideal case for traversal-based solutions. Thus, in this scenario, Arya is running 3 to 6 times slower than exact mining solutions.

Table 3 shows results for Arya's **intermediate state caching** technique under the scenario when running three

Mico	Triangle-Triangle	5-House	Triangle
No Cache	13.3s	4.8s	0.079s
Cache	14.6s	3.0s	0.037s
Speed Up/Down	0.91×	1.6×	21.2×
Youtube	**Triangle-Triangle**	**5-House**	**Triangle**
No Cache	188.7s	297.9s	0.32s
Cache	198.7s	127.6s	0.011s
Speed Up/Down	0.95×	2.3×	27.9×

Table 3: **Arya's intermediate state caching technique.** *This table summarizes runtime and speedup of applying intermediate state caching technique when mining three patterns consecutively. Since these patterns share a common subpattern triangle, Arya caches the triangle samples in mining Triangle-Triangle and reuse them in the 5-House mining. Similarly, Arya also caches additional triangle samples when mining 5-House. These cached triangle samples accelerate the Triangle mining task significantly.*

Pattern	Graph	System	Runtime
Triangle	RMAT-5B	Arya (10%)	89s
	RMAT-5B	Arya (5%)	337s
	RMAT-5B	Peregrine	Crashed
3Star-2Star	RMAT-5B	Arya (10%)	395s
	RMAT-5B	Arya (5%)	1583s
	RMAT-5B	Peregrine	Crashed

Table 4: **Scaling single-machine Arya to giant graphs with PMEM.** *This table summarizes runtime of Arya (10% and 5% error rates) and Peregrine when mining on RMAT-5B.*

mining tasks (Triangle-Triangle, 5-House, and Triangle) on the same graph. Arya can mine multiple patterns one by one. Except for the last pattern Triangle, the sampled subpatterns and their actual sampling probabilities are cached; starting from the second pattern, we can reuse the cached subpatterns instead of sampling new ones and thus the running time is reduced. This experiment shows that when mining multiple patterns with shared subpatterns, Arya can achieve significant speedups (e.g., up to 27.9× for the last task) as the performance bottleneck is sampler computation and performing caching has negligible performance overheads.

We add **persistent memory** into the single machine to mimic large memory machines. On a giant 5-billion-edge graph (RMAT-5B), Arya counts triangles in 337 sec and mines a complex pattern of 7 vertices (3Star-2Star) in less than 30min while Peregrine fails to complete (Table 4).

Arya vs. ASAP. Figure 4 compares the running time of Arya and ASAP for different error rates. Both approximate systems, as expected, require more samplers to achieve lower error rates. However, the performance differences of the two approaches lie in two key factors: (1) the number of samplers needed and (2) the running time of each sampler. Compared to ASAP with the same error rate, Arya usually achieves better runtime because it requires fewer samplers (due to decomposition) and/or individual samplers run faster (due to Arya uses edge sampling while ASAP uses neighborhood sampling and Arya's system optimizations). For instance, when the graph is large (e.g., YouTube), the pattern is complex (e.g., 5-House, Triangle-Triangle), Arya requires fewer samplers, each of which is also faster than that of ASAP. Thus, for example, Youtube graph and 5-House pattern (Figure 4 (c)), Arya achieves less than a 5% error rate in 1.2s, whereas ASAP takes 3 min (145× slower). For small dense graphs (e.g., Mico, Figure 4 (b)), Arya and ASAP have comparable performances because they require comparable numbers of samplers, and their samplers have similar running times.

7.2 Scaling Arya on Distributed Settings

Arya can mine graphs that are (a) replicated across servers and (b) partitioned across servers (e.g., simulating geo-distributed graphs). We use a cluster with 4 to 32 servers.

7.2.1 Distributed Replicated Graphs

When graphs are replicated across nodes, Arya mines on each node in parallel and aggregates sampled results in a "reduce" phase. Many existing systems (e.g., Fractal and GraphPi) scale to multiple nodes using replicated graphs.

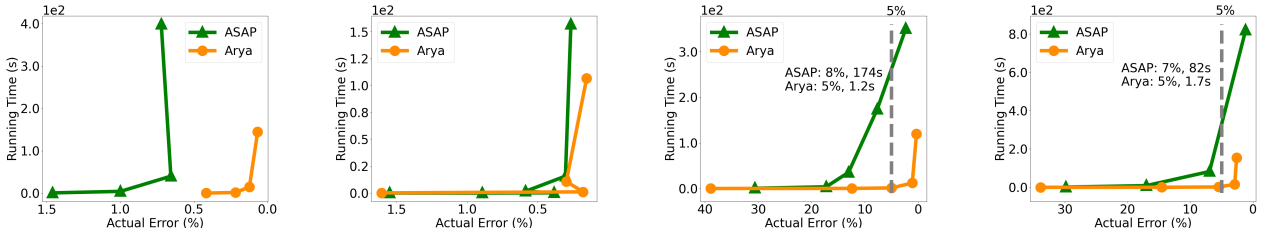
Overall performance. As depicted in Table 5, we compare Arya to Fractal and GraphPi on a 4-node cluster. In summary, Arya outperforms both Fractal and GraphPi, especially when graph is large and pattern is complex. For example, when mining triangles on the Twitter Graph, Arya achieves a 988× speedup over GraphPi. When the pattern is changed to Triangle-Triangle (a 6-vertex complex pattern), neither Fractal nor GraphPi can complete execution within a day, whereas Arya takes only 393 seconds. Overall, our results show that Arya has a significant advantage when mining complex patterns on large graphs.

Table 6 compares the performance of Arya, GraphPi, and ASAP (Spark version) on larger clusters. Arya outperforms both ASAP (by up to 55×) and GraphPi (by up to 1084×) on simple (e.g., 3-Motif) and complex patterns (e.g. 5-House). Referring to one of GraphPi's pattern (P2) mining results on a world-class supercomputer (up to 1024 nodes), we can see that approximate graph mining system Arya can achieve even better performance with only 16 nodes, demonstrating significantly better scalability.

Scalability in a Cluster. Figure 6 illustrates how Arya scales as more nodes (or cores) are added to the cluster. The runtime of Arya decreases as more machines are assigned to it. Arya can scale for both small and large graphs, whereas GraphPi cannot scale for larger Twitter graph with more than eight nodes. We find that the scaling of Arya is slightly worse than linear scaling. This is due to increased synchronization overheads of the final results when there are more nodes.

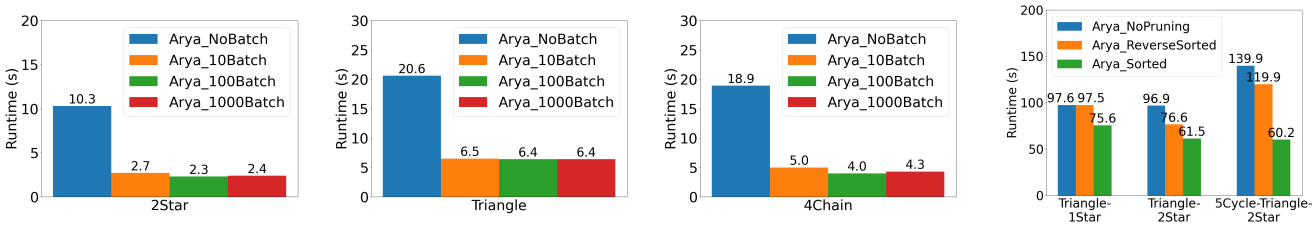
7.2.2 Distributed Partitioned Graphs

Unlike in a replicated graph setup, the graphs are partitioned across machines, and Arya samplers now require communi-



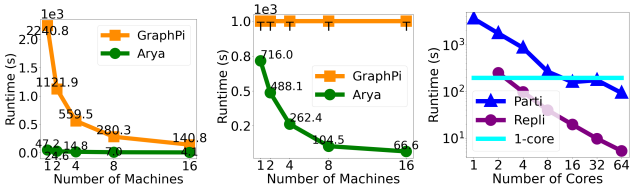
(a) Youtube, 4-Chain (b) Mico, 5-House (c) Youtube, 5-House (d) Youtube, Triangle-Triangle

Figure 4: **Comparing ASAP and Arya on running time vs. actual errors.** *This figure compares ASAP (our reimplementation for fairness) and Arya's running time (y axis) v.s. estimation errors (x axis, descending order). We report median absolute error rate \|%\| from 10 runs of each experiment. As shown, Arya requires fewer samplings and less runtime for the same error rate as ASAP, especially for large graphs and complex patterns.*



(a) Batching - Friendster, 8 Nodes (b) Probability-aware sampler scheduling

Figure 5: **Effectiveness of Arya batching and probability-aware sampler scheduling.** *This figure compares the performance of Arya with and without 1) batching and 2) probability-aware sampler scheduling techniques. In figure (a), Arya_NoBatch represents Arya without batching. Arya_KBatch represents Arya with K batched sampling and batched communication. We vary K between 10 and 1000. In figure (b), Arya_NoPruning represents the basic version of Arya, which samples all sampling blocks and then judges them all together. Arya_Sorted represents Arya when sampling according to the fail probability of each sampling block and terminating the estimator after any block fails. Arya_ReverseSorted is Arya that does sampling based on fail probability in a descending order.*



(a) Mico, Δ_Δ (b) Twitter, 5-House (c) Friendster, 4-Chain

Figure 6: **Scalability of Arya.** *This figure shows the performance of Arya when the number of nodes (cores) with replicated and partitioned graphs varies. We examine both small (Mico) and large (Twitter and Friendster) graphs, as well as various patterns. The letter 'T' indicates that the execution time exceeds 24 hours. Δ_Δ denotes Triangle-Triangle pattern. (a) and (b) shows the scalability of replicated-graph version Arya compared with GraphPi. (c) shows the replicated-graph and partitioned-graph versions of Arya compared with single-thread Arya showing the COST metric on Friendster graph and 4-Chain pattern, where "Parti" represents partitioned-graph version and "Repli" represents replicated-graph version.*

cations with remote nodes to obtain the necessary sampled edges or neighbors of vertices for testing. In this experiment, graphs are partitioned into machines based on their vertices and associated edges, and is stored in a Memcached instance on each machine. For simplicity, we evenly partition the graph to Memcached nodes.

Effectiveness of Arya Scaling Techniques. We begin by demonstrating the efficacy of batching and probability-aware scheduling techniques in improving Arya performance on partitioned graph setups. As shown in Figure 5 (a), batching can significantly improve Arya performance on partitioned graph setups. Batching improves Arya 2-Star, Triangle, and 4-Chain mining by 4.5×, 3.2×, and 4.7×, respectively, on eight nodes with Friendster graph. Because we've found that batching more than 100 samplers together yields minimal benefits, we set the default batching size to 100.

Figure 5 (b) shows how probability-aware sampler scheduling can help with Arya mining complex patterns. In this experiment, we use two nodes and the Mico graph. As an example, consider the following subpatterns: 2-Stars, triangle, and 5-Cycle. These subpatterns have very different sampling success probabilities: 2-Stars: 99.5%, Triangles: 8%, and 5-Cycles: 0.09%. When mining complex patterns containing these subpatterns, we can see that Arya samples with sorted likely-to-fail subpattern samplers and the early pruning achieves up to 2.3× (for 5Cycle-Triangle-2Star) better performance than no pruning. Arya's samplers with other orderings of subpatterns (e.g., *ReverseSorted*) cannot achieve comparable performance to fail probability sorted sampling.

McSherry's COST metric [55]. Figure 6 (c) shows the scalability of Arya's distributed replicated and partitioned versions compared with the runtime of a single thread Arya. The replicated version's COST is around 2.7 cores because the MPI implementation uses a master thread to poll results from worker threads, using at least 1 core. The partitioned version's

Pattern	Graph	System	Runtime	Error/Speedup
Triangle	Mico	**Arya**	**0.5s**	0.74%
		Fractal	145s	278×
		GraphPi	5.4s	10×
	Youtube	**Arya**	**0.55s**	0.78%
		Fractal	317s	576×
		GraphPi	38s	69×
	Twitter	**Arya**	**3.8s**	0.96%
		Fractal	>24h	22,736×
		GraphPi	3755s	988×
4-Motif	Mico	**Arya**	**3.3s**	0.42%
		Fractal	205s	62×
		GraphPi	33s	10×
	Youtube	**Arya**	**123s**	0.42%
		Fractal	29966s	243×
		GraphPi	219s	1.8×
	Twitter	**Arya**	**360s**	0.23%
		Fractal	failed	N/A
		GraphPi	>24h	240×
5-House	Mico	**Arya**	**0.8s**	0.63%
		Fractal	1822s	2366×
		GraphPi	6.3s	8×
	Youtube	**Arya**	**18s**	0.65%
		Fractal	2479s	142×
		GraphPi	36s	2×
	Twitter	**Arya**	**265s**	4.06%
		Fractal	failed	N/A
		GraphPi	>24h	326×
Δ_Δ	Mico	**Arya**	**1.5s**	0.71%
		Fractal	>24h	56,842×
		GraphPi	560s	368×
	Youtube	**Arya**	**15s**	1.13%
		Fractal	>24h	5760×
		GraphPi	11696s	779×
	Twitter	**Arya**	**393s**	N/A
		Fractal	failed	N/A
		GraphPi	>24h	220×

Table 5: **Distributed replicated graphs (4-nodes).**

COST is around 13 cores due to large communication costs in Memcached. In this version, scaling 1-core to 16-core experiments run on a single machine, and 32-core and 64-core experiments run on 2 and 4 machines of 16 cores.

Overall performance. Table 7 summarizes Arya's overall performance in comparison to G-thinker and Kudu. Kudu is a system that converts single-machine or distributed replicated graph mining systems (such as GraphPi and Automine) to partitioned graph setups. As shown in the table, Arya outperforms all existing exact graph mining systems on small (e.g., Mico) and large (Friendster) graphs, mining simple (e.g., 2-Star) and complex patterns (e.g., Triangle-2Star). The improvement is most noticeable on complex patterns. G-thinker, for example, fails to execute both Triangle-1Star and Triangle-2Star on a small Mico graph within a day; however, Arya can finish in seconds, yielding a speedup of more than 44000×.

Mining 10-billion edges graph on a large cluster. Table 8 shows the Arya runtime with 32 nodes mining patterns on a 10 billion edges graph (RMAT-10B). As shown in the table, Arya can mine huge graphs quickly. It completes triangle counting in 22 minutes for a 5% error rate and 358s for a 10% error rate. When mining 4-Chain, we see similar levels of speed. Arya can mine complex patterns even though it requires more time of sampling on a huge graph; for example, for a pattern with 7 vertices like 3Star-2Star, Arya finishes in 4.2h with a 10% error rate.

Pattern	Graph	System	Runtime	Error/Speedup
3-Motif	Twitter	**Arya, 16 × 8**	2.8s	0.34%
		ASAP, 16 × 8	150s	55×
		GraphPi, 16 × 8	2971s	1084×
5-House	Twitter	**Arya, 16 × 16**	60s	4.06%
		ASAP, 16 × 16	738s	12×
		GraphPi, 16 × 16	> 24h	1440×
Δ_Δ	Twitter	**Arya, 16 × 20**	100s	N/A
		GraphPi, 16 × 20	> 24h	864×
P2 [64]	Twitter	**Arya, 16 × 20**	856s	N/A
		GraphPi, 16 × 20	23.2h	98×
		GraphPi, 128 × 24	10000s	12×
		GraphPi, 1024 × 24	3000s	3.5×
P4 [64]	Twitter	**Arya, 16 × 20**	1600s	N/A
		GraphPi, 16 × 20	> 24h	54×

Table 6: **Comparing Arya, GraphPi, and ASAP on larger clusters.** *This table presents Arya/ASAP/GraphPi runtime on different clusters. The "system" column indicates system, the number of machines × the number of cores per machine. Arya is set to a 5% error target. GraphPi, 128 × 24 and 1024 × 24 results are picked from GraphPi paper [64].*

Pattern	Graph	System	Runtime	Error/Speedup
2-Star	Friendster	**Arya** 4 Nodes	0.58s	0.70%
		G-thinker 4 Nodes	52.4s	90×
		Arya 8 Nodes	0.64s	0.70%
		G-thinker 8 Nodes	30.8s	48×
Triangle	Friendster	**Arya** 4 Nodes	0.94s	1.24%
		G-thinker 4 Nodes	99s	105×
		Arya 8 Nodes	0.76s	1.24%
		G-thinker 8 Nodes	58s	76×
		Kudu-GraphPi 8 Nodes	79s	104×
		Kudu-Automine 8 Nodes	84s	110×
Triangle-1Star (5 vertices)	Mico	**Arya** 2 Nodes	1.93s	0.95%
		G-thinker 2 Nodes	>24h	44766×
Triangle-2Star (6 vertices)	Mico	**Arya** 2 Nodes	1.73s	0.40%
		G-thinker 2 Nodes	Crashed	N/A

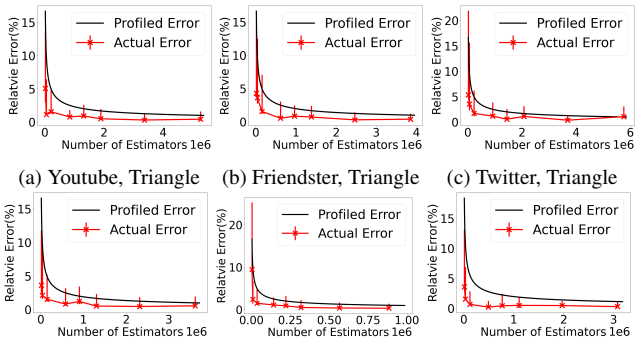
Table 7: **Distributed Partitioned Graphs: Arya vs. G-thinker vs. Kudu-GraphPi, Kudu-Automine.** *This table compares Arya and G-thinker and Kudu performance on partitioned graphs. The system column indicates both system name and how many nodes the graph is partitioned to.*

7.3 Effectiveness of Arya ELP

Finally, we evaluate the effectiveness of Arya Error-Latency Profiling. In this experiment, we compare the actual error vs. the predicted error from Arya ELP given a variety of amount of samplers. The Arya runtime is proportional to the number of samplers. As depicted in Figure 7, we investigate various patterns (Triangle and 3-Star) on various graphs (Youtube, Friendster, Twitter). We build the error profile by running

Pattern	Graph	System	Runtime
Triangle	RMAT-10B	Arya (10%)	358s
	RMAT-10B	Arya (5%)	1275s
4-Chain	RMAT-10B	Arya (10%)	171s
	RMAT-10B	Arya (5%)	688s
3Star-2Star	RMAT-10B	Arya (10%)	4.2h
	RMAT-10B	Arya (5%)	16.5h

Table 8: **Arya mining 10-billion edges huge graph.** *This table summarizes runtime of Arya (10% and 5% error rate) when mining on RMAT-10B on a 32-node cluster.*



(a) Youtube, Triangle (b) Friendster, Triangle (c) Twitter, Triangle

(d) Youtube, 3_Star (e) Friendster, 3_Star (f) Twitter, 3_Star

Figure 7: **Relative errors vs. number of estimators for YouTube, Friendster, and Twitter graphs.** *Actual error is obtained by compare Arya results with ground truth, and profiled error is the error estimated by ELP given a number of samplers.*

different numbers of samplers. We run ten trials for each x-axis value and report the *median* and *variance* error bars. As we can see, Arya ELP yields good upper bounds for the required error targets.

8 Related Work

Single-machine mining systems. A number of single machine exact mining systems have been proposed [26, 28, 43, 45, 54, 70]. These systems leverage a wide spectrum of system optimizations to prune the intermediate state and accelerate the subgraph exploration process. For instance, Peregrine [45] focuses on pattern-aware techniques to reduce the exploration of unnecessary subgraphs. Essentially, it prunes the incomplete subgraphs early if they cannot later form the pattern. Automine [54] and RStream [70] use guided exploration strategies but reduce memory usage. The fundamental performance bottleneck of these exact systems is that *regardless of how optimized the exploration techniques are, one must still explore all the patterns in the graph*. When the pattern occurrences are dense in the graph, this bottleneck will be significant. Arya leverages decomposition-based sampling, which significantly reduces exploration complexity. Another related work is DwarvesGraph [26], which also uses a type of pattern composition to count the decomposed subpatterns separately and thus reduces the overall computation. However, DwarvesGraph's decomposition technique cannot be applied with sampling techniques to further reduce search complexity. Some related architecture works leverage different architectures and hardware to accelerate enumerating graph patterns [27, 62, 65–67], while Arya and compared baselines run on general-purpose CPUs.

Distributed mining systems. To scale graph mining tasks on larger graphs, a wide range of distributed mining systems are proposed [10, 17, 25, 33, 53, 64, 69, 72]. Recent systems such as Fractal [33] and GraphPi [64] focused on supporting general-purpose mining tasks. Fractal extends the "embedding-aware" processing model by introducing the concept of *fractoids* and reduces the complexity of its depth-first search exploration. GraphPi is a high-performance graph miner that optimizes computation and communication overheads by introducing a 2-cycle-based automorphism elimination algorithm. GraphPi scales to supercomputers (up to 1024 compute nodes) to support complex pattern mining in large graphs. Arya's decomposition-based sampling technique further improves the scalability by several orders of magnitude.

Graph approximation theory. There have been rich efforts from theory community to analyze and propose approximate graph algorithms for various graph analytical tasks [12, 13, 18–20, 32, 42, 60, 71]. Among these efforts, only a small subset of them are used in graph systems. None of them are aimed at distributed scenarios, nor do they introduce methods to understand the real-world performance of the algorithms. To bring theory into practice, we entail careful understanding of the algorithmic tradeoffs and the actual computation scenarios. We leverage this rich theoretical foundation to further improve the sampling-based approximate systems and propose a series of sampling-friendly optimizations.

9 Conclusions

We observe that existing graph pattern mining systems cannot scale to complex pattern mining over large graphs as they fail to cope with the explosively growing mining complexity. We propose Arya as an approximate graph miner that combines graph decomposition theory with sampling techniques to achieve optimized mining complexity over arbitrary patterns. Arya can deal with large billion-level graphs even in a single machine and can scale to larger graphs in distributed settings. Our evaluation demonstrates that Arya outperforms state-of-the-art mining systems by up to five orders of magnitude. We posit that Arya can potentially be applied to extreme mining scenarios (e.g., trillion edges) on a small computing base, and we plan to explore this for future work.

Acknowledgments. We would like to thank the anonymous reviewers and our shepherd Peter Pietzuch for their helpful comments. This work was supported in part by NSF grants CNS-2107086, CNS-2106946, SaTC-2132643, and Red Hat Collaboratory at Boston University.

References

[1] Ant financial's innovations and practices in online graph computing. https://www.alibabacloud.com/blog/ant-financials-innovations-and-practices-in-online-graph-computing_595846.

[2] Arya graph pattern mining codebase. https://github.com/Froot-NetSys/Arya.

[3] General Data Protection Regulation. https://gdpr-info.eu/.

[4] Graph databases: Updates on their growing popularity. https://www.dataversity.net/graph-databases-updates-on-their-growing-popularity/.

[5] Neo4j Privacy Shield: The Graph Solution for GDPR. https://neo4j.com/use-cases/gdpr-compliance/.

[6] Stanford Large Network Dataset Collection. https://snap.stanford.edu/.

[7] Weg graph. http://webdatacommons.org/, 2014.

[8] A comparison of state-of-the-art graph processing systems. https://code.facebook.com/posts/319004238457019/a-comparison-of-state-of-the-art-graph-processing-systems/, 2016.

[9] E. Abdelhamid, I. Abdelaziz, P. Kalnis, Z. Khayyat, and F. Jamour. Scalemine: Scalable parallel frequent subgraph mining in a single large graph. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 716–727. IEEE, 2016.

[10] E. Abdelhamid, I. Abdelaziz, P. Kalnis, Z. Khayyat, and F. Jamour. Scalemine: Scalable parallel frequent subgraph mining in a single large graph. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 716–727. IEEE, 2016.

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

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

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

[14] M. Aliakbarpour, A. S. Biswas, T. Gouleakis, J. Peebles, R. Rubinfeld, and A. Yodpinyanee. Sublinear-time algorithms for counting star subgraphs via edge sampling. *Algorithmica*, 80(2):668–697, 2018.

[15] M. Aliakbarpour, A. S. Biswas, T. Gouleakis, J. Peebles, R. Rubinfeld, and A. Yodpinyanee. Sublinear-time algorithms for counting star subgraphs via edge sampling. *Algorithmica*, 80(2):668–697, 2018.

[16] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. C. Sahinalp. Biomolecular network motif counting and discovery by color coding. *Bioinformatics*, 24(13):i241–i249, 2008.

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

[18] S. Assadi, M. Kapralov, and S. Khanna. A simple sublinear-time algorithm for counting arbitrary subgraphs via edge sampling. *Innovations in Theoretical Computer Science (ITCS)*, 2018.

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

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

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

[22] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. Tao: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA. USENIX.

[23] J. Brynielsson, J. Högberg, L. Kaati, C. Mårtenson, and P. Svenson. Detecting social positions using simulation. In *2010 International Conference on Advances in Social Networks Analysis and Mining*, pages 48–55. IEEE, 2010.

[24] C. Chen, C. Liang, J. Lin, L. Wang, Z. Liu, X. Yang, J. Zhou, Y. Shuang, and Y. Qi. Infdetect: a large scale graph-based fraud detection system for e-commerce insurance. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 1765–1773. IEEE, 2019.

[25] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng. G-miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–12, 2018.

[26] J. Chen and X. Qian. Dwarvesgraph: A high-performance graph mining system with pattern decomposition. *arXiv preprint arXiv:2008.09682*, 2020.

[27] Q. Chen, B. Tian, and M. Gao. Fingers: exploiting fine-grained parallelism in graph mining accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 43–55, 2022.

[28] X. Chen, R. Dathathri, G. Gill, L. Hoang, and K. Pingali. Sandslash: a two-level framework for efficient graph pattern mining. In *Proceedings of the ACM International Conference on Supercomputing*, pages 378–391, 2021.

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

[30] W. H. Cunningham and J. Edmonds. A combinatorial decom-

position theory. *Canadian Journal of Mathematics*, 32(3):734–765, 1980.

[31] A. Czumaj and C. Sohler. Estimating the weight of metric minimum spanning trees in sublinear time. *SIAM Journal on Computing*, 2009.

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

[33] V. Dias, C. H. Teixeira, D. Guedes, W. Meira, and S. Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1357–1374, 2019.

[34] T. Eden, D. Ron, and C. Seshadhri. On approximating the number of k-cliques in sublinear time. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 722–734, 2018.

[35] T. Eden and W. Rosenbaum. Lower bounds for approximating graph parameters via communication complexity. *arXiv preprint arXiv:1709.04262*, 2017.

[36] T. Eden and W. Rosenbaum. On sampling edges almost uniformly. *arXiv preprint arXiv:1706.09748*, 2017.

[37] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment*, 7(7):517–528, 2014.

[38] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 323–336, 2002.

[39] W. Fan, X. Wang, and Y. Wu. Diversified top-k graph pattern matching. *Proceedings of the VLDB Endowment*, 6(13):1510–1521, 2013.

[40] I. Finocchi, M. Finocchi, and E. G. Fusco. Clique counting in mapreduce: Algorithms and experiments. *Journal of Experimental Algorithmics (JEA)*, 20:1–20, 2015.

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

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

[43] C. Gui, X. Liao, L. Zheng, P. Yao, Q. Wang, and H. Jin. Sumpa: Efficient pattern-centric graph mining with pattern abstraction. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 318–330. IEEE, 2021.

[44] A. P. Iyer, Z. Liu, X. Jin, S. Venkataraman, V. Braverman, and I. Stoica. {ASAP}: Fast, approximate graph pattern mining at scale. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 745–761, 2018.

[45] K. Jamshidi, R. Mahadasa, and K. Vora. Peregrine: a pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[46] C. Jedrzejek, J. Bak, and M. Falkowski. Graph mining for detection of a large class of financial crimes. In *17th International Conference on Conceptual Structures, Moscow, Russia*, volume 46, 2009.

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

[48] F. Khorasani, R. Gupta, and L. N. Bhuyan. Scalable simd-efficient graph processing on gpus. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, PACT '15, pages 39–50, 2015.

[49] M. Kurant, M. Gjoka, Y. Wang, Z. W. Almquist, C. T. Butts, and A. Markopoulou. Coarse-grained topology estimation via graph sampling. In *Proceedings of the 2012 ACM workshop on Workshop on online social networks*, pages 25–30, 2012.

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

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

[52] T. Lipcon, D. Alves, D. Burkert, J.-D. Cryans, A. Dembo, M. Percy, S. Rus, D. Wang, M. Bertozzi, C. P. McCabe, et al. Kudu: Storage for fast analytics on fast data. *Cloudera, inc*, 28, 2015.

[53] D. Mawhirter, S. Reinehr, W. Han, N. Fields, M. Claver, C. Holmes, J. McClurg, T. Liu, and B. Wu. Dryadic: Flexible and fast graph pattern matching at scale. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 289–303. IEEE, 2021.

[54] D. Mawhirter and B. Wu. Automine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 509–523, 2019.

[55] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what {COST}? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[56] K. Michalak and J. Korczak. Graph mining approach to suspicious transaction detection. In *2011 Federated conference on computer science and information systems (FedCSIS)*, pages 69–75. IEEE, 2011.

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

[58] J. Mondal and A. Deshpande. Stream querying and reasoning on social data. In *Encyclopedia of Social Network Analysis and Mining*, pages 2063–2075, 2014.

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

[60] A. Pavan, K. Tangwongsan, S. Tirthapura, and K.-L. Wu. Counting and sampling triangles from a graph stream. *Proc.*

VLDB Endow., 6(14):1870–1881, Sept. 2013.

[61] A. Pavan, K. Tangwongsan, S. Tirthapura, and K.-L. Wu. Counting and sampling triangles from a graph stream. *Proceedings of the VLDB Endowment*, 6(14):1870–1881, 2013.

[62] G. Rao, J. Chen, J. Yik, and X. Qian. Sparsecore: stream isa and processor specialization for sparse computation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 186–199, 2022.

[63] V. Sekar, M. K. Reiter, and H. Zhang. Revisiting the case for a minimalist approach for network flow monitoring. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 328–341, 2010.

[64] T. Shi, M. Zhai, Y. Xu, and J. Zhai. Graphpi: high performance graph pattern matching through effective redundancy elimination. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020.

[65] J. Su, L. He, P. Jiang, and R. Wang. Exploring pim architecture for high-performance graph pattern mining. *IEEE Computer Architecture Letters*, 20(2):114–117, 2021.

[66] N. Talati, H. Ye, S. Vedula, K.-Y. Chen, Y. Chen, D. Liu, Y. Yuan, D. Blaauw, A. Bronstein, T. Mudge, et al. Mint: An accelerator for mining temporal motifs.

[67] N. Talati, H. Ye, Y. Yang, L. Belayneh, K.-Y. Chen, D. T. Blaauw, T. N. Mudge, and R. G. Dreslinski. Ndminer: accelerating graph pattern mining using near data processing. In *ISCA*, pages 146–159, 2022.

[68] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. {RIPQ}: Advanced photo caching on flash for facebook. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pages 373–386, 2015.

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

[70] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu. {RStream}: Marrying relational algebra with streaming for efficient graph mining on a single machine. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 763–782, 2018.

[71] T. Wang, Y. Chen, Z. Zhang, T. Xu, L. Jin, P. Hui, B. Deng, and X. Li. Understanding graph sampling algorithms for social network analysis. In *2011 31st international conference on distributed computing systems workshops*, pages 123–128. IEEE, 2011.

[72] D. Yan, G. Guo, M. M. R. Chowdhury, M. T. Özsu, W.-S. Ku, and J. C. Lui. G-thinker: A distributed framework for mining subgraphs in a big graph. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1369–1380, 2020.

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

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

[75] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[76] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. {FlashGraph}: Processing {Billion-Node} graphs on an array of commodity {SSDs}. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, 2015.

A More Details on Predicate Matching

In predicate "all", queries specify a predicate that is applied to every edge or vertex. For instance, one can query "5-House patterns where all edges have the property NSDI". To support such a query, Arya introduces *a conservative sampling* stage to generate a new graph (of edges and their vertices) from the original graph, where the predicate condition is applied to every edge; and in the sampling phase, Arya runs only on this new property graph. This step ensures that "all" edges match the predicate.

In predicate "at-least-one", queries specify a predicate that is applied to at-least one edge or vertex. For example, one such query is "5-House patterns where at-least-one edge has a weight > 66". To support such predicate queries, we change the runtime workflow to take two passes on the graph. In the first pass, edges matching the predicate of weight > 66 are generated as a new graph. In the second pass, every sampler picks the first edge randomly from the new graph. This ensures that the pattern found by the sampler (if it does find one) meets the predicate. For the rest of the edges, the sampler continues sampling on the original graph, which can add zero or more edges that satisfy the predicate. If a duplicated edge is found, we disregard this sampler. This ensures that the probability analysis of Arya to estimate the error still holds.

Similarly, in the predicate "at-least-percentage", we use the two-pass approach as in the predicate "at-least-one". The only difference is we need to sample a percentage of edges from the newly generated graph.

B Detailed Explanation of Sampling Algorithms

In this section, we introduce the data structures and probability calculation used in the decomposition sampling. Algorithm 4 and Algorithm 2 describe the building blocks of decomposition sampling in our implementation as [18]. We use sampler trees as sampling data structures for maintaining the (inverse) probability. Figure 8 shows a $(2k+1)$-odd cycle sampler tree and an l-star sampler tree representation. Each subpattern (an odd cycle or a star) always has a two-layer sampler tree structure. The first layer is a root node, the second layer contains one or more nodes as leaves and we assign the inverse probability to each leaf.

Odd Cycle Sampler Tree. In Figure 8 (a), $e_1, ..., e_k$ in the cycle sampler tree root denotes first k edges sampled in Algorithm 4 (line 5 and 6) and $n_1, ..., n_b$ in b leaves denotes vertices sampled in line 7 and 8 where $b = \lceil d(u_1)/\sqrt{m} \rceil$. This set of nodes and edges can potentially form at most b $(2k+1)$-odd cycles in the original graph which are represented in b branches in the cycle sampler tree. The inverse probability of one leaf i is $Pr_i[C_{2k+1}] = Pr[e_1] \cdot Pr[e_2] \dots Pr[e_k] \cdot Pr[n_i] = \frac{1}{m}(\frac{1}{2m})^{k-1}\frac{1}{d(u_1)}$ if the root edges and the node in the leaf can form an odd cycle; otherwise, the inverse probability is defined as $Pr_i[C_{2k+1}] = 0$.

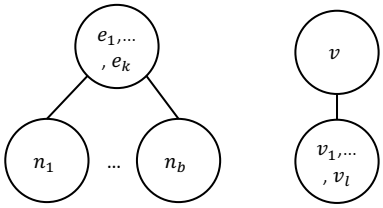
Algorithm 4 Odd Cycle Sampler Tree

1: **Input**:
2: Graph $G = (V, E)$ where $|E| = m$, an odd cycle C_{2k+1}
3: **Output**:
4: A set consisting of C_{2k+1} or 0
5: Sample an edge $e_1 = (u_1, v_1)$ from graph such that $d(u_1) \leq d(v_1)$. ▷ *sample first edge with an order*
6: Sample $k-1$ edges $e_2 \dots e_k$ with replacement from G. ▷ *sample rest edges as the cycle skeleton*
7: **for** $i = 1$ to $\lceil d(u_1)/\sqrt{m} \rceil$ **do** ▷ *Sample last hoop edge*
8: Sample a vertex n_i from the neighbors of u_1.
9: Test if there are edges in G to complete an odd cycle.

Star Sampler Tree. In Figure 8 (b), v in the root node is the central vertex of the star and $v_1, ..., v_l$ in the leaf node are l petals. The inverse probability of the leaf is $Pr[S_l] = Pr[v] \cdot Pr[petal\_vertices] = \frac{d_v}{2m}\binom{d_v}{l}$, where m is the total number of edges in graph G.

Approximation for the Original Pattern. Supposing a pattern P consists of o odd cycles $C_{2k_1+1}, ..., C_{2k_o+1}$ and s stars $S_1, ..., S_s$, and $z = 2o + 2s$. A pattern-sampler tree will be a z-level tree which consists of odd cycle sampler subtrees and star sampler subtrees. To obtain the final pattern-sampler tree, we run subpattern samplers in some order. The pattern-sampler tree keeps extending two new layers by connecting each last-layer leaf-node to a new subpattern subtree root. A final sampler tree is shown in Figure 9 (a). We also show a 5-House sampler tree example in Figure 9 (b). As 5-House is decomposed into a triangle and an 1-star (see Figure 1), the first two layers of 5-House sampler tree represent a triangle sampler tree, and for the last two layers each branch represents a 1-star sampler tree.

A path from the root to a leaf in the pattern sampler tree forms a potential pattern. If path j passes connectivity test with remaining edges, the probability of the path is $Pr[P_j] = Pr[C_j^1] \dots Pr[C_j^o] Pr[S_j^1] \dots Pr[S_j^s]$ because subpatterns are sampled independently. The output of a sampler path is $R[P_j] = \frac{1}{Pr[C_j^1] \dots Pr[C_j^o] Pr[S_j^1] \dots Pr[S_j^s]}$ if it forms a pattern after testing; or $R[P_j] = 0$ if it's not. The estimated pattern number outputs by a pattern sampler tree is the average of each path's estimation output, which is $R[P] = \frac{\sum_{j=1}^{w} R[P_j]}{w}$ supposing we have w final-layer leaves. We aggregate results from all pattern sampler trees as their average number $\frac{\sum_{i=1}^{n} R_i[P]}{n}$, supposing there are n sampler trees and tree i outputs $R_i[P]$. [18] proves the expectation estimated by $R_i[P]$ is the number of pattern P in graph G (denoted as $\#P$) and variance is bounded.



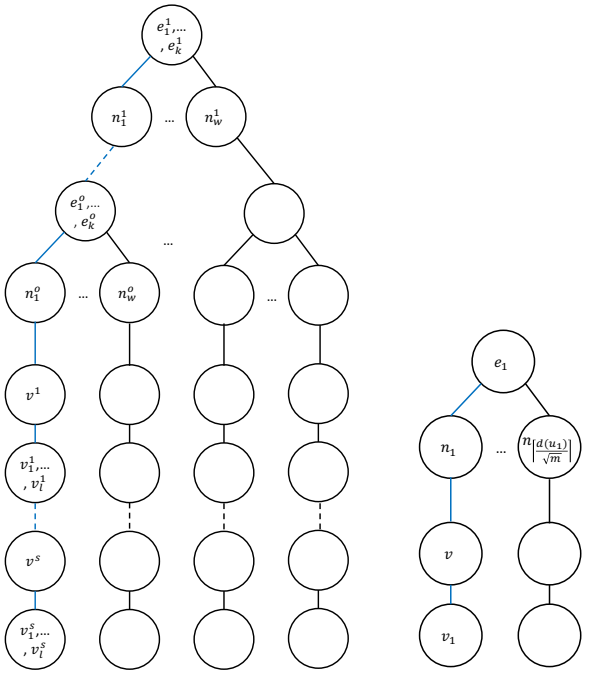
(a) $(2k+1)$-odd-cycle sampler tree (b) l-star sampler tree

Figure 8: **Subpattern sampler trees used in decomposition.** *This figure shows $(2k+1)$-odd-cycle sampler tree and l-star sampler tree each corresponding to Algorithm 4 and 2.*

C Computation-Communication Ratio Analysis in Partitioned-graph Setting

In this section, we calculate computation-communication ratio based on sampler tree implementation. In terms of computation cost, in an l-star sampler, we sample l neighbors randomly and thus the contribution to computation cost is $\Theta(l)$. In a $(2k+1)$-odd cycle sampler, we sample an edge from the graph first, which is $\Theta(1)$, and then sample $k-1$ edges with cost $\Theta(k-1)$. And then we sample $\lceil d(u_1)/\sqrt{m} \rceil$ vertices from the first node u_1's neighbors, which costs $\Theta(d(u_1)/\sqrt{m})$, where m is the number of edges in the large graph and $d(u_1)$ is the degree of the start vertex of the first sampled edge. Let $\overline{\Delta}$ denote the average degree of the large graph. Testing completeness of these $\lceil d(u_1)/\sqrt{m} \rceil$ cycles needs to test k edges, whose cost is $\Theta(k \cdot \overline{\Delta})$ after amortizing since our neighbor checking goes through all the neighbors of the start vertex. We also test the remaining edges of the entire pattern connecting each subpattern, and the computation cost is $\Theta(x \cdot \overline{\Delta})$ supposing there are x remaining edges. Supposing the pattern contains s stars and o odd cycles, the total computation cost for one sampler is $\sum_j^s l_j + \sum_i^o (k_i + \frac{\overline{\Delta}}{\sqrt{m}} + k_i \cdot \overline{\Delta}) + x \cdot \overline{\Delta}$ by amortizing the degrees among multiple sampling trials.

Supposing we have p partitions ($p \geq 2$). In our evaluation, we partition the vertices nearly uniformly by hashing the vertices to a machine, the probability of a sampler may not have a vertex's neighbors locally is $\frac{p-1}{p}$. For an l-star sampler, if the central vertex of this star is local to the machine running the sampler algorithm, the communication cost is 0 because we have all the neighbors of a vertex belonging to the partition stay in the same machine; Otherwise, the communication cost is $\Theta(d(u_{central}))$. Therefore, the l-star sampler communication cost is $\Theta(\frac{p-1}{p} d(u_{central}))$. For a $(2k+1)$-cycle sampler, we first sample an edge, if this edge is not local, we will retrieve all the neighbors for the start vertex. When sampling the next k edges, we may retrieve the neighbor list of a vertex for each edge. Thus the communication cost is $\Theta(\frac{p-1}{p} k \cdot \overline{\Delta}))$. To test these k edges and a neighbor of the first vertex form a cycle, we need to test the connectivity of k remaining edges in the $(2k+1)$-cycle, which cost is $\frac{p-1}{p} k \cdot \overline{\Delta}$.



(a) A pattern sampler tree (b) 5-House sampler tree

Figure 9: **Pattern sampler trees.** *Dotted lines represent there can be multiple other odd cycles or stars in the middle of the layers. Solid lines are connecting root and leaves in a subpattern sampler subtree or from a last cycle leaf node to a first star root node. The blue lines form a path from the root to a last-layer leaf. The labels of some nodes are omitted in the figure.*

Plus the x remaining-edge test of the entire pattern, the total communication cost is $\frac{p-1}{p}(\sum_i^o 2k_i \cdot \overline{\Delta} + \sum_j^s \overline{\Delta} + x \cdot \overline{\Delta})$.

Therefore, the computation-communication cost of partitioned Arya is $\frac{p}{p-1} \cdot \frac{\sum_j^s l_j + \sum_i^o k_i + \sum_i^o (\frac{1}{\sqrt{M}} + k_i) \cdot \overline{\Delta} + x \cdot \overline{\Delta}}{(\sum_i^o 2k_i + s + x) \cdot \overline{\Delta}}$, which is approximately $c\frac{p}{p-1}$ where c is a constant related to the pattern and the graph. This communication cost can be reduced by using batching technique mentioned in Section 5.2.