gSWORD: GPU-accelerated Sampling for Subgraph Counting

Anonymous Author(s)

ABSTRACT

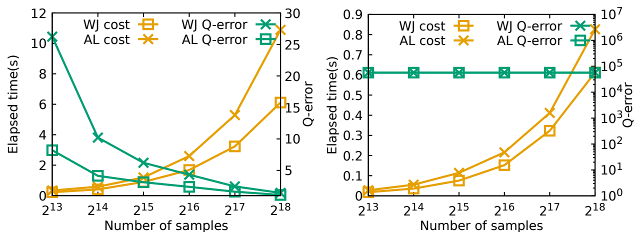
Subgraph counting is a fundamental component for many downstream applications such as graph representation learning and query optimization. Since obtaining the exact count is often intractable, there have been a plethora of approximation methods on graph sampling techniques. Nonetheless, the state-of-the-art sampling methods still require massive samples to produce accurate approximations on large data graphs. We propose gSWORD, a GPU framework that leverages the massive parallelism of GPUs to accelerate iterative sampling algorithms for subgraph counting. Despite the embarrassingly parallel nature of the samples, there are unique challenges in accelerating subgraph counting due to its irregular computation logic. To address these challenges, we introduce two GPU-centric optimizations: (1) *sample inheritance*, enabling threads to inherit samples from neighboring threads to avoid idling, and (2) *warp streaming*, effectively distributing workloads among threads through a streaming process. Moreover, we propose a CPU-GPU co-processing pipeline that overlaps the sampling and enumeration processes to mitigate the underestimation issue. Experimental results demonstrate that deploying state-of-the-art sampling algorithms on gSWORD can perform millions of samples per second. The co-processing pipeline substantially improves the estimation accuracy in the cases where existing methods encounter severe underestimations with negligible overhead.

ACM Reference Format:

Anonymous Author(s). 2023. gSWORD: GPU-accelerated Sampling for Subgraph Counting. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Subgraph counting determines the number of subgraphs in a data graph G isomorphic to a connected query graph q. As one of the fundamental graph processing operations, subgraph counting facilitates many downstream applications like graph kernels for representation learning [34, 39] and probabilistic models for image segmentation [17, 50]. However, the subgraph isomorphism problem, which decides whether there exists a subgraph of G isomorphic to q, is NP-complete [9]. The counting problem is more challenging as millions or billions of instances are typically found even in relatively small graphs. Hence, researchers propose a variety of approximation approaches [8, 21, 24, 28, 31] to estimate the count instead of computing the exact value. Particularly, the methods



(a) eu2005 (b) WordNet

Figure 1: The q-error and CPU runtime by WanderJoin and Alley for a query of 8 vertices in eu2005 and WordNet.

based on random walks (RW) gain great interest due to their superior performance over other approximation approaches in terms of both accuracy and efficiency [21, 28].

RW estimators compute the frequency of q appearing in G by executing a set of samples, each of which independently extracts a subgraph from G. A sample starts from a subgraph G' having a seed vertex, expands G' by adding one vertex at each iteration, and terminates until G' has the same number of vertices as q. The sample quality (i.e., the chance of a sample leading to G' isomorphic to q) has a key impact on the sampling error of the estimation. As such, studies on RW-based techniques [21, 23, 24, 54] focus on optimizing the strategies of expanding G' to improve the quality of samples, which makes RW estimators mainly differ in the methods of sampling the vertices to extend G'. Take two state-of-the-art approaches, WanderJoin [24] and Alley [21], as examples. At each step, WanderJoin samples a vertex from vertices adjacent to G' because the query graph q is connected. To further improve the sample quality, Alley maintains a candidate set for sampling and refines the set at each step based on the vertex just added to G'.

To investigate the performance factors of WanderJoin and Alley, we conduct a preliminary experiment on a query with 8 vertices in eu2005 and WordNet dataset. Figure 1 presents the estimation error (q-error) and runtime with the number of samples increasing. In eu2005, both methods produce more accurate estimates with an increase in the sample size. Although Alley converges faster than WanderJoin, it takes longer time to process the same number of samples due to the cost of *refining* candidate sets. Further, both WanderJoin and Alley exceed 6 seconds to push the q-error below 1.5, which is equivalent to a 50% relative error to the true value. In WordNet, both methods yield poor estimates. This is because it is hard to finding a valid instance of the query graph even given a large number of samples. As a result, the estimated count is empty, significantly deviating from the true count. In summary, the RW approaches to subgraph counting are still computationally expensive to give an accurate estimation, and cannot meet the rigid requirement of many time-critical applications such as dynamic network analysis [12, 13] and query optimization [2, 53].

With thousands of cores, GPUs outperform CPUs in raw computational power, making them well-suited for embarrassingly parallel workloads like RW estimators, where each core can independently process samples. Some recent studies propose to accelerate conventional RW workloads on GPUs [18, 27, 41]. However, adapting subgraph counting estimators to GPUs is a non-trivial task. Specifically, we encounter three challenges given the unique characteristics of RW estimators for subgraph counting and the SIMT (single instruction multiple thread) computational paradigm of GPUs.

- **Validate Imbalance.** Conventional RW samples often terminate after performing the same number of iterations, and therefore fit into SIMT seamlessly because threads in a warp (i.e., the basic execution unit in GPUs) must execute the same instruction. In contrast, samples in RW estimators may stop at any iteration if the extracted subgraph fails to pass the *Validate* step and cannot be extended further to an isomorphic subgraph. As a result, thread resources are underutilized. One appealing approach to address this issue is to start a new sample *immediately* after an existing one becomes invalid. However, our experiments reveal that this method performs poorly in practice due to suboptimal memory access patterns.
- **Refine Imbalance.** Conventional RW samples simply select a vertex from the neighbors of the current residing vertex at each iteration, the computation of which is lightweight. Thus, the workloads of threads in a warp are similar. By comparison, samples in RW estimators require complex operations (e.g., set intersections on neighbor sets of selected vertices) to refine candidate sets at each iteration. As such, threads in a warp can be fed with heavily skewed workloads given the high variance of vertex degrees in real-world graphs. This phenomenon leads to poor performance especially for threads in the same warp because these threads execute in lockstep and their speed is determined by the slowest one.
- **Underestimation.** Although the RW estimators are theoretically unbiased, they could encounter the problem of underestimation in practical scenarios [21, 52]. This issue arises from the low percentage of valid samples, particularly for larger queries on datasets with skewed distributions, where finding isomorphic subgraphs is challenging. In extreme cases, such as the `WordNet` dataset illustrated in Figure 1, no valid samples are found. As a result, the RW estimators can only generate an empty estimate. This highlights the limitations of the RW estimators on datasets that have complex isomorphic subgraph distributions.

In this paper, we present `gSWORD`, a novel GPU-based framework to accelerate the RW-based sampling process for subgraph counting. We design an iterative workflow consisting of *Refine-Sample-Validate* steps to unify the state-of-the-art RW estimators, i.e., `WanderJoin` and `Alley`. At each iteration of a sample, the *Refine* step first prunes the candidate sets to reduce the sample space size, the subsequent *Sample* step selects a vertex from the refined sets to extend the extracted subgraph G', and the *Validate* step finally decides whether the sample can be stopped given G'. In the workflow, we design novel optimization techniques to overcome the challenges associated with processing RW workloads for subgraph counting on GPUs.

First, we propose a *sample inheritance* strategy to address the validate imbalance. When a thread invalidates a sample, we keep the thread busy by *inheriting* a valid sample from another thread in the same GPU warp. Note that the inherited samples lead to biased estimations if they are treated independently. Hence, we propose an efficient method to adjust the sample weights according to a recursive yet unbiased estimator. In this way, we collected more samples while executing the same number of iterations.

Second, we devise a *warp streaming* strategy for handling the refine imbalance. If a sample contains enough refinement workloads to be parallelized by a warp, we dynamically stream its workloads to the warp where a member thread is assigned to a candidate at a time. Once a candidate passes the refinement step, the thread sends the candidate to a reservoir sampler [21]. Thereby, we keep all threads occupied while ensuring the vertices are desirably sampled from the refined candidate set after the streaming process.

Third, we propose a *trawling* strategy that dynamically transitions selective samples to the enumeration process. By leveraging the enumerated samples, we address the issue of underestimation caused by a lack of valid samples. To minimize the computational overhead of graph enumeration, we design a CPU-GPU co-processing pipeline. This pipeline effectively overlaps the GPU-based sampling process with the CPU-based enumeration process, resulting in little additional cost.

We thereby summarize our contributions as follows:

- We introduce `gSWORD`, a unified workflow for accelerating RW estimators for subgraph counting on GPUs. To the best of our knowledge, this is the first study of its kind.
- We propose two novel GPU-centric optimizations, i.e., *sample inheritance* and *warp streaming*, to address the workload imbalance issues of RW estimators.
- We devise the *trawling* strategy to discover more valid samples, along with a CPU-GPU co-processing pipeline that overlaps the sampling and enumeration processes with negligible overhead.
- The experimental results reveal that `gSWORD` significantly improves the efficiency of RW estimators over the CPU and GPU baselines. In particular, `gSWORD` achieves 341x speedup on average over the CPU baselines and 9x speedup for the GPU baselines. Further, the trawling strategy shows orders of improvement on q-error on cases where existing RW estimators have severe underestimation issues.

The rest of the paper is organized as follows. Section 2 introduces the background on subgraph counting. In Section 3, we propose the workflow of `gSWORD` as well as the discussions on the framework implementation. Section 4 presents the optimization techniques for workload imbalance. Section 5 presents our CPU-GPU co-processing pipeline. The experimental evaluations are discussed in Section 6. Finally, we discuss the related works and conclude the paper in Sections 7 and 8, respectively.

2 PRELIMINARIES

2.1 Notations and Problem Definition

Given a graph $g = (\mathcal{V}_g, \mathcal{E}_g, \mathcal{L}_g)$, $\mathcal{V}_g$ represents the vertex set, $\mathcal{E}_g$ represents the edge set, $\mathcal{L}_g$ is a function that maps a vertex $v \in \mathcal{V}_g$ to a label l. q and G denote the query graph and the data graph respectively. We call vertices and edges of q (resp. G) query vertices and query edges (resp. data vertices and data edges), respectively. For ease of presentation, we focus on undirected graphs but our approaches can support directed graphs. A graph g is isomorphic to g' if there exist a *graph isomorphism* (Definition 1) from g to g'.

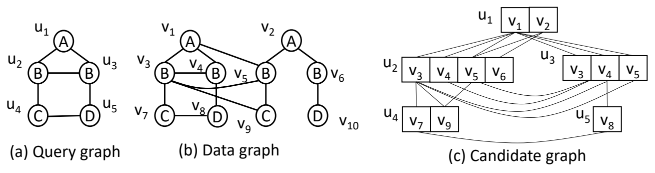


Figure 2: Query graph, data graph and candidate graph.

Definition 1 (Graph Isomorphism). *Given g and g', a graph isomorphism from g to g' is a bijective function $f: \mathcal{V}_g \to \mathcal{V}_{g'}$ s.t.*

- *$\forall u \in \mathcal{V}_g, \mathcal{L}_g(u) = \mathcal{L}_{g'}(f(u))$; and*
- *$\forall u, v \in \mathcal{V}_g, e(u,v) \in \mathcal{E}_g \iff e(f(u), f(v)) \in \mathcal{E}_{g'}$.*

Problem Definition. The subgraph counting problem finds the number of subgraphs in the data graph G that are isomorphic to the query graph q. For brevity, each isomorphic subgraph in G is called an *instance* of q.

Following the literature [36], we present some important concepts used in the subgraph counting algorithms.

Definition 2 (Matching Order). *The matching order φ is a permutation of query vertices for iterative matching to data vertices. $\varphi[i]$ denotes the i-th query vertex to be matched.*

Definition 3 (Partial Instance). *A partial instance in G is a prefix match of the first i query vertices by φ where $i \leq |\varphi|$.*

Definition 4 (Candidate Sets). *A global candidate set $C(u)$ of a query vertex u is a set of data vertices such that for each $v \in \mathcal{V}_G$, if a mapping (u, v) exists in an instance of q in G, then $v \in C(u)$. Given a query edge $e(u, u')$ and a data vertex $v \in C(u)$, the local candidate set $C(u, u', v)$ is the set of neighbors of v belonging to $C(u')$, i.e., $C(u, u', v) = N(v) \bigcap C(u')$.*

Definition 5 (Candidate Graph). *A candidate graph for q on G consists of the global candidate set $C(u)$ for each $u \in \mathcal{V}_q$. Further, there is an edge connecting candidates $v \in C(u)$ and $v' \in C(u')$ if $e(u, u') \in \mathcal{E}_q$ and $e(v, v') \in \mathcal{E}_G$.*

Example 1. *In Figure 2, we present a data graph G, a query graph q and the corresponding candidate graph. We set the matching order as $\varphi = (u_1, u_2, u_3, u_4, u_5)$. (v_1, v_3) is a partial instance for q where v_1 matches u_1 and v_3 matches u_2. For the same reason, (v_1, v_4), (v_1, v_5), (v_2, v_5) and (v_2, v_6) are all partial instances. There is only one instance $(v_1, v_3, v_4, v_7, v_8)$ of q in G. In the candidate graph, $C(u_2) = \{v_3, v_4, v_5, v_6\}$ is a global candidate set and $C(u_2, u_4, v_3) = \{v_7, v_9\}$ is a local candidate set. Note that all instances can be found in the candidate graph but the candidate graph may store vertices and edges that are not included in any instance of q in G, e.g., vertex v_2 and edge $e(v_2, v_6)$.*

2.2 RW Estimators for Subgraph Counting

RW estimators are effective approaches for subgraph counting with theoretical guarantees. For a query graph q, RW estimators iteratively sample data vertices to match with q. Note that RW estimators are originally designed to sample on the data graph G directly, but it is straightforward to sample on the candidate graph for reducing the sample space. Here, we briefly review `WanderJoin` [24] and `Alley` [21] as the state-of-the-art RW estimators.

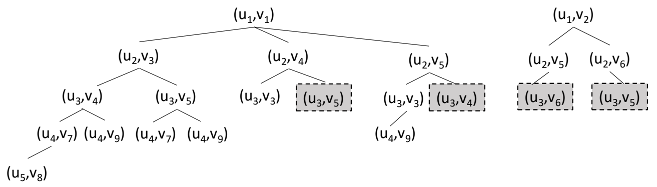


Figure 3: Sample space of RW estimators. The shaded nodes are in the sample space of `WanderJoin` but not `Alley`.

WanderJoin. Each sample of `WanderJoin` is obtained by performing RW on the candidate graph. A sample consists of a sequence of data vertices where the first vertex is sampled from a global candidate set, and each subsequent vertex is sampled from a local candidate set based on a vertex that has been previously selected into the sequence. If the sampled sequence is not a valid partial instance by the validity check, then `WanderJoin` starts a new sample sequence. Given a sequence s, the probability of sampling s is $\prod_{i=1}^{|s|} \frac{1}{|C_i|}$ where C_i is the candidate set of the i-th vertex.

Alley. A major drawback for `WanderJoin` is that many samples can be invalid and thus creates high variance of estimation. To reduce the number of invalid samples, `Alley` imposes a *refinement* step on the candidate set before selecting the next vertex. The refinement step guarantees that each vertex in the refined candidate set always yields a *valid* partial instance. When a sample sequence encounters an empty refined candidate set, `Alley` starts a new sample sequence. Given a sequence s, the probability of sampling s is $\prod_{i=1}^{|s|} \frac{1}{|C_i|}$ where C_i is the refined candidate set of the i-th vertex.

Remark: `Alley` [21] proposes two optimizations, *branching* and *synopses*, to improve the sampling performance. Given a branching factor b, *branching* samples b vertices at each step, and therefore a sample generates a tree consisting of multiple paths. Compared with sampling a vertex (i.e., $b = 1$) at each step, *branching* can reduce the execution time of sampling the same number of paths because candidate sets generated in a tree can be shared by multiple paths. Orthogonal to sampling, *synopses* is an indexing technique to prune the paths that cannot lead to valid results. Particularly, it performs a number of random walks to build an index for the patterns hard to be processed by sampling.

However, we have chosen not to incorporate these two optimizations into our GPU-based framework. The maintenance of a tree sample for branching introduces complex control flows and frequent random accesses, making it unsuitable for the SIMT architecture of GPUs. Additionally, the construction of *synopses* requires several hours to build the index [21] and is unable to handle graphs with frequent updates due to the associated overhead.

HT estimator. With the sample probability, both `WanderJoin` and `Alley` employs the Horvitz-Thompson (HT) estimator to approximate the subgraph count.

Definition 6. *Let Y_i, $i = 1, 2, \ldots, n$ be an independent sample and π_i is the inclusion probability of sampling Y_i. The Horvitz-Thompson estimator of the mean is given by $\frac{\sum_{i=1}^{n} Y_i/\pi_i}{n}$.*

In the context of `WanderJoin` and `Alley`, π_i is the probability of a sample s_i, and Y_i denotes an indicator random variable $\mathbb{I}(s_i)$ where s_i is an instance of q in G. Hence, the estimator for `WanderJoin` and `Alley` is expressed as:

$$H = \frac{\sum_{i=1}^{n} \left(\left(\prod_{j=1}^{d} |C_{ij}| \right) \mathbb{I}(s_i) \right)}{n} \tag{1}$$

where C_{ij} denote the candidate set (possibly refined) for query vertex u_j in sample s_i. If s_i finds a match, then the indicator is 1. Otherwise, it is 0.

Example 2. *Figure 3 shows the sample space for* `WanderJoin` *and* `Alley` *according to the matching order $\varphi = (u_1, u_2, u_3, u_4, u_5)$ on the candidate graph in Figure 2.* `WanderJoin` *may sample a sequence $s_1 = (v_1, v_4, v_3)$ with a probability $\frac{1}{2} \cdot \frac{1}{3} \cdot \frac{1}{2} = \frac{1}{12}$.* `Alley` *may sample the same sequence s_1, but with a higher probability $\frac{1}{2} \cdot \frac{1}{3} \cdot 1 = \frac{1}{6}$. This is because when $s_1 = (v_1, v_4)$,* `Alley` *refines the candidate set for u_3 by removing the candidate v_5 since v_5 is not a common neighbor of v_1 and v_4. Hence, the sample space of* `Alley` *is smaller compared with that of* `WanderJoin`*. Both* `WanderJoin` *and* `Alley` *have the same probability to sample the only instance of q as $s_2 = (v_1, v_3, v_4, v_7, v_8)$, i.e., $\frac{1}{2} \cdot \frac{1}{3} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot 1 = \frac{1}{24}$. With s_1 (invalid sample) and s_2 (valid sample), both* `WanderJoin` *and* `Alley` *yield the same estimate as $\frac{0+24}{2} = 12$.*

3 GSWORD FRAMEWORK

3.1 Framework Design

gSWORD is a generic GPU sampling framework for RW estimators on subgraph counting. All RW samples are independent and the estimated value of each sample is aggregated using the HT estimator. As the number of RW samples increases, the accuracy of the estimator improves. Our system enhances subgraph counting performance by gathering more samples within a given time budget. For each iteration of a RW sample, a new vertex is sampled. To ease the development of RW estimators on GPUs, we abstract a sampling iteration into the *Refine-Sample-Validate* (RSV) steps.

- `Refine`. A refined set is computed based on a local candidate set for sampling the next vertex. The estimation wrt. a sample is more accurate when the refined set is smaller.
- `Sample`. One of the vertex in the refined set is sampled as a new vertex for the sample.
- `Validate`. We check if the sample remains a valid partial/full instance after the new vertex is added. The sample is terminated when an invalid instance is found.

Candidate Graph Format. We design a general yet efficient format to store the candidate graph *cg* as depicted in Figure 4. All query edges $e(u, u')$ are stored in a compressed sparse row (CSR) format, i.e., the first two arrays in *cg* are the offset list and the edge list of the query graph CSR respectively. For each edge $e(u, u')$, the candidates for u and the corresponding candidates for u' are stored using a second and a third CSRs. Our design enables efficient lookup for global and local candidate sets. Given a query edge $e(u, u')$ and a data vertex $v \in C(u)$, we use the first CSR to locate the edge. Then, we search the second CSR for v as a global candidate for u, and the local candidate set $C(u, u', v)$ is found in the third CSR.

Example 3. *In Figure 4, the query edge $e(u_1, u_2)$ found in the first CSR corresponds with two tuples (u_1, u_2, v_1) and (u_1, u_2, v_2) where $v_1, v_2 \in C(u_1)$ are stored in the second CSR. The local candidate set $C(u_1, u_2, v_1) = \{v_3, v_4, v_5\}$ can be retrieved from the third CSR.*

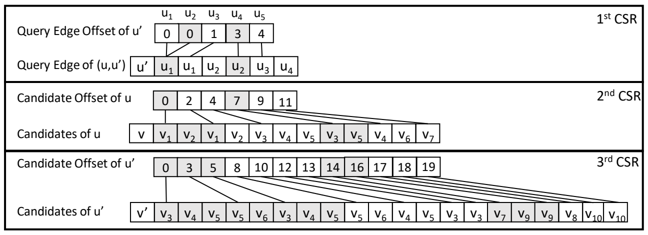


Figure 4: Efficient candidate graph format.

gSWORD framework can easily support existing RW estimators, such as `WanderJoin` and `Alley`. Furthermore, it also allows users to develop and test new RW estimators with ease. Due to space limit, we put the detailed implementation of `WanderJoin` and `Alley` within our framework in Appendix.

3.2 Framework Implementation

Algorithm 1: The RSV abstraction of gSWORD.

```
   Input  : Candidate graph $cg$, block sample pool $bp$
   Output : Thread HT estimation $H$, sample size $nSample$
1  refine ← Init();
2  nSample ← 0;
3  H ← 0;
4  while $bp \neq \emptyset$ do
5      s ← FetchSampleTask(bp);
6      d ← 1;
7      while $d \leq |\mathcal{V}_q|$ do
8          (cand,clen) ← GetMinCandidate(cg,s,d);
9          rlen ← Refine(s,d,cand,clen,refine);
10         (v,prob) ← Sample(s,d,refine,rlen);
11         valid ← Validate(s,d,v,prob);
12         break if valid == false;
13         d ← d + 1;
14     if $d == |\mathcal{V}_q| + 1$ then
15         H ← H + $\frac{1}{s.prob}$;
16     nSample ← nSample+1;
17 return ($H$, $nSample$);        ▷ Estimate aggregation omitted.
```

Algorithm 1 presents an overview of the gSWORD framework. Lines 1-3 initialize the data structures for each thread. Since each sample may incur varying computational cost, distributing a fixed number of samples to each thread results in workload-imbalance. Thus, we devise a sample pool *bp* for each GPU block where all threads share the workload within the same block. Lines 4-5 of Algorithm 1 depict the block sharing approach where a new sample task is fetched from *bp* as long as there remains some samples to be computed in the block. The fetch operation is efficiently handled by *atomic* instructions in the shared memory. After obtain a sample task *s*, we iteratively sample vertices into *s* (Lines 7-13). For each sample iteration, we retrieve the *smallest* candidate set *cand* with its length *clen* from the candidate graph *cg* and send *cand* for refinement. Note that *cand* is the local candidate set given the partial result rather than the global candidate set of a query vertex. Then, we sample a vertex *v* from the refined set *refine* as the *d*-th vertex of the instance in s. If s remains to be a valid partial instance after adding v, we continue to the next sample iteration. Otherwise, the inner loop terminates and a new sample task is fetched from bp. Once an instance of $|\mathcal{V}_q|$ vertices is obtained, we update the HT estimator value with the sampling probability of s in Lines 14-15. Finally, we return the HT estimator value and the sample size within a thread. We omit the estimate aggregation among all threads since it can be directly implemented by the efficient parallel reduce for GPUs [7].

Sample Synchronization vs. Iteration Synchronization. In the current implementation, threads in a warp are synchronized *implicity* on completing the samples. Each thread will wait for the other threads in the warp to finish before fetching a new sample task. We call such an approach *sample synchronization*. The problem with this approach is *warp divergence* because threads in a warp process different samples in lock steps, and the samples can terminate at different iterations. To address this issue, *iteration synchronization* is an alternative approach where threads synchronize after each vertex is sampled (one iteration of the inner while loop in Algorithm 1). In fact, iteration synchronization is the common approach widely adopted in GPU graph processing frameworks [26, 43, 51] as well as GPU-based subgraph enumeration [15, 22, 37, 46].

The benefit of the alternative is: without waiting for other threads in the warp to complete their current samples, a thread can start a new sample at any iteration once the current sample is invalid. To enable iteration synchronization on GPUs, we can place an *explicit* barrier after each iteration and start a new sample immediately once a current sample is invalid. Surprisingly, iteration synchronization is even slower than sample synchronization with an average run time slowdown of 1.3 times.

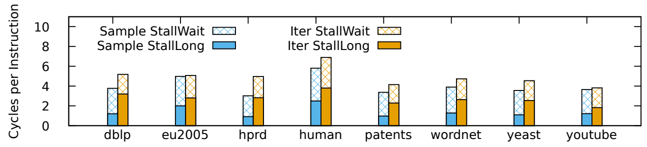
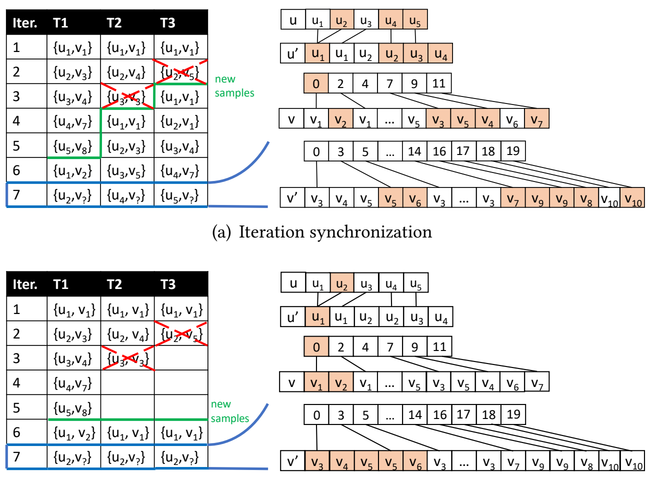


Figure 5: Micro-benchmark on the warp efficiency of sample and iteration synchronization with nsight.

Micro-benchmark. We use the *nsight* profiler to examine the top-2 warp stall factors for the sample and iteration synchronization methods (`Alley` is used as the sampling method). Figure 5 presents the profiling results. *StallLong* is the number of cycles stalled for memory load instructions. *StallWait* is the number of cycles stalled due to instruction loading issues. Although the iteration synchronization has better instruction-level parallelism (i.e., fewer StallWait cycles), the benefit is overwhelmed by the cost of memory accesses (i.e., more StallLong cycles).

Our investigation reveals that *irregular* memory access pattern of the iteration synchronization leads to these results. Specifically, for iteration synchronization, threads in a warp can process different query vertices at an iteration, which leads to the access of candidate sets of different query vertices. By comparison, for sample synchronization, threads in a warp always process the same query vertex at an iteration, so the same candidate set is usually accessed, which has a much better memory locality. More details are explained in Example 4. Compared with subgraph enumeration



Iter.	T1	T2	T3
1	$\{u_1,v_1\}$	$\{u_1,v_1\}$	$\{u_1,v_1\}$
2	$\{u_2,v_3\}$	$\{u_2,v_4\}$	~~$\{u_2,v_5\}$~~
3	$\{u_3,v_4\}$	~~$\{u_3,v_3\}$~~	$\{u_1,v_1\}$
4	$\{u_4,v_7\}$	$\{u_1,v_1\}$	$\{u_2,v_1\}$
5	$\{u_5,v_8\}$	$\{u_2,v_3\}$	$\{u_3,v_4\}$
6	$\{u_1,v_2\}$	$\{u_3,v_5\}$	$\{u_4,v_7\}$
7	$\{u_2,v_?\}$	$\{u_4,v_?\}$	$\{u_5,v_?\}$

new samples
u | u_1 | u_2 | u_3 | u_4 | u_5
u' | u_1 | u_1 | u_2 | u_2 | u_3 | u_4
0 | 2 | 4 | 7 | 9 | 11
v | v_1 | v_2 | v_1 | ... | v_5 | v_3 | v_5 | v_4 | v_6 | v_7
0 | 3 | 5 | ... | 14 | 16 | 17 | 18 | 19
v' | v_3 | v_4 | v_5 | v_5 | v_6 | v_3 | ... | v_3 | v_7 | v_9 | v_9 | v_8 | v_{10} | v_{10}

(a) Iteration synchronization

Iter.	T1	T2	T3
1	$\{u_1, v_1\}$	$\{u_1, v_1\}$	$\{u_1, v_1\}$
2	$\{u_2,v_3\}$	$\{u_2, v_4\}$	~~$\{u_2, v_5\}$~~
3	$\{u_3,v_4\}$	~~$\{u_3, v_3\}$~~	
4	$\{u_4,v_7\}$		
5	$\{u_5,v_8\}$		
6	$\{u_1, v_2\}$	$\{u_1, v_1\}$	$\{u_1, v_1\}$
7	$\{u_2,v_?\}$	$\{u_2,v_?\}$	$\{u_2,v_?\}$

new samples
u | u_1 | u_2 | u_3 | u_4 | u_5
u' | u_1 | u_1 | u_2 | u_2 | u_3 | u_4
0 | 2 | 4 | 7 | 9 | 11
v | v_1 | v_2 | v_1 | ... | v_5 | v_3 | v_5 | v_4 | v_6 | v_7
0 | 3 | 5 | ... | 14 | 16 | 17 | 18 | 19
v' | v_3 | v_4 | v_5 | v_5 | v_6 | v_3 | ... | v_3 | v_7 | v_9 | v_9 | v_8 | v_{10} | v_{10}

(b) Sample synchronization

Figure 6: Access patterns of iteration and sample synchronization, the table shows data vertex sampled at each step.

where the processing workload is heavy due to large intermediate results, the workload for each RW sample is lightweight benefiting from the pruning of the candidate graph. Thus, the nature of the RW samples favors a regular memory access pattern even at the cost of the workload imbalance. Thus, we implement the sample synchronization approach in `gSWORD`.

EXAMPLE 4. *Figure 6 illustrates the process of sample synchronization and iteration synchronization with* `Alley` *as the RW estimator. For brevity, a warp has three threads T1, T2 and T3 in the example. Each thread executes a sample on graphs in Figure 2. The table lists query vertices processed at each iteration. Samples in T2 and T3 terminate at iteration 3 and 2, respectively, since their instances are invalid. The method with iteration synchronization immediately starts a new sample, whereas the sample synchronization method starts a new sample for each thread until all threads complete the samples assigned initially.*

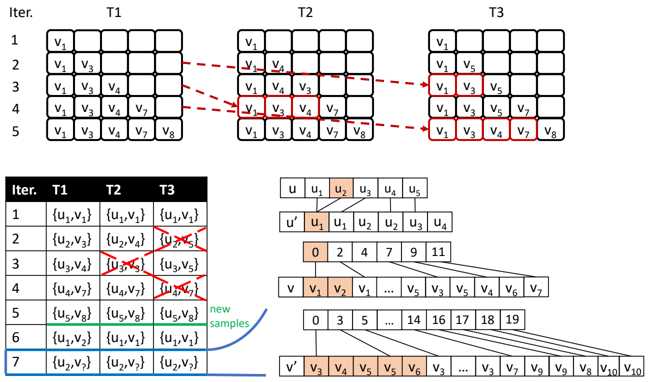
Candidates highlighted on the CSRs are accessed by threads at iteration 7. Iteration synchronization accesses candidates scattered across the candidate graph because threads process different query vertices. In contrast, sample synchronization has much better spatial locality because all threads process the same query vertex. As a result, sample synchronization outperforms the iteration synchronization alternative despite the wasted thread resources.

4 GPU-CENTRIC OPTIMIZATIONS

As discussed in Section 3.2, processing different samples by parallel threads independently can lead to severe workload imbalance. There are two major sources of imbalance: *validate imbalance* and *refine imbalance*, which occurs in the `Validate` and `Refine` step. In this section, we introduce two GPU-centric optimization techniques, i.e., *sample inheritance* and *warp streaming* to address these two sources of workload imbalance respectively.

4.1 Sample Inheritance

With the sample synchronization, samples from different threads may terminate at any iteration after the `Validate` step. The threads



Iter.	T1	T2	T3
1	$\{u_1,v_1\}$	$\{u_1,v_1\}$	$\{u_1,v_1\}$
2	$\{u_2,v_3\}$	$\{u_2,v_4\}$	~~$\{u_2,v_5\}$~~
3	$\{u_3,v_4\}$	~~$\{u_3,v_3\}$~~	$\{u_3,v_5\}$
4	$\{u_4,v_7\}$	$\{u_4,v_7\}$	~~$\{u_4,v_7\}$~~
5	$\{u_5,v_8\}$	$\{u_5,v_8\}$	$\{u_5,v_8\}$
6	$\{u_1,v_2\}$	$\{u_1,v_1\}$	$\{u_1,v_1\}$
7	$\{u_2,v_?\}$	$\{u_2,v_?\}$	$\{u_2,v_?\}$

Figure 7: Example of sample inheritance.

with terminated samples will remain idle while other threads in the same warp are still busy with active workload, hence leading to *validate imbalance*. To address this issue, we propose the sample inheritance approach. Once a thread discovers an invalid sample, the thread will immediately "inherit" a valid partial sample from one of its neighborhood threads in the same warp. In this way, all threads in the same warp are always working on the same iteration and thus render a cohesive data access pattern. Despite additional workload assigned, sample inheritance (gSWORD) only incurs marginal memory access overhead while processing additional samples.

Example 5. *Figure 7 explains the sample inheritance approach. Following Example 4, Threads T1, T2,and T3 belong to the same warp, and each starts a new sample independently. The partial instance* (v_1, v_5) *from T3 is invalid at iteration 2. T3 randomly selects a thread in the warp to inherit the search path. Here, T3 inherits the partial instance* (v_1, v_3) *from T1. The instance* (v_1, v_4, v_3) *from T2 is invalid since* v_4 *maps to* u_2 *but it does not connect to a data vertex with label* C*. Thus, T2 inherits the instance* (v_1, v_3, v_4) *from T1. At iteration 4, the instances from T3 become invalid again, so it inherits the same instance held by T1, i.e.,* (v_1, v_3, v_4, v_7)*. At the last iteration, all threads sample the last vertex for their respective instances. Note that all threads are occupied with sampling tasks when inheritance is enabled. The data access pattern is cohesive as the threads always process the same query vertex on the same iteration (bottom right of Figure 7).*

Although we keep the threads busy with the inheritance optimization, the inherited samples are correlated and biased. This is because more samples will be generated towards larger search tree branches as the partial instances have higher chance to be valid.

Example 6. *In Figure 7, there are three initial samples managed by each thread. Further, there are three additional inherited samples (indicted by the dashed boxes). If we treat all samples independent, the estimate is* $\frac{1}{6} \cdot (3 \cdot 24 + 3 \cdot 0) = 12$ *(3 validate samples each with a sample probability of* $1/24$ *and 3 invalid samples), which significantly deviates from the ground-truth value of 1.*

To eliminate the bias, we observe that once a partial instance is inherited, the inherited samples can be used to collectively estimate the new search space determined by the inherited partial instance. Based on this observation, we present our refine inheritance optimization in Algorithm 2. Instead of terminating an invalid sample in

Algorithm 2: Refine Inheritance

```
  // Replace Line 12 in Algorithm 1
1 if _any(valid) then
2     parentId ← _ballot(valid);
3     idleThreads ← _reduce_sum(valid==false);
4     if threadId == parentId then
5         s.prob = s.prob/(idleThreads+1);
6     if valid==false then
7         s ← _shfl(s,parentId);
8 else
9     break;
```

Algorithm 1, a thread participates in a voting process to see if there are still valid partial samples to be inherited in its warp. We take advantage of *warp-level primitives* to enable efficient collaboration via registers across the threads (functions with underline prefix).

If there exists any valid partial sample in the warp indicated by the _any primitive, a thread holding a valid sample s is selected as the parent using _ballot. In Line 3, we count the number of idle threads that run into invalid samples. Since there are in total $idleThreads+1$ threads that will inherit s from the parent (including the parent itself), we update the probability of s before the parent shares its sample with all idle threads using the _shfl primitive. After the inheritance process, all threads hold a valid partial sample and can proceed with the next sample iteration as in Algorithm 1.

Example 7. *At iteration 4, T3 inherits the partial instance* $s = (v1, v3, v4, v7)$ *from T1 and we obtain an estimate of* $\frac{1}{2} \cdot (1 + 1) = 1$ *conditioned* s *(from the two valid partial samples of T1 and T3).*

Recursive Estimator. Despite its simplicity, Algorithm 2 is in fact a *push-down* evaluation of a recursive estimator. Let $R_i(s)$ denote the estimator at level i of the search tree given a partial instance s. $R_i(s)$ is recursively estimated as follows:

$$R_i(s) = \begin{cases} \sum_t \dfrac{R_{i-1}(s \cup \{v_t\})}{n_i}, & \text{if } i > 1 \text{ and s is inherited to t} \\ R_{i-1}(s \cup \{v\}), & \text{if } i > 1 \text{ and s is not inherited} \\ \dfrac{\mathbb{I}(s)}{\mathbb{P}(s)}, & \text{if } i = 1 \text{, aka. the leaf level} \end{cases}$$

where n_i denote the number of inherited samples at level i.

Push-down evaluation. A thread initially holds an empty sample s at the root level, i.e., $R_{|\mathcal{V}_q|}(s = \emptyset)$. For each iteration, the thread moves to a lower level on the search tree until it reaches the leaf level. The normalization factors n_i are all pushed down via the sample probability $\mathbb{P}(s)$ to the leaf level. The push down evaluation enables efficient one-pass estimation *without* recursive calls for backward propagation of the values from the leaf to the root. We prove that R is an unbiased estimator as the HT estimator H.

Theorem 1. $\mathbb{E}[R_i(s)] = \mathbb{E}[H_i(s)]$ *for any given partial instance* s *where* $H_i(s)$ *denote the HT estimator in Equation 1 to sample* i *vertices without inheritance and* $i \in [1, |\mathcal{V}_q|]$.

Discussion. The sample inheritance optimization shares similarities with the branching technique of Alley in a sense that both sample multiple vertices from the candidate set given a partial instance. However, branching *always* selects multiple vertices when the size of a candidate size is greater than eight, which results in a sample tree and the size of which cannot be determined in advance. This introduces complicated control logic and dynamic memory management, which is very challenging to be optimized on the GPU architecture. In contrast, inheritance occurs *only* when some threads in a warp experience invalid samples. This strategy can be efficiently optimized on the GPU architecture with warp-level intrinsics and static memory management.

4.2 Warp Streaming

Within each sample iteration, Refine scans the candidate array *cand* and outputs a *refine* array for Sample. Both arrays can have varying lengths for different samples, which causes the *refine imbalance*. To overcome the issue, we dynamically distribute the Refine and Sample workloads among threads in the same warp in a streaming manner. Intuitively, if a sample contains enough workloads to be parallelized by a warp, we keep the threads busy by streaming the Refine and Sample operations to the warp.

The warp streaming approach is presented in Algorithm 3. We initialize four variables to keep track of the streaming process for each sample iteration. *curIter* keeps track of the next vertex to process in the *cand* array. *curV* and *curW* denote the current vertex sampled and its corresponding sample weight. *curTotal* is the total sample weight accumulated among all processed candidates. There are two phases in streaming the candidates: *Collaborative Phase* and *Independent Phase*.

Collaborative Phase (Line 5-17). In Line 5, _any checks if there exists a thread holding a sample that has enough candidates to be processed by the warp. If so, one of the qualifying thread takes the leader role with _ballot and share its sample s to all threads in the warp via _shfl (Line 6-7). The leader also shares its candidate array *cand* so that each thread is assigned to process a unique candidate in *cand* (Line 8). The candidate is later fed to Refine followed by Sample. Now each thread holds a sampled vertex v with its sample weight w set, we need to select at most one vertex v^* from the warp and make sure its sampling probability is proportional to its sample weight w^*. To this end, we adapt the A-Res streaming algorithm [11] where a key is randomly generated as $r^{1/w}$ for any vertex v and weight w where r is a uniform random number in $(0, 1)$. Subsequently, _reduce_max safely selects the vertex v^* and its weight w^* with the largest key value. Furthermore, _reduce_sum computes the total weight *totalW* among the warp. The leader thread can then update *curV* and *curW* with v^* and w^*. Line 15-16 guarantees the selected vertex sample *curV* always has a sampling probability proportional to *curW* among processed candidates with total sampling weight *curTotalW*. The leader also moves *curIter* to indict 32 candidates that have been processed.

Independent Phase (Line 18-22). When no thread holds more than 32 candidates to process, each thread streams the remaining candidates independently. A thread incrementally refines a candidate and samples a vertex v with weight w. Line 21-22 uses the same approach as Line 15-16 to ensure *curV* has the correct sampling probability wrt. *curW*.

Finally, we check if $s \cup \{curV\}$ form a valid partial/full instance. The following theorem guarantees that *curV* is selected with the correct probability distribution.

Algorithm 3: Warp Streaming

```
   // Replace Line 9-11 in Algorithm 1
1  curIter ← 0;
2  curV ← dummy;
3  curW ← 0;
4  curTotalW ← 0;
5  while _any(clen - curIter ≥ 32) do
6      leaderId ← _ballot(clen - curIter ≥ 32);
7      leaderSample ← _shfl(s, leaderId);
8      workerCand ← _shfl(cand,leaderId) + threadId;
9      rlen ← Refine(leaderSample,d,workerCand,1,refine);
10     (v,w) ← Sample(leaderSample,d,refine,rlen);
11     key ← UniformRand()^{1/w} if w ≠ 0 and 0 otherwise;
12     (key*,v*,w*) = _reduce_max(tuple<>(key,v,w));
13     totalW ← _reduce_sum(w);
14     if threadId == leaderId then
15         curTotalW ← curTotalW + totalW;
16         (curV,curW) ← (v*,w*) with probability $\frac{totalW}{curTotalW}$
17         curIter = curIter + 32;
18 while curIter < clen do
19     rlen ← Refine(cg,s,d,cand+curIter,1,refine);
20     (v,w) ← Sample(s,d,refine,rlen);
21     curTotalW ← curTotalW + w;
22     (curV,curW) ← (u,w) with probability $\frac{w}{curTotalW}$;
23 valid ← Validate(s,d,curV,$\frac{curW}{curTotalW}$);
```

Theorem 2. *curV is sampled with a probability* $\frac{curW}{curTotalW}$ *is an invariant in Algorithm 3.*

5 CPU-GPU CO-PROCESSING

While gSWORD demonstrates high efficiency in generating a large number of samples and providing accurate estimations for various query types, our empirical evaluation has revealed certain challenging scenarios where the estimation deviates significantly despite that millions of samples are generated. This discrepancy can be attributed to the underestimation problem encountered by RW estimators when the sample space exhibits significant skewness, resulting in a dearth of valid samples that match the query graph. Previous research studies have also reported cases of underestimation in their investigations [21, 52]. Furthermore, certain extreme cases, such as queries within the WordNet dataset, present an even greater hurdle, as many of them do not yield a single valid sample.

Trawling Strategy. A simple strategy to overcome the deficiency of valid samples is to directly enumerate all valid instances. However, the enumeration is simply prohibitive to handle large search spaces. In contrast, the sampling methods can estimate large search

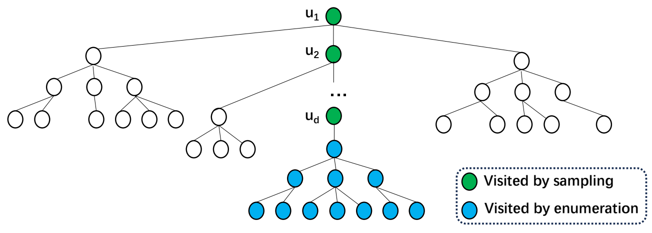


Figure 8: Example of trawling.

spaces but suffer from invalid samples. Hence, we combine the merits of sampling and enumeration and propose a *trawling* strategy to mitigate the issue of underestimation.

As shown in Figure 8, we sample d vertices as a partial instance of $s_i(d)$ where $d \le |\mathcal{V}_q|$ and then enumerate the remaining $|\mathcal{V}_q| - d$ vertices to obtain the number of valid instances given $s_i(d)$ as the partial instance. As such, the sampled partial instance can quickly navigate through the large sample space. By effectively reducing the sample space, the subsequent enumeration process focuses on collecting as many valid instances as possible, a process we refer to as *trawling*. Algorithm 4 represents the process of trawling strategy. We select d to determine the number of nodes to sample. The estimate H_s of sample s with d vertices is obtained using the HT estimator in Algorithm 1. Then we invoke the enumeration algorithm to count cnt, which is the total number of valid instances that can be extended from s. Line 6 updates the estimator H using cnt and H_s. Note that all enumeration processes on CPUs will be executed after all samples on GPUs are completed.

Algorithm 4: The Trawling Strategy.

```
  Input  :Candidate graph $cg$, block sample pool $bp$
  Output :Estimate $H$, sample size $nSample$
1 nSample, $H \leftarrow 0$;
2 while $bp \neq \emptyset$ do
3     $d \leftarrow$ Select($|\mathcal{V}_q|$);
4     Get the estimate $H_s$ by sampling a partial instance $s$
        with $d$ vertices using Algorithm 1;
5     cnt ← Enumeration(cg,s);
6     $H \leftarrow H + H_s \cdot$ cnt;
7     nSample ← nSample + 1;
8 return ($H$, $nSample$);
```

Selection of d. Selecting an appropriate value for d strikes a balance between efficiency and accuracy. When $d = 0$, it corresponds to a purely enumeration-based approach that yields the exact answer but becomes computationally impractical for large sample spaces. Conversely, as d increases, the efficiency improves due to the utilization of a sampled partial instance. However, a higher d also increases the likelihood of including more invalid samples, resembling a purely sampling-based method when $d = |\mathcal{V}_q|$.

Motivated by the need for this trade-off, we devise a random selection process based on a geometric distribution. Specifically, we set $\mathbb{P}(d = j) \propto 2^{-j}$, where j ranges from 3 to $|\mathcal{V}_q|$. We initiate the enumeration process only from the third vertex onwards to avoid the prohibitively high cost associated with full enumeration. In practice, the sampling method can generate a considerable number of valid partial instances with three vertices, while the enumeration process ensures a comprehensive coverage of the search space given the vertices that have been sampled.

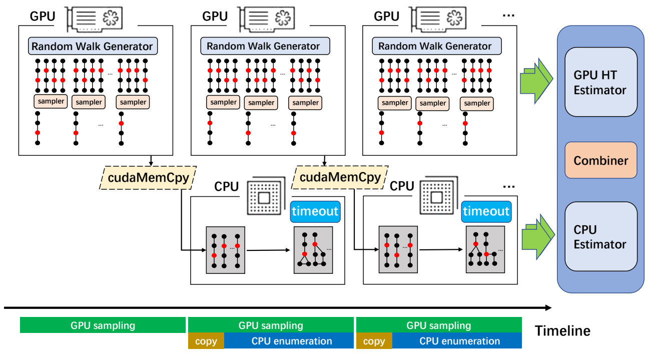


Figure 9: CPU-GPU co-processing pipeline.

CPU-GPU Co-processing. Deploying the trawling strategy solely on GPUs would not be efficient because the additional enumeration operations can occupy GPU computing resources and degrade the existing sampling process.

Moreover, recent advancements in complex pruning techniques for subgraph matching are not suitable for GPUs' SIMT architecture. CPUs, on the other hand, excel at handling such intricate logic, making them an ideal choice for enumeration using state-of-the-art CPU-based methods [3–6, 20, 36].

To address this issue, we propose a CPU-GPU co-processing pipeline, as depicted in Figure 9. Sampling batches are scheduled and the GPU generates complete samples for each batch using previously discussed techniques. This ensures continuous production of sampling estimates, regardless of whether the trawling strategy is activated. Additionally, we uniformly select t samples and transfer the selected samples to CPUs for trawling such that the CPU-GPU data transfer overhead is limited by $O(t|\mathcal{V}_q|)$. For now, we set t to be the number of cores on the GPUs. On the CPU side, we produce a separate estimate by trawling the received samples. We use the CPU method from [36] but any alternative can be easily substituted.

The CPU-GPU co-processing pipeline incurs minimal overhead compared to a GPU-only implementation of gSWORD. Figure 9 illustrates the concurrent execution of CPU and GPU processing, which maximizes resource utilization of both sides. To prevent stalls caused by expensive and unpredictable CPU-side enumeration overhead, a timeout mechanism is employed. Once the GPU sampling batch completes, CPU-side enumeration is terminated, and only samples that have completed the enumeration process are considered for estimation. This parallel batch execution allows for the full exploitation of the distinctive processing characteristics of CPUs and GPUs. For better overlapping the CPU enumeration and GPU sampling, a small batch size is preferred. However, if the batch size is too small, it may lead to poor estimates due to insufficient enumeration. We conducted an experiment to assess the trade-off between batch size and estimation accuracy in Section 6.5.

In the Appendix of our extended report [1], we show that the proposed trawling strategy produces an unbiased estimation to the subgraph count under the CPU-GPU co-processing pipeline.

Table 1: Dataset Statistics.

Categories	Dataset	$\mathcal{V}$	$\mathcal{E}$	d	$\mathcal{L}$
Biology	Yeast	3,112	12,519	8.0	71
	HPRD	9,460	34,998	7.4	307
Lexical	WordNet	76,853	120,399	3.1	5
Citation	Patents	3,774,768	16,518,947	8.8	20
	DBLP	317,080	1,049,866	6.6	15
Social	Orkut	3,072,441	117,185,083	38.14	150
Web	eu2005	862,664	16,138,468	37.4	40
	uk2002	18,520,486	298,113,762	16.1	200

6 EXPERIMENTAL EVALUATION

6.1 Experimental Setup

Compared Methods. We compare gSWORD with G-Care [28], the-state-of-the-art CPU-based sampling framework for subgraph counting, and *NextDoor* [18], the state-of-the-art GPU-based sampling framework for conventional RW workloads. Specifically, we regard each sample as a task unit and keep load balance among threads with the *dynamic scheduling* [30]. It achieves high performance on CPUs because RW estimators are embarrassingly parallel. We study two sampling algorithms including WanderJoin (WJ) and Alley (AL). Alley cannot be directly deployed to*NextDoor* because the framework does not consider the refinement step in Alley. Therefore, we implement Alley without any frameworks but following the same parallelization strategy of *NextDoor* as the baseline on GPUs. In summary, we compare the following methods.

- CPU-WJ and CPU-AL are the baselines on CPUs, implemented within G-Care [28].
- GPU-WJ and GPU-AL are the baseline on GPUs, implemented by following the computation method of *NextDoor* [18].
- gSWORD-WJ and gSWORD-AL are the implementations of gSWORD.

Data graphs. We conduct experiments on eight real-world datasets from diverse categories. These datasets are widely used in previous studies [5, 16, 36, 38, 53]. Table 1 presents the statistics of the datasets. Three of them are labeled graphs, namely, Yeast, HPRD, and WordNet. The rest are unlabeled graphs. For unlabeled graphs, we follow the same method used in existing works [6, 16] to randomly generate labels for the vertices.

Queries. To align with previous research [4–6, 36], the queries are extracted from the data graphs using random walks. The number of vertices in queries are set to 4, 8, 16 with 16 as the default. We generate 20 queries for the same number of query vertices on each data graph. For queries with 8 or 16 vertices, we generate 10 sparse queries and 10 dense queries where a sparse query has the maximum degree less than 3. For all compared methods, we generate 10^6 samples for each query by default as the RW estimators can converge for most datasets as discussed in Section 6.4.

In our efficiency evaluation, we do not take into account the cost associated with constructing and transferring the candidate graph from CPUs to GPUs. We compare the current implementations to those that directly sample the data graph *without* using candidate graphs in the Appendix of the extend report [1]. Our results indicate the running time of sampling directly on data graph is consistently higher than the approach of sampling on the candidate graph, even when accounting for the preparation costs associated with constructing and transferring candidate graphs.

Matching order. We adopt the matching order from QuickSI [33] as our default matching order, which achieves a good performance based on the performance study [37]. Both Alley and WanderJoin follow the same matching order. We also compare QuickSI and the matching order in G-CARE and the results of the two orders in terms of efficiency and accuracy are similar. Please refer to the Appendix of the extended report [1] for the experimental results.

Environment. All experiments are conducted on a server equipped with Intel Xeon W-2133 CPUs (12 cores,3.6GHz), 64 GB main memory, and two RTX 2080 Ti GPUs. All source codes are implemented in CUDA/C++ and compiled by O3 optimizations.

6.2 Main Efficiency Results

Overall Comparison. The average running time of all the compared approaches on processing a query under the default setting is presented in Table 2. A detailed analysis of the errors in the sampling algorithms will be discussed in Section 6.4 later on.

Among the CPU-based methods, CPU-AL is slower than CPU-WJ due to the overhead of the refinement operations. For example, CPU-WJ is 2.7 times faster than CPU-AL on eu2005. Nonetheless, the running time of all CPU-based methods ranges from 3-223 seconds for processing one query. This result demonstrates the necessity of accelerating RW estimators on GPUs.

The GPU-based methods produce massive efficiency boost. On average, the GPU baselines achieve 90.0x and 21.3x speedup over the CPU counterparts for WanderJoin and Alley, respectively. The speedup of GPU-AL is lower compared with those of GPU-WJ because the refinement of GPU-AL incurs heavy memory access cost. In general, the running time of all GPU baselines falls within the range of 0.2-9.7 seconds. gSWORD exhibits superior performance by completing the queries within 0.7 seconds. On average, gSWORD achieves a speedup of 341x compared to the CPU baselines and is 9x faster than the GPU baselines. These results clearly demonstrate the remarkable efficiency of gSWORD as proposed in this paper.

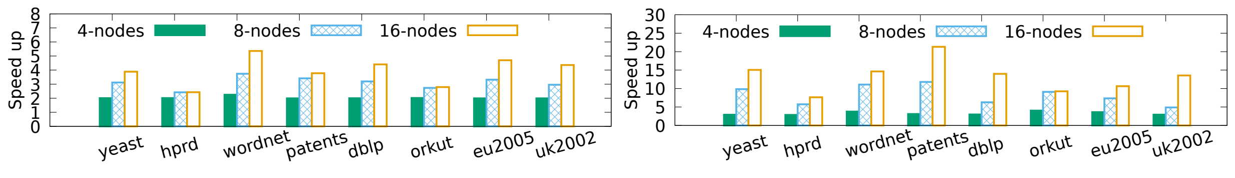
Given the significant performance advantage of GPU-based methods over CPU-based methods, our focus in the subsequent experiments is on the GPU-based approaches. Consequently, we proceed to evaluate the performance of gSWORD while varying the query size and query type.

Varying Query Size & Query Type. In Figure 10, we observe the speedup achieved by gSWORD over the GPU baselines as the query size increases from 4 to 16. Notably, Alley exhibits a significantly higher speedup compared to WanderJoin. This is attributed to the fact that WanderJoin does not face the issue of workload imbalance during refinement, unlike Alley. As the query size increases, gSWORD achieves a higher speedup in both methods, particularly in Alley. This is primarily due to the GPU baselines encountering significant workload imbalances during validation and refinement operations for larger queries. The increased number of iterations and overhead associated with refining larger queries contribute to this observation. The results clearly demonstrate the effectiveness of the optimization techniques proposed in Section 4.

Figure 11 depicts the speedup achieved by gSWORD on dense and sparse queries. gSWORD performs well on queries with varying structures, highlighting the robustness of the framework.

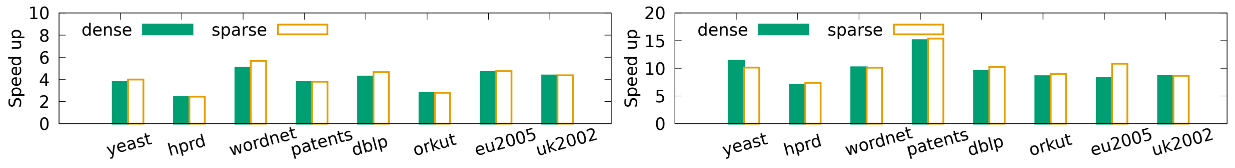
Table 2: Average running time (milliseconds) per query of the compared approaches with standard deviations (±).

Methods	Yeast	HPRD	WordNet	Patents	DBLP	Orkut	eu2005	uk2002
CPU-WJ	2929±457	15396±599	6744±1127	82788±3907	22843±3329	79056±43199	46228±10055	99296±7790
CPU-AL	4571±792	16711±690	7278±1204	132309±5696	38219±4016	171921±17796	124693±35402	223441±33915
GPU-WJ	237±55	260±29	246±55	302±14	348±96	569±310	456±97	1160 ±91
GPU-AL	2697±233	2768±95	1026±169	2281±236	2880±302	5912±601	3822±1085	9614±1459
gSWORD-WJ	61±10	107±8	46±8	80±3	79±11	204±51	97±20	266±20
gSWORD-AL	179±14	361±12	70±11	107±11	205±22	639±70	359±103	709±107



(a) WanderJoin (b) Alley

Figure 10: The speedup of gSWORD over GPU baselines with the query size increasing.



(a) WanderJoin (b) Alley

Figure 11: The speedup of gSWORD over GPU baselines with the query type varied.

6.3 Evaluation of GPU-Centric Optimizations

To evaluate the impact of each optimization proposed in Section 4, we conducted an ablation study on `WanderJoin` and `Alley`, specifically focusing on runtime reduction for the proposed optimizations.

We measure the runtime while enabling sample inheritance and warp streaming optimizations incrementally, and the results are depicted in Figure 12. When only the inheritance optimization was enabled, a significant reduction in runtime was observed for both `WanderJoin` and `Alley`, resulting in speedups of 3.9x and 2.5x, respectively. The higher speedup in `WanderJoin` can be attributed to its heavier validate workload imbalance compared to `Alley`. Furthermore, upon enabling the warp streaming optimization, a 5.3x further reduction in runtime was achieved specifically in `Alley`. However, the runtime for `WanderJoin` did not experience further reduction as it does not have a refine stage for sampling. The results confirm the effectiveness of the optimizations in addressing the imbalance issues for enabling RW estimators on GPUs.

6.4 Evaluation of RW estimators

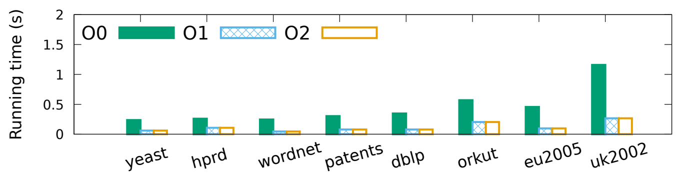
We measure the accuracy of RW estimators with *q-error*, a widely used metric for the cardinality estimation problem [25]. Suppose that the estimated cardinality is $\hat{c}$ and the ground truth is c. Q-error is equal to $max(max(1, c)/max(1, \hat{c}), max(1, \hat{c})/max(1, c))$. The value of the q-error represents the quality of estimation, where a smaller value indicates a better estimation. To present the q-error of overestimated and underestimated queries, we display the q-error of overestimated queries *upward* and the q-error of underestimated queries *downward* with respect to a reference value of 1.

Figure 13 illustrates the q-errors of the two RW estimators for different query sizes. Comparatively, `Alley` consistently outperforms `WanderJoin` when employing the same number of samples. For 4-node queries, both methods yield accurate estimations. As the number of query vertices increases to 8, `Alley` maintains accurate estimates, while `WanderJoin` exhibits larger but still reasonable errors. However, as the number of query vertices further increases to 16, we observe a significant deterioration in the accuracy of `WanderJoin`. On the other hand, `Alley` maintains stable accuracy across different query sizes, except in the case of the `WordNet` dataset where a notable increase in q-error is observed for both estimators. Specifically, the maximum q-error for 16-nodes queries is 10^6 for `Alley` but 10^9 for `WanderJoin`. It is important to note that both `Alley` and `WanderJoin` suffer from severe underestimations in their estimations.

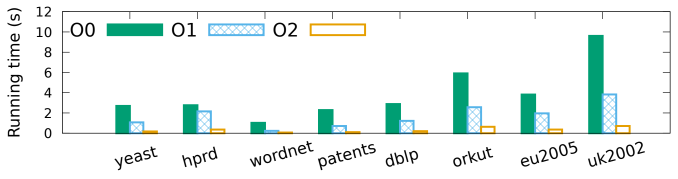
In order to investigate the severe underestimation problem observed in `WordNet`, we conducted an experiment to analyze the percentages of valid samples for `Alley` across all datasets. The results are presented in Figure 14. The chance of obtaining a valid sample for 16-node queries was found to be extremely low, falling below 10^{-5}%. This implies that out of 10^6 samples, there were less than one valid sample obtained on average. The significantly low percentages of valid samples in the `WordNet` dataset render the RW estimators unreliable for such cases.

6.5 Evaluation of CPU-GPU Co-processing

To showcase the effectiveness of our CPU-GPU co-processing solution in mitigating the underestimation problem, we provide a comprehensive analysis of the q-error and runtime for all 20 queries

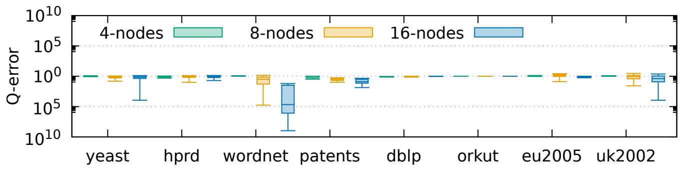


(a) gSWORD-WJ

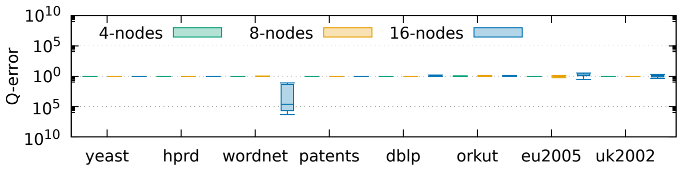


(b) gSWORD-AL

Figure 12: Runtime of the proposed methods with no optimization (O0), sample inheritance optimization only (O1) and sample inheritance+warp streaming optimizations (O2).



(a) WanderJoin



(b) Alley

Figure 13: Q-error of the RW estimators under study.

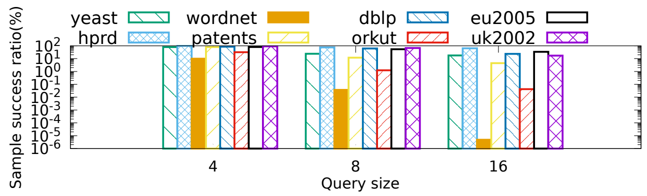


Figure 14: Sample success ratio of ALLEY

with 16 vertices in the WordNet dataset, which exhibit the most severe underestimation issue.

Q-error Reduction. Figure 15 demonstrates the impact of employing trawling on the q-errors in both WanderJoin and Alley. Utilizing the trawling technique significantly reduces the q-errors by a considerable factor of $5.7 \cdot 10^5$ in WanderJoin and $1.7 \cdot 10^5$ in Alley. Furthermore, the maximum q-error in WanderJoin, across all queries, is reduced from 10^9 to $1.2 \cdot 10^4$, while the maximum q-error in Alley is reduced from $2 \cdot 10^6$ to $1.2 \cdot 10^4$ through trawling. These improvements underscore the effectiveness of the trawling strategy in mitigating underestimation issues.

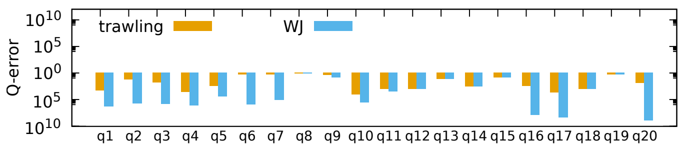
Runtime Overhead. In addition, we conducted an evaluation to assess the overhead of CPU-GPU co-processing by measuring the individual components of GPU sampling and CPU enumeration. The results were then compared with the total execution time of CPU-GPU co-processing, as shown in Figure 16. Notably, the execution times are very similar, indicating that the overhead of CPU enumeration is effectively overlapped with GPU sampling. This demonstrates the efficient coordination and synchronization between CPU and GPU in the co-processing solution.

Tuning The Number Of Batches. Figure 17 illustrates the impact of the number of batches on CPU-GPU co-processing performance using five representative queries. Increasing the number of batches ideally reduces the q-error by facilitating more overlaps between CPU enumeration and GPU sampling. This enables a greater number of samples to be enumerated on the CPU side, improving estimation accuracy and reducing q-error. However, we observed an increase in q-error specifically for queries $q2$ and $q3$ when the number of batches reached 8, while the overall execution time remained stable across different batch sizes. This increase in q-error is attributed to limited execution time available for the enumeration process within each batch when the number of batches is too large. Insufficient time allocated for accurate enumeration leads to poorer estimation results. To strike a balance between sampling and enumeration overlap and accurate estimation, we determined that 6 is the default setting for the number of batches in our experiment.

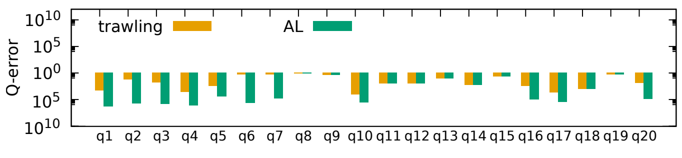
Enumeration With Multiple Threads. We conduct an experiment by varying the number of CPU threads to demonstrate the adaptability across systems with differing CPU and GPU computational power. We find that increasing the number of threads is associated with a decrease in Q-error for representative queries. The results are presented in Figure 18. Take query $q3$ for an example, the Q-error is 300 when deploying a single thread, but it reduces to 64 after deploying 12 threads. This is because CPU enumeration is much more expensive than GPU sampling. In parallel execution for a batch, when the GPU finishes the sampling process, it prompts the CPU to terminate. As a result, additional CPU resources can complete more enumeration tasks, enhancing accuracy without extending the overall runtime.

7 RELATED WORK

Random walk (RW) is an effective approach to extract information from large graphs. For example, many RW algorithms are proposed to learn graph representations such as DeepWalk [29] and Node2vec [14]. Many surveys [10, 32, 44, 45] on RW-based algorithms have been published. To accelerate RW algorithms, recently researchers have proposed a variety of frameworks to implement different RW algorithms. These frameworks consider each random walk as a parallel task unit and focus on accelerating the sampling procedure that chooses a vertex from the neighbors of the current residing vertex given the probability distribution customized by users. In the following, we will review the CPU and GPU based systems, followed by learning-based subgraph counting.



(a) WanderJoin



(b) Alley

Figure 15: Q-error of RW estimators v.s. trawling in WordNet with 16-nodes queries.

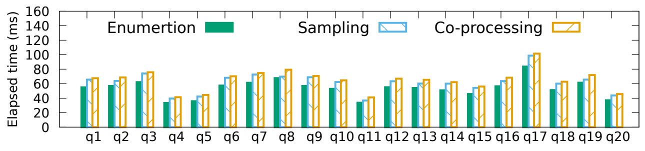


Figure 16: The component time in CPU-GPU co-processing.

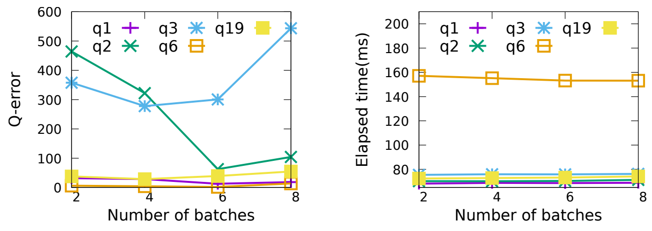


Figure 17: Q-error and runtime with varying batches.

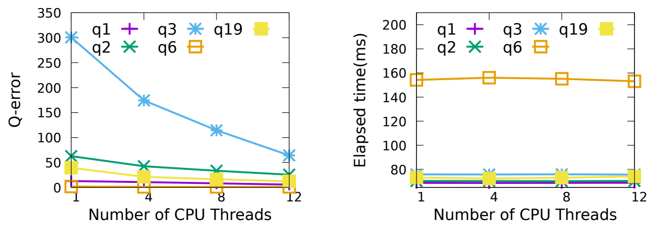


Figure 18: Q-error and runtime with varying threads.

CPU-based Systems. KnightKing [47] is a distributed framework adopting the BSP (bulk synchronous parallel) model. In particular, it expands one vertex for all queries in each iteration. To reduce the communication cost across machines, it uses the *rejection sampling* method to avoid scanning the whole neighbor set to sample one vertex. GraphWalker [42] is an I/O efficient framework on a single machine. It adopts the asynchronous model where each thread executes a walk independently. It divides the graph that cannot reside in the memory into a set of partitions and optimizes the scheduling method of loading partitions into memory to minimize the disk I/Os. ThunderRW [35] focuses on improving in-memory computation efficiency for random walks. Specifically, it proposes the step interleaving technique that executes different queries in an interleaving manner to reduce CPU pipeline stalls incurred by random memory accesses. Uninet [48] proposes a sampler called M-H to generate a sample vertex in constant time.

GPU-based Systems. To further improve the performance, several GPU-based systems are proposed. C-SAW [27] adopts the BSP model that executes one step for all queries in each iteration. It implements the *inverse transformation sampling* method to sample a vertex, which needs to scan the neighbors of the current residing vertex. To improve the GPU computation efficiency, C-SAW supports RW with the same length only to enforce the regular workload. NextDoor [18] is a framework targeting at graph sampling applications. It adopts the *rejection sampling* method. Users can implement a wide variety of RW-algorithms by defining a "next" function, which specifies the logic of sampling the next vertex. SkyWalker [41] improves the performance on large graphs by optimizing the *alias sampling* method on GPUs. In particular, it assigns computation resources to build the alias table based on the vertex degrees to achieve balanced workload.

Nevertheless, all existing CPU and GPU systems focus on the optimization for the sampling stage in traditional random walks. However, as discussed in Section 1, the heavy refinement step in RW estimators for subgraph counting incurs the validate workload imbalance and refine workload imbalance problems, which lead to the under utilization of GPUs' resources. Different from all previous works, we focus on designing an efficient framework to accelerate RW estimators for subgraph counting.

Learning-based Subgraph Counting. Recently, researchers propose several learning-based subgraph counting methods [40, 49, 52]. The active learned sketch method [52] estimates subgraph count by creating a sketch with a neural network regression model and updating the sketch with an active learner given new arrival queries. NeurSC [40] extracts representative substructures from the data graph for each query and then estimates subgraph count based on learned representations. LSSMatch [49] improves the estimation for complex cyclic graph queries by the query decomposition based on learning techniques. However, the error of these methods cannot be bounded due to the neural network components and hence the learned methods incur expensive training and maintenance costs. Additionally, these methods focus on algorithmic design, whereas we aim to build an efficient framework for RW estimators.

8 CONCLUSION

We present gSWORD, a GPU framework that accelerates RW estimators for subgraph counting. There are unique challenges to accelerate the workflow on GPUs due to severe workload imbalance. We propose sample inheritance for refine imbalance and warp streaming for validate imbalance. Moreover, existing sampling-based solutions suffer from underestimation issues due to the lack of valid samples. We propose a CPU-GPU co-processing pipeline which overlaps the sampling and enumeration processes to mitigate this issue. Through extensive experiments, we verify the efficacy of gSWORD over the state-of-the-art CPU and GPU baselines. Ablation studies further validate the effectiveness of our optimization strategies. Moreover, we verify that our co-processing pipeline enhances estimation accuracy, particularly in scenarios where obtaining valid samples is challenging.

REFERENCES

[1] 2023. The technical report for gsword. https://anonymous.4open.science/r/gsword-FEF6/report/report.pdf.

[2] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems(TODS)* 42, 4 (2017), 1–44.

[3] Junya Arai, Yasuhiro Fujiwara, and Makoto Onizuka. 2023. GuP: Fast Subgraph Matching by Guard-based Pruning. *PACMMOD* 1, 2 (2023), 1–26.

[4] Blair Archibald, Fraser Dunlop, Ruth Hoffmann, Ciaran McCreesh, Patrick Prosser, and James Trimble. 2019. Sequential and parallel solution-biased search for subgraph algorithms. In *CPAIOR*. 20–38.

[5] Bibek Bhattarai, Hang Liu, and H Howie Huang. 2019. Ceci: Compact embedding cluster index for scalable subgraph matching. In *SIGMOD*. 1447–1462.

[6] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *SIGMOD*. 1199–1214.

[7] Ian Buck. 2007. Gpu computing with nvidia cuda. In *ACM SIGGRAPH 2007 courses*. 6–es.

[8] Xiaowei Chen, Yongkun Li, Pinghui Wang, and John CS Lui. 2016. A general framework for estimating graphlet statistics via random walk. *PVLDB* 10, 3 (2016), 253–264.

[9] Stephen A Cook. 1971. The complexity of theorem-proving procedures. In *STOC*. 151–158.

[10] Peng Cui, Xiao Wang, Jian Pei, and Wenwu Zhu. 2018. A survey on network embedding. *IEEE Transactions on Knowledge and Data Engineering(TKDE)* 31, 5 (2018), 833–852.

[11] Rayane El Sibai, Yousra Chabchoub, Jacques Demerjian, Zakia Kazi-Aoul, and Kablan Barbar. 2016. Sampling algorithms in data stream environments. In *ICDEc*. 29–36.

[12] Jessica Enright and Rowland Raymond Kao. 2018. Epidemics on dynamic networks. *Epidemics* 24 (2018), 88–97.

[13] Philippe Fournier-Viger, Ganghuan He, Chao Cheng, Jiaxuan Li, Min Zhou, Jerry Chun-Wei Lin, and Unil Yun. 2020. A survey of pattern mining in dynamic graphs. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery(DMKD)* 10, 6 (2020), e1372.

[14] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *KDD*. 855–864.

[15] Wentian Guo, Yuchen Li, and Kian-Lee Tan. 2020. Exploiting reuse for GPU subgraph enumeration. *IEEE Transactions on Knowledge and Data Engineering(TKDE)* 34, 9 (2020), 4231–4244.

[16] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *SIGMOD*. 1429–1446.

[17] Zaïd Harchaoui and Francis Bach. 2007. Image classification with segmentation graph kernels. In *CVPR*. 1–8.

[18] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. 2021. Accelerating graph sampling for graph machine learning using GPUs. In *EuroSys*. 311–326.

[19] Rajesh Jayaram, Gokarna Sharma, Srikanta Tirthapura, and David P Woodruff. 2019. Weighted reservoir sampling from distributed streams. In *PODS*. 218–235.

[20] Xun Jian, Zhiyuan Li, and Lei Chen. 2023. SUFF: Accelerating Subgraph Matching with Historical Data. *PVLDB* 16, 7 (2023), 1699–1711.

[21] Kyoungmin Kim, Hyeonji Kim, George Fletcher, and Wook-Shin Han. 2021. Combining Sampling and Synopses with Worst-Case Optimal Runtime and Quality Guarantees for Graph Pattern Cardinality Estimation. In *SIGMOD*. 964–976.

[22] Zhuohang Lai, Xibo Sun, Qiong Luo, and Xiaolong Xie. 2022. Accelerating multi-way joins on the GPU. *VLDB J.* 31, 3 (2022), 529–553.

[23] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander join: Online aggregation via random walks. In *SIGMOD*. 615–629.

[24] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2019. Wander join and XDB: online aggregation via random walks. *ACM Transactions on Database Systems(TODS)* 44, 1 (2019), 1–41.

[25] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing bad plans by bounding the impact of cardinality estimation errors. *PVLDB* 2, 1 (2009), 982–993.

[26] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming irregular graphs for gpu-friendly graph processing. *ACM SIGPLAN Notices* 53, 2 (2018), 622–636.

[27] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S Li, and Hang Liu. 2020. C-SAW: A framework for graph sampling and random walk on GPUs. In *SC*. 1–15.

[28] Yeonsu Park, Seongyun Ko, Sourav S Bhowmick, Kyoungmin Kim, Kijae Hong, and Wook-Shin Han. 2020. G-CARE: a framework for performance benchmarking of cardinality estimation techniques for subgraph matching. In *SIGMOD*. 1099–1114.

[29] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *KDD*. 701–710.

[30] Imran Qureshi. 2014. Cpu scheduling algorithms: A survey. *International Journal of Advanced Networking and Applications(IJANA)* 5, 4 (2014), 1968.

[31] Pedro Ribeiro, Pedro Paredes, Miguel EP Silva, David Aparicio, and Fernando Silva. 2021. A survey on subgraph counting: concepts, algorithms, and applications to network motifs and graphlets. *ACM Computing Surveys(CSUR)* 54, 2 (2021), 1–36.

[32] Purnamrita Sarkar and Andrew W Moore. 2011. Random walks in social networks and their applications: a survey. In *Social Network Data Analytics*. 43–77.

[33] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB* 1, 1 (2008), 364–375.

[34] Nino Shervashidze, SVN Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten Borgwardt. 2009. Efficient graphlet kernels for large graph comparison. In *AISTATS*. 488–495.

[35] Shixuan Sun, Yuhang Chen, Shengliang Lu, Bingsheng He, and Yuchen Li. 2021. ThunderRW: An in-memory graph random walk engine. *PVLDB* 14, 11 (2021), 1992–2005.

[36] Shixuan Sun and Qiong Luo. 2020. In-memory subgraph matching: An in-depth study. In *SIGMOD*. 1083–1098.

[37] Xibo Sun and Qiong Luo. 2023. Efficient GPU-Accelerated Subgraph Matching. *PACMMOD* 1, 2 (2023), 1–26.

[38] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient subgraph matching on billion node graphs. *PVLDB* 5, 9 (2012), 788–799.

[39] Vladimir Vacic, Lilia M Iakoucheva, Stefano Lonardi, and Predrag Radivojac. 2010. Graphlet kernels for prediction of functional residues in protein structures. *Journal of Computational Biology(J. Comput. Biol)* 17, 1 (2010), 55–72.

[40] Hanchen Wang, Rong Hu, Ying Zhang, Lu Qin, Wei Wang, and Wenjie Zhang. 2022. Neural Subgraph Counting with Wasserstein Estimator. In *SIGMOD*. 160–175.

[41] Pengyu Wang, Chao Li, Jing Wang, Taolei Wang, Lu Zhang, Jingwen Leng, Quan Chen, and Minyi Guo. 2021. Skywalker: Efficient Alias-Method-Based Graph Sampling and Random Walk on GPUs. In *PACT*. 304–317.

[42] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John CS Lui. 2020. GraphWalker: An I/O-Efficient and Resource-Friendly Graph Analytic System for Fast and Scalable Random Walks. In *USENIX ATC*. 559–571.

[43] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *PPoPP*. 1–12.

[44] Feng Xia, Jiaying Liu, Hansong Nie, Yonghao Fu, Liangtian Wan, and Xiangjie Kong. 2019. Random walks: A review of algorithms and applications. *IEEE Transactions on Emerging Topics in Computational Intelligence(TETCI)* 4, 2 (2019), 95–107.

[45] Feng Xia, Ke Sun, Shuo Yu, Abdul Aziz, Liangtian Wan, Shirui Pan, and Huan Liu. 2021. Graph learning: A survey. *IEEE Transactions on Artificial Intelligence(TAI)* 2, 2 (2021), 109–127.

[46] Lizhi Xiang, Arif Khan, Edoardo Serra, Mahantesh Halappanavar, and Aravind Sukumaran-Rajam. 2021. cuTS: scaling subgraph isomorphism on distributed multi-GPU systems using trie based data structure. In *SC*. 1–14.

[47] Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. 2019. Knightking: a fast distributed graph random walk engine. In *SOSP*. 524–537.

[48] Xingyu Yao, Yingxia Shao, Bin Cui, and Lei Chen. 2021. Uninet: Scalable network representation learning with metropolis-hastings sampling. In *ICDE*. 516–527.

[49] Hao Zhang, Qiyan Li, Kangfei Zhao, Jeffrey Xu Yu, and Yuanyuan Zhu. 2022. How Learning Can Help Complex Cyclic Join Decomposition. In *ICDE*. IEEE, 3138–3141.

[50] Luming Zhang, Mingli Song, Zicheng Liu, Xiao Liu, Jiajun Bu, and Chun Chen. 2013. Probabilistic graphlet cut: Exploiting spatial structure cue for weakly supervised image segmentation. In *CVPR*. 1908–1915.

[51] Yu Zhang, Yuxuan Liang, Jin Zhao, Fubing Mao, Lin Gu, Xiaofei Liao, Hai Jin, Haikun Liu, Song Guo, Yangqing Zeng, et al. 2022. Egraph: efficient concurrent GPU-based dynamic graph processing. *IEEE Transactions on Knowledge and Data Engineering(TKDE)* 35, 6 (2022), 5823–5836.

[52] Kangfei Zhao, Jeffrey Xu Yu, Hao Zhang, Qiyan Li, and Yu Rong. 2021. A learned sketch for subgraph counting. In *SIGMOD*. 2142–2155.

[53] Peixiang Zhao and Jiawei Han. 2010. On graph query optimization in large networks. *PVLDB* 3, 1-2 (2010), 340–351.

[54] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random sampling over joins revisited. In *SIGMOD*. 1525–1539.

APPENDIX

Implementations of RW estimators in gSWORD

We present the implementation detail of `Alley` and `WanderJoin` in Figure 19. Both of them have three components which is Refine, Sample, and Validate. The `WanderJoin` has a light refine stage which designates the global candidate set as the output candidate arrays. In contrast, `Alley` incurs a significant overhead during its refinement, as it scans every candidate for refinement, and each refinement step involves intersecting multiple candidate sets. During the sampling stage, both methods randomly select a vertex from the input array and update the current sample probability. In the validate stage, `WanderJoin` assesses the validity of the current sample, whereas `Alley` directly produces the final estimate.

The design of RSV stages provides users with greater flexibility in allocating different workloads between the refine and validate stages. This means that, in response to the specific demands of applications, users can create their custom RW estimators by adjusting the number of elements to be refined, effectively balancing the trade-off between efficiency and accuracy.

Proofs of Theorems

Theorem. *$\mathbb{E}[R_i(s)] = \mathbb{E}[H_i(s)]$ for any given partial instance s where $H_i(s)$ denote the HT estimator in Equation 1 to sample i vertices without inheritance and $i \in [1, |\mathcal{V}_q|]$.*

Proof. We use induction to prove the theorem. For the base case $i = 1$, we have $\mathbb{E}[R_1(s)] = \mathbb{E}[\frac{\mathbb{I}(s)}{\mathbb{P}(s)}] = \mathbb{E}[H_1(s)]$. Assume the equality holds for $i = d$. There are two cases in the inductive step for $i = d + 1$: (1) s is not inherited; (2) s is inherited. For case (1), $\mathbb{E}[R_{d+1}(s)] = \mathbb{E}[R_d(s \cup \{v\})] = \mathbb{E}[H_d(s \cup \{v\})] = \mathbb{E}[H_{d+1}(s)]$. For case (2), $\mathbb{E}[R_{d+1}(s)] = \mathbb{E}[\sum_t \frac{R_d(s\cup\{v_t\})}{n_d}] = \frac{1}{n_d}\sum_t \mathbb{E}[R_d(s \cup \{v_t\})] = \frac{1}{n_d}\sum_t \mathbb{E}[H_d(s\cup\{v_t\})] = \frac{1}{n_d}\sum_t \mathbb{E}[H_{d+1}(s)] = \mathbb{E}[H_{d+1}(s)]$. Thus, $\mathbb{E}[R_{d+1}(s)] = \mathbb{E}[H_{d+1}(s)]$ and the theorem is proved. □

Theorem. *$curV$ is sampled with a probability $\frac{curW}{curTotalW}$ is an invariant in Algorithm 3.*

Proof. In the collaborative phase, we can show that any v^* is sampled with a probability of $\frac{w^*}{totalW}$ by the result of [19]. Thus, the total probability of v^* replacing existing $curV = u$ in Line 16 is:

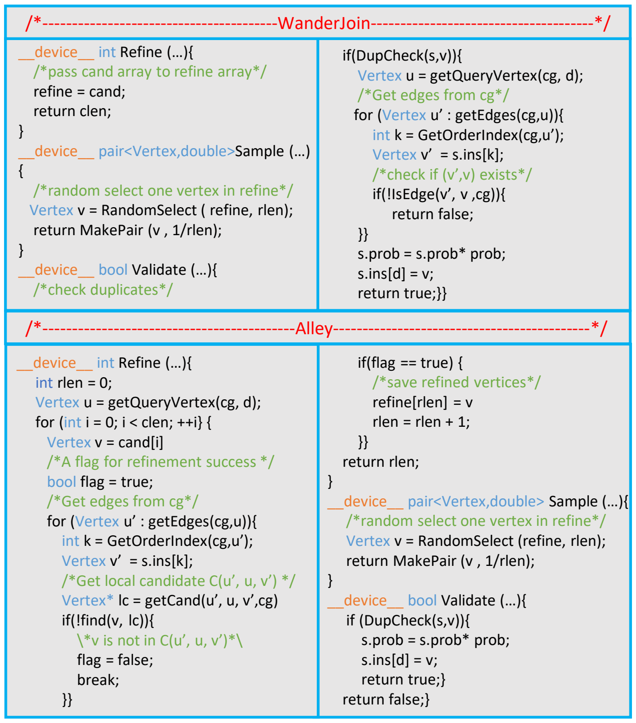
$$\frac{w^*}{totalW} \cdot \frac{totalW}{curTotalW} = \frac{w^*}{curTotalW} = \frac{curW}{curTotalW} \tag{2}$$

Further, u has a sampling probability of $\frac{curW}{curTotalW - totalW}$ and it remains to be $curV$ after seeing v^* is:

$$\frac{curW}{curTotalW - totalW} \cdot \frac{curTotalW - totalW}{curTotalW} = \frac{curW}{curTotalW} \tag{3}$$

In both cases, the invariant holds for the collaborative phase. The invariant for the sample update in Line 21-22 of the independent phase can be proved similarly. □

Next, we show that the proposed trawling strategy produces an unbiased estimation to the subgraph count under the CPU-GPU co-processing pipeline. Let $s_i(d)$ denote a partial instance sampled with d vertices and processed on CPUs for trawling. The estimation by $s_i(d)$ is computed as $T = \left(\prod_{j=1}^{d} |C_{ij}|\right) \mathbb{C}\,(s_i(d))$, where $\mathbb{C}\,(s_i(d))$ represents the number of valid instances given $s_i(d)$ as the partial instance and C_{ij} denote the candidate sets during the sample process for $s_i(d)$ on GPUs, the same as the notation used in the HT estimator H (Equation 1). We present the following theorem to show that the trawling strategy is unbiased.



```
/*--------------------------------------WanderJoin--------------------------------------*/
__device__ int Refine (...){                      if(DupCheck(s,v)){
  /*pass cand array to refine array*/               Vertex u = getQueryVertex(cg, d);
  refine = cand;                                    /*Get edges from cg*/
  return clen;                                      for (Vertex u' : getEdges(cg,u)){
}                                                     int k = GetOrderIndex(cg,u');
__device__ pair<Vertex,double>Sample (...)            Vertex v'  = s.ins[k];
{                                                     /*check if (v',v) exists*/
  /*random select one vertex in refine*/              if(!IsEdge(v', v ,cg)){
  Vertex v = RandomSelect ( refine, rlen);              return false;
  return MakePair (v , 1/rlen);                     }}
}                                                   s.prob = s.prob* prob;
__device__ bool Validate (...){                     s.ins[d] = v;
  /*check duplicates*/                              return true;}}
/*-----------------------------------------Alley-----------------------------------------*/
__device__ int Refine (...){                        if(flag == true) {
  int rlen = 0;                                       /*save refined vertices*/
  Vertex u = getQueryVertex(cg, d);                   refine[rlen] = v
  for (int i = 0; i < clen; ++i) {                    rlen = rlen + 1;
    Vertex v = cand[i]                              }}
    /*A flag for refinement success */              return rlen;
    bool flag = true;                             }
    /*Get edges from cg*/                         __device__ pair<Vertex,double> Sample (...){
    for (Vertex u' : getEdges(cg,u)){               /*random select one vertex in refine*/
      int k = GetOrderIndex(cg,u');                 Vertex v = RandomSelect (refine, rlen);
      Vertex v'  = s.ins[k];                        return MakePair (v , 1/rlen);
      /*Get local candidate C(u', u, v') */       }
      Vertex* lc = getCand(u', u, v',cg)          __device__ bool Validate (...){
      if(!find(v, lc)){                             if (DupCheck(s,v)){
        \*v is not in C(u', u, v')*\                  s.prob = s.prob* prob;
        flag = false;                                 s.ins[d] = v;
        break;                                        return true;}
    }}                                              return false;}
```

Figure 19: User-defined APIs in gSWORD.

Theorem. *T is unbiased as $\mathbb{E}[T] = \mathbb{C}$ where $\mathbb{C}$ is exact count.*

Proof. We have the following equation:

$$\mathbb{C} = \sum_d \mathbb{C} \cdot \mathbb{P}(d) = \sum_d \sum_i \mathbb{C}(s_i(d)) \cdot \mathbb{P}(d)$$
$$= \sum_d \sum_i \frac{\mathbb{C}(s_i(d))}{\mathbb{P}(s_i(d))} \cdot \mathbb{P}(d) \cdot \mathbb{P}(s_i(d)) = \mathbb{E}[T]$$

where $\mathbb{P}(d)$ denote the probability to select d vertices for trawling and $\mathbb{P}(s_i(d))$ denote the probability to sample $s_i(d)$ with d vertices. Note that the inner summation after the second equality is taken over all partial instances with d vertices. Finally, the last equality holds since $\mathbb{P}(s_i(d)) = \prod_{j=1}^{d} |C_{ij}|$. □

EVALUATION OF MATCHING ORDER

The matching order determines the sequence in which vertices of query graphs are matched, consequently shaping the sample space of RW estimators. Initially, `Alley` and `WanderJoin` would determine the best matching order in a round-robin fashion, evaluating each order using a heuristic and selecting the one with the smallest variance. However, for large queries, exploring every possible matching order results in a significant overhead. Instead of reusing samples from different orders for the final estimation, we set a maximum execution time of finding the best matching order to 10 minutes. Subsequently, we collect all 10^6 samples associated with the best order that was identified. We conducted an experiment to investigate the impact of the matching order of QuickSI and Alley on the performance of gSWORD.

Figure 22 illustrates the running time of gSWORD for 16-node queries with G-Care's and QuickSI's orders, respectively. Both two orders achieve a similar running time. On average, the running time with the order of QuickSI is 7.1% lower than that of G-Care and 6.5% for WanderJoin. For queries with 4 and 8 vertices, both also achieve a similar result presented in Figure 20 and 21.

In Figure 25, we present the Q-error results for 16-node queries in gSWORD using both matching orders. Across all datasets, Alley exhibits similar Q-errors with G-Care's and QuickSI's order, and the same is true for WanderJoin. Notably, in WordNet, the Q-error associated with the order of G-Care tends to be higher than that of QuickSI. This is due to G-Care resulting in a suboptimal order as the number of total orders grows exponentially. For queries with 4 and 8 vertices, the order of G-Care achieves slightly lower Q-error than that of QuickSI, which is presented in Figure 23 and 24.

In a nutshell, generally, both orders yield comparable performance in the terms of efficiency and accuracy for RW estimators. In the case of small queries, G-Care's ordering is marginally better than QuickSI's. However, for larger queries, G-Care's ordering can result in suboptimal performance.

Table 3: The costs of construction and transfer of candidate graphs (milliseconds) for queries with different vertices.

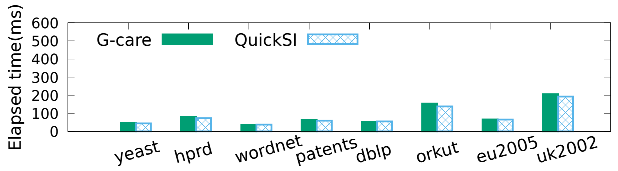
Dataset	Construction			CPU-GPU Transfer		
	4	8	16	4	8	16
Yeast	0.12	0.31	0.37	0.63	1.83	0.97
HPRD	0.20	0.17	0.37	0.98	0.55	1.01
WordNet	7.79	9.093	22.57	5.19	4.35	11.03
Patents	64.92	72.88	52.16	1.39	1.49	1.38
DBLP	0.38	0.31	0.41	1.55	1.46	1.70
Orkut	325.1	523.12	646.47	16.71	17.52	19.34
eu2005	64.59	58.13	60.48	1.53	1.74	2.09
uk2002	683.20	640.75	628.65	16.97	16.49	16.29

EVALUATION OF CANDIDATE GRAPH

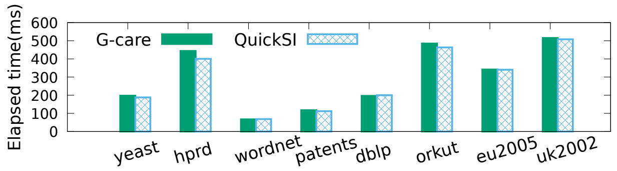
Candidate graphs provide smaller memory consumption, better cache locality and less computation workload for identifying edges on data graph. However, building and transferring it to the GPU incurs additional cost. Table 3 presents the constructing cost and transferring cost in our experiments. To further assess the impact, we compare the running time of gSWORD when using candidate graphs with the running time when not using them.

Figure 28 shows the results of queries with 16 vertices. For both Alley and WanderJoin in all datasets, the running time of sampling directly on data graph is consistently higher than the approach of sampling on the candidate graph, even when accounting for the costs associated with constructing and transferring candidate graphs. For Alley, adopting candidate graph is 34x faster than directly sampling on data graphs. And for WanderJoin adopting candidate graph is 18.5x faster than directly sampling on data graphs. Moreover, for large graphs, the performance benefits of utilizing candidate graphs becomes notably more prominent. In small graphs, like Yeast and HPRD, adopting candidate graphs reduces the average running of gSWORD by 1.5x for both Alley and WanderJoin. However, this reduction significantly escalates to 30x and 23.8x for Alley and WanderJoin, respectively, in the case of the larger eu2005 dataset.

Similar results are also achieved for queries with 4 vertices and 8 vertices in Figure 26 and 27. Generally, the benefits of using candidate graphs outweigh the associated costs.

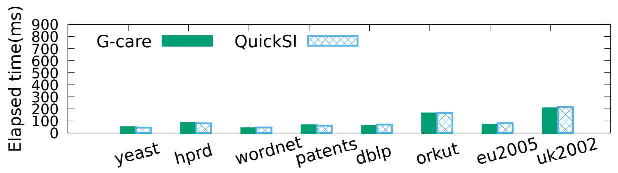


(a) WanderJoin

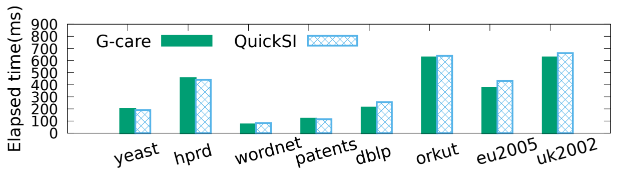


(b) Alley

Figure 20: The eslapsed time of gSWORD for 4-node queries with different matching order

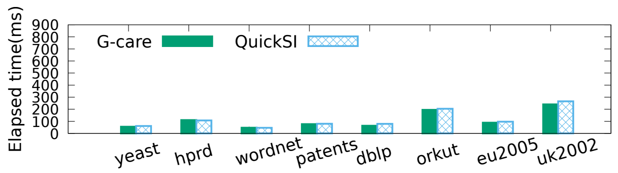


(a) WanderJoin

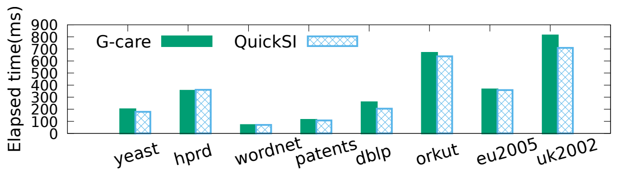


(b) Alley

Figure 21: The eslapsed time of gSWORD for 8-node queries with different matching order

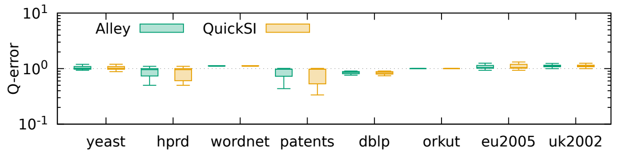


(a) WanderJoin

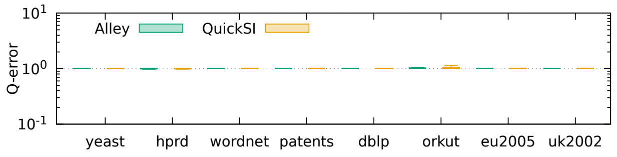


(b) Alley

Figure 22: The eslapsed time of gSWORD for 16-node queries with different matching order

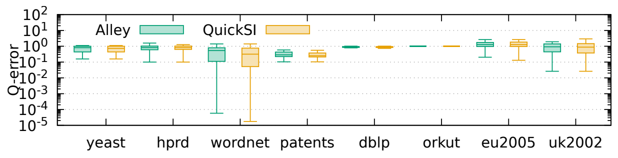


(a) WanderJoin

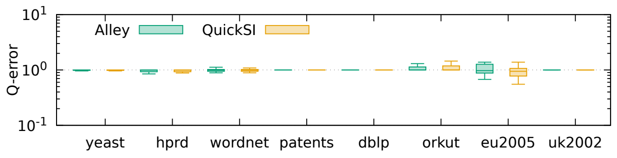


(b) Alley

Figure 23: The Q-error of gSWORD for 4-node queries with different matching order

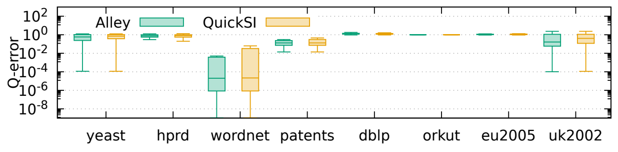


(a) WanderJoin

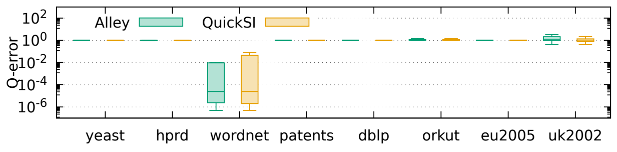


(b) Alley

Figure 24: The Q-error of gSWORD for 8-node queries with different matching order

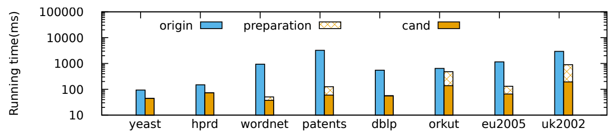


(a) WanderJoin

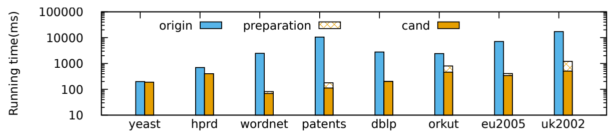


(b) Alley

Figure 25: The Q-error of gSWORD for 16-node queries with different matching order

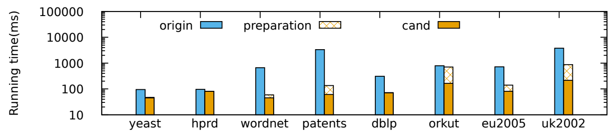


(a) WanderJoin

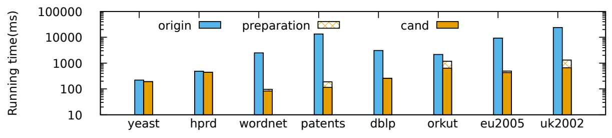


(b) Alley

Figure 26: The eslapsed time of gSWORD for 4-node queries with/without candidate graph

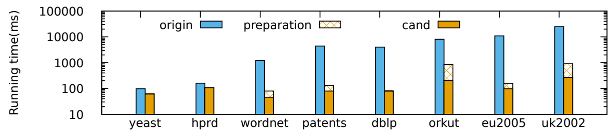


(a) WanderJoin

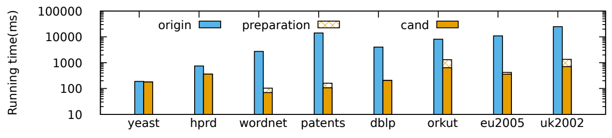


(b) Alley

Figure 27: The eslapsed time of gSWORD for 8-node queries with/without candidate graph



(a) WanderJoin



(b) Alley

Figure 28: The eslapsed time of gSWORD for 16-node queries with/without candidate graph