NosWalker: A Decoupled Architecture for Out-of-Core Random Walk Processing

Shuke Wang
Tsinghua University
Beijing, China
wangsk17@mails.tsinghua.edu.cn

Mingxing Zhang*
Tsinghua University
Beijing, China
zhang_mingxing@mail.tsinghua.edu.cn

Ke Yang
Tsinghua University & Beijing HaiZhi XingTu Technology
Beijing, China
yangke@stargraph.cn

Kang Chen
Tsinghua University
Beijing, China
chenkang@tsinghua.edu.cn

Shaonan Ma
Tsinghua University
Beijing, China
msn18@mails.tsinghua.edu.cn

Jinlei Jiang
Tsinghua University
Beijing, China
jjlei@tsinghua.edu.cn

Yongwei Wu
Tsinghua University
Beijing, China
wuyw@tsinghua.edu.cn

ABSTRACT

Out-of-core random walk system has recently attracted a lot of attention as an economical way to run billions of walkers over large graphs. However, existing out-of-core random walk systems are all built upon general out-of-core graph processing frameworks, and hence do not take advantage of the unique properties of random walk applications. Different from traditional graph analysis algorithms, the sampling process of random walk can be decoupled from the processing of the walkers. It enables the system to reserve only pre-sample results in memory, which are typically much smaller than the entire edge set. Moreover, in random walk, it is not the number of walkers but the number of steps moved per second that dominates the overall performance. Thus, with independent walkers, there is no need to process all the walkers simultaneously.

In this paper, we present NosWalker, an out-of-core random walk system that replaces the graph oriented scheduling with a decoupled system architecture that provides walker oriented scheduling. NosWalker is able to adaptively generate walkers and flexibly adjust the distribution of reserved pre-sample results in memory. Instead of processing all the walkers at once, NosWalker only tries its best to keep a few walkers able to continuously move forward. Experimental results show that NosWalker can achieve up to two orders of magnitude speedup compared to state-of-the-art out-of-core random walk systems. In particular, NosWalker demonstrates superior performance when the memory capacity can only hold about 10%-50% of the graph data, which can be a common case when the user needs to run billions of walkers over large graphs.

*Corresponding Author: Mingxing Zhang (zhang_mingxing@mail.tsinghua.edu.cn)

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9918-0/23/03.
https://doi.org/10.1145/3582016.3582025

CCS CONCEPTS

• **Theory of computation** → **Graph algorithms analysis**; • **Hardware** → **External storage**; • **Mathematics of computing** → *Probabilistic algorithms.*

KEYWORDS

graph processing, random walk, out-of-core

ACM Reference Format:
Shuke Wang, Mingxing Zhang, Ke Yang, Kang Chen, Shaonan Ma, Jinlei Jiang, and Yongwei Wu. 2023. NosWalker: A Decoupled Architecture for Out-of-Core Random Walk Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3582016.3582025

1 INTRODUCTION

Random walk is one of the most important building blocks for graph analysis algorithms [10, 16, 20–22, 27, 56, 62, 63, 74]. It serves as a foundation for many applications, such as DeepWalk [53] and node2vec [31] for node embedding, Random Walk Domination [44], Graphlet Concentration [54, 55] and Network Community Profiling [26] for graph mining, and so on. Nevertheless, as the size of graphs increases rapidly, large graphs with billions of edges are now widely used in industry [19, 69, 76], and their size can be several terabytes, exceeding the memory capacity of a single machine. As the price of memory remains relatively high, it is still expensive to build a distributed cluster that can hold all the data in memory. In contrast, the price of SSDs has fallen substantially in recent years, and their read bandwidth has increased to gigabytes per second. This makes it attractive to build out-of-core random walk processing systems that iteratively load only a necessary part

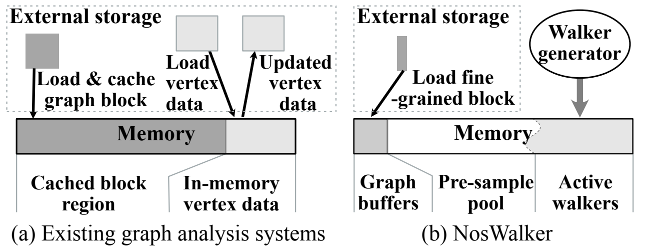

Figure 1: Memory layout of existing out-of-core graph analysis systems v.s. NosWalker.

of the graph into memory for the current analysis epoch, which democratizes large-scale graph analysis. Recent out-of-core random walk systems [39, 68] have already achieved substantial improvements in reducing random I/O and increasing I/O utilization. However, according to our investigation, these existing systems are all built upon general out-of-core graph processing frameworks, and hence cannot take advantage of many unique properties of random walk applications. We advocate a specialized architecture design to substantially accelerate the processing speed, especially when the memory capacity can only hold about 10%-50% of graph data.

Specifically, memory is undoubtedly the most precious resource in an out-of-core processing system. Thus, it is of great importance to study how existing systems use their memory, and whether it is suitable for their workloads. According to our investigation, the memory layout of all the existing out-of-core graph processing systems [12, 17, 18, 33, 36, 40, 45, 57, 60, 65, 77, 81, 83] (including existing out-of-core random walk systems [39, 42, 68]) can be viewed as a block-centric design that provides **graph oriented scheduling** as depicted in Figure 1(a). The reason for this design is that the execution and I/O patterns of most traditional graph analysis applications [28, 51, 58, 82] can be reduced into Generalized Sparse Matrix-Vector Multiplication (GSpMV) algorithms [13, 24, 75]. Thus, the focus of existing out-of-core processing systems is how to improve the read bandwidth of graph data. They carefully organize the graph data on disk so that they can load a block of edge data (a 1D [33, 36, 45, 57, 81] or 2D [12, 17, 18, 45, 65, 83] or a sliding window [40, 60] partition of the graph) in a small set of sequential disk reads using the memory layout shown in Figure 1(a). Almost directly determined by this scheduling mechanism and memory layout, the computation of these existing systems is partitioned into several disjoint epochs. In each epoch, the system is required to access only the data contained in the current block and the corresponding properties.

Although it is natural for traditional graph analysis algorithms, we find that this graph oriented scheduling is not optimal for random walk applications. For traditional graph analysis tasks, the computation states are statically attached to each vertex, where the graph oriented scheduling is appropriate. In contrast, for random walk applications, the walker states are dynamically attached to vertices. If multiple walkers currently reside on the same vertex, its vertex data will contain multiple walker states. Thus, the computation states of the random walk application are unevenly distributed among the vertices. Even worse, the walkers may jump to other vertices after each step, leading to dynamic and uneven

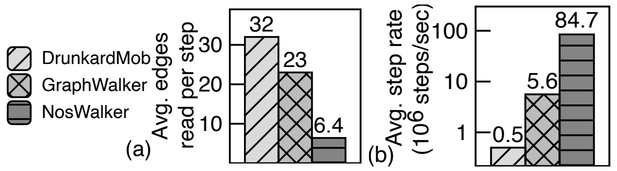

Figure 2: (a) Comparing the average edges read per step. (b) Comparing the average number of steps per second.

distribution by the computation. In this case, a pre-defined 1D/2D graph partition and the corresponding graph oriented scheduling frequently mismatches with the hot region of the walkers, and the processing would frequently stall due to waiting for the graph data to be loaded. This harms the performance.

The state-of-the-art out-of-core random walk system GraphWalker [68] also observes this problem and tries to mitigate it via (1) prioritizing the loading order of blocks so that blocks with more walkers (the hotter blocks) are loaded earlier, and (2) taking advantage of CLIP [12]'s re-entry method to allow walkers to jump more than one step in each epoch. However, since the current block is only a small part of the large graph, the walkers tend to jump out of it after moving forward. Even though GraphWalker tries to load the current hottest block into memory, this block will **cool down very quickly** after a few steps, so the system will soon have to wait for another disk I/O to refill the memory. In other words, more disk loading is needed to move the walkers forward. As we will discuss later in §5.1, this "cool down" process also greatly reduces the effectiveness of existing "dynamic" graph partition techniques. To demonstrate this problem, we measure the "average edges read per step" metric for two out-of-core random walker systems, GraphWalker and DrunkardMob [39, 68]. The metric is the average number of loaded edges that walkers are moved forward for each step. In graph oriented scheduling system, this metric can be computed by counting the total number of loaded edges divided by the total number of steps moved. As we can see from Figure 2(a), these existing systems require much more edges to be loaded per step than NosWalker, which leads to the significant performance gap in Figure 2(b). This gap hurts the performance of the existing systems even further when the graph has edge properties.

To improve the performance of random walk applications, we exploit two unique properties that distinguish random walk from traditional graph analysis algorithms. These properties enlighten us to concentrate on walkers rather than the graph, leading to a **walker oriented scheduling**. *Property (a)* the sampling process of random walk is related to only edge data. Thus, the processing of sampling can be decoupled from the processing of walkers, which are only related to vertices. After reading a complete outgoing edge set of a vertex, the system can use pre-sampling technique [72] to compute multiple sampling results and store them for future use. The key advantage of this pre-sampling technique is that the sampled results can be viewed as a succinct representation of the original edge data **whose size are largely reduced**. *Property (b)* the walkers are independent from each other. It means that there is **no need to coordinate the pace among different walkers**. The real critical performance metric is the step throughput, i.e. the number of steps moved per second, rather than the number of walkers moved per second. In other words, instead of caring about all the walkers, we just need to continuously make sure that there is a small set of walkers (no need to be much larger than the number of threads) that can be moved forward.

Based on the above two properties, we design a specialized decoupled memory layout (Figure 1(b)) for NosWalker, which provides walker oriented scheduling. It gives the system more flexibility to reserve needed data in memory. The design of this novel architecture is based on the decomposability of random walk, and is therefore walker oriented. As we can see from Figure 1, instead of the cached block region, a pre-sample result pool is located in the center of the memory layout. Taking advantage of Property (a), this design decouples the disk loading part and the walker processing part of our system. The disk loading part reads the graph data and fills the pre-sample results into this pool. The walker processing part can keep moving as long as there are enough results in the pool. A key benefit of this decoupled layout is that the system can flexibly adjust the distribution of reserved sampling results in the pool as opposed to being constrained in the currently loaded block. Thus, with a careful pre-sampling strategy, the distribution of reserved samples can represent our prediction of not only the current, but also the future hot regions of the entire graph. To achieve this goal, once a block is loaded from disk, if the system forecasts that other walkers may jump into this block soon, edges more than currently needed will be sampled from the block, the surplus sampled edges will be reserved in the pool.

The walker management of NosWalker is also very different from existing systems. Based on Property (b), we keep only a small number of walkers in memory and never spill them out to disk, so that the moving walkers never get stall due to swapping between memory and disk. This management is possible for random walk applications because, since all these walkers are independent of each other, we can simply generate new walkers after old walkers terminate. We also propose several implementation optimizations, such as adaptive granularity of disk I/O and long-tail mitigation described in §3.3.

We evaluate NosWalker on several real-world and synthetic graph datasets with up to more than one hundred billion edges and a set of representative real-world random walk applications. According to our evaluation, NosWalker achieves up to two orders of magnitude speedup compared to GraphWalker [68], Drunkardmob [39], and GraSorw [42] which are state-of-the-art out-of-core random walk systems. We use different settings, such as the number of walkers, the length of walkers, the memory budget, the number of SSDs, and the sensitivity to the graph structure to demonstrate the impact and analyze the reason for our optimizations. We also compare with two state-of-the-art in-memory random walk systems [61, 73], which shows that NosWalker's performance is comparable to parallel in-memory processing in several realistic settings.

2 BACKGROUND AND MOTIVATION

In this section, we first present some background on random walk processing. Then, using a thorough example (Figure 3), we introduce the state-of-the-art out-of-core random walk systems and their limitations, which motivate our novel system.

2.1 Random Walk

Random walk is a stochastic process consisting of a succession of random steps on the graph [7]. Each "walker" starts from a given vertex and executes a random step by 1) randomly sampling an edge from the outgoing edges of the currently located vertex; and then 2) moving the walker to the destination of the sampled edge. The termination condition of the walker is determined by the requirements of the specific application.

As one of the most important building blocks of graph analysis applications, random walk typically serves as an upstream task of the entire application pipeline. The task is to extract a large number of random sequences from the same graph. These extracted random sequences are considered a good representation of the relationships between entities in the graph. Then, these sequences are fed to the downstream learning task of the pipeline to compute gradients and update models (e.g., a graph embedding model) [31, 53]. The exact number of sequences depends on the convergence speed of the downstream learning task, but 10^6 (for a median-size graph) to 10^9 (for a large graph) is typically sufficient and has been used by existing works [39, 42, 68, 73].

More importantly, according to recent investigations, the random walk process usually dominates the cost of the whole application pipeline, such as Random Walk Domination [44], Graphlet Concentration [54, 55] and Network Community Profiling [26], etc. Many studies [6, 73, 80] have also observed that the cost of extracting the random sequences accounts for even more than 90% of the whole execution time in node embedding applications [31, 53]. Therefore, all the existing random walk systems [39, 42, 61, 68, 72, 73] focus on improving the performance of executing a large number of walkers.

2.2 Out-of-Core Random Walk System

With the price of SSD dropping, out-of-core processing has recently become more and more attractive. According to our investigation, the current prices of memory and NVMe SSD are about 9.9\$/GB [8] and 0.13\$/GB [9] respectively. Thus, if the size of the required memory can be reduced to only 10% of a graph, the storage cost can be reduced by $9.9/(0.99 + 0.13) = 8.8\times$. This calculation does not even include the additional cost of machines, high-speed networks, and management of a cluster when the graph size exceeds the memory capacity of a single machine. As a result, many systems [12, 18, 18, 33, 40, 45, 47, 57, 65, 66, 77, 77, 81, 83] have been proposed to make it practical and economical to analyze large graphs on a single machine, which is particularly valuable in terms of democratizing big data analysis.

Following this trend, DrunkardMob [39] proposes the first out-of-core random walk system that allows users to simulate billions of random walkers on large graphs. Its implementation is based on an existing graph oriented out-of-core system GraphChi [40], so that the edge data can be streamed from disk in a few sequential disk I/Os for each iteration. In contrast, the walker states (contained in the vertex data) are all held in memory. In each iteration, DrunkardMob loads a subgraph from the disk into memory and moves all walkers residing in that subgraph one step forward.

Figure 3(a) depicts a sample graph that is partitioned into two subgraphs/blocks, where $v_0 \sim v_2$ and all their outgoing edges belong to block A and the others in block B. Without loss of generality,

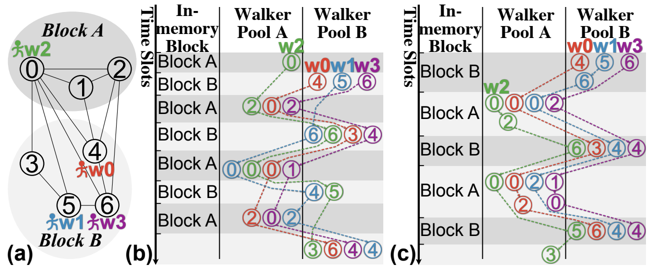

Figure 3: (a) The partitioned graph, (b) execution process in DrunkardMob and (c) execution process in GraphWalker.

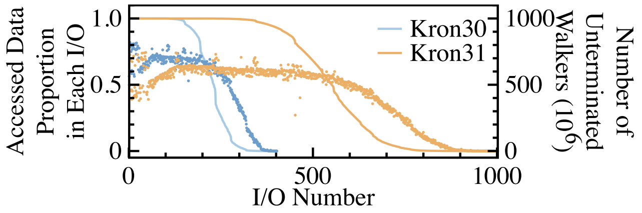

Figure 4: Results of tasks with 10^9 walkers on Kron30 and Kron31. (Line) number of unterminated walkers. (Dots) the accessed data proportion in each I/O.

we will use an application with four walkers to demonstrate the process of random walk, and each walker will terminate after five steps. As we can see from Figure 3(b), DrunkardMob first randomly initializes all the walkers (states) in memory. Then, for each computation epoch, it sequentially loads a subgraph into memory and moves each movable walker for one step. So DrunkardMob first loads block A. Since only walker w_2 locates on a vertex in block A, DrunkardMob samples an destination vertex (v_2 in our example) from the six outgoing edges ($v_0 \longrightarrow v_0, v_1, v_2, v_3, v_4, v_5, v_6$) of v_0 and then moves w_2 to v_2. In the next epoch, DrunkardMob loads the next block (block B) and moves all the walkers in that block for one step – moves w_0, w_1 and w_3 to v_0, v_6 and v_2 respectively. DrunkardMob then iterates over these two types of epochs (loading block A or B) and ends after all the walkers have been moved for five steps. DrunkardMob finishes all its work after 7 epochs and loads a total of 91 edges in our example.

2.3 Existing Optimizations

To further improve the I/O efficiency of existing out-of-core graph analysis frameworks, DynamicShards [66] attempts to dynamically adjust graph blocks to reduce the loading of useless data in each iteration. DynamicShards still follows the old graph oriented scheduling and block-centric design so that, even if there is only one active walker in the block, it still needs to load the entire block into memory.

As far as we know, GraphWalker, which is also built on GraphChi, is the current state-of-the-art out-of-core random walk system. To solve the I/O efficiency problem of iteration-based synchronized execution, GraphWalker introduces state-aware I/O model and asynchronous walker updating. In GraphWalker, the blocks with the maximum walkers can be loaded first, and walkers can move as many steps as possible on the block under processing. As shown in Figure 3(c), GraphWalker loads block B before block A because the former has more walkers. Moreover, rather than move every walker for only one step in each epoch, GraphWalker moves it as much as possible until it goes out of the current block. For example, GraphWalker first moves w_1 to v_6, which is still in block B. Thus, GraphWalker can move it one more step to v_0. In this way, the I/O efficiency is greatly improved, so as the walk performance — more than an order of magnitude speedup can be achieved over DrunkardMob. As we can see from Figure 3(c), GraphWalker finishes the work in only 5 epochs with a total of 65 edges loaded.

Nevertheless, according to our investigation, the optimizations made by GraphWalker are still not enough. The main reason is that, even though GraphWalker tries to load the current hottest block into memory, this block will *cool down very quickly* after a few steps since the current block is only a small part of the large graph. As we can see in Figure 3(c), in the toy graph, the walkers are moved out of the block in less than 2 steps on average in each epoch. This cooling phenomenon will become more apparent on real-world graphs. In our evaluation, only about 3% of the walkers remain in the currently processed block after one step when the graph is partitioned into 33 blocks.

We also measure the proportion of data accessed in each epoch of GraphWalker, which is an indirect indicator of I/O utilization (only edge data I/O, not including vertex data I/O). It is calculated by dividing the size of the data actually accessed (for moving walkers) in disk page granularity by the total size of that block. As we can see in Figure 4, even though GraphWalker uses a priority-based scheduler, the proportion of data accessed is still not high, especially when the number of walkers decreases after several epochs. This long tail problem reduces the efficiency of GraphWalker and cannot be solved simply by loading the block containing straggler walkers first. It is caused by the contradiction between the sparsity of walkers and the inflexibility of graph data blocks.

2.4 Unique Properties of Random Walk

As mentioned in §1, we observed that there are two unique properties in random walk applications that lead to the opportunity of a walker oriented scheduling. In this section, we will describe our motivation by using several evaluation results that demonstrate the potential effectiveness of these two properties. When used properly, they can substantially reduce the disk I/O for edge data and vertex data (walker states), respectively. However, as we will discuss later in this section, existing systems, such as DrunkardMob [39] and GraphWalker [68], cannot directly take advantage of these optimizations because they are built upon general out-of-core graph analysis frameworks, and thus inherit many of their basic designs. An architectural change is desired to build a specialized out-of-core random walk system.

2.4.1 Pre-Sampling Edges. For the edge data, as discussed above, we found that there is a huge gap between the size of the loaded data and the actually accessed data in random walk applications. This gap is rooted in the sampling mechanism of random walk applications, and can be mitigated by the pre-sampling mechanism demonstrated in Figure 5(a). In traditional systems, for each step on a vertex, which may have an arbitrarily large number of edges, the walker samples only one edge from the entire outgoing edge set. Thus, the actually accessed data (only one destination vertex) is much smaller than the loaded data (6 edges in our example). However, understanding this mechanism also opens up an opportunity of reducing the gap. Specifically, after reading a complete outgoing edge set of a vertex, the system can use pre-sampling technique to compute multiple numbers of sampling results and store them in memory for future use. As demonstrated in Figure 3(a), the system samples three destination vertices and uses only one of them to move the current walker w_2 from v_0 to v_2. The other 2 destination vertices (v_3 and v_4) can be reserved for future use. The key advantage of the pre-sampling technique is that the sampled results can be viewed as a succinct representation of the original edge data whose size is largely reduced (2 reserved sample results V.S. 6 edges in our example). Thus, it is possible that by holding the same size of data in memory, a better system can move the walkers for much more steps than existing systems. How we take advantage of these reserved samples by designing a novel decoupled architecture will be described in §3.

2.4.2 Dynamic Generated Walkers. Different from traditional graph analysis algorithms, the size of vertex data (walker states) in random walk is proportional to the number of walkers, not the number of vertices. As a result, the cost of loading and evicting/saving vertex data can be very large if the users need to run billions of walkers. This is why DrunkardMob tries to manage all the walkers in memory. But, even though it designs a compact data structure to represent walker states, DrunkardMob's scalability is limited by memory capacity. In contrast, GraphWalker uses a small fixed-length walker buffer to record the walker states, which is swapped out when necessary. We evaluate the overhead of this swapping in GraphWalker. The results show that it contributes up to more than 60% of the total disk I/O cost. However, although the number of walkers can be very large, these walkers are actually independent of each other. There is no need to coordinate the pace between different walkers. Since GraphWalker is still based on GraphChi, it inherits the classical architecture of first generating all the vertex data (walker states) and updating them accordingly. In contrast, we found that we only need to keep a part of the walkers in memory and never spill them out to disk. These walkers never stall due to swapping their states between memory and disk, and we can simply generate new walkers after old walkers terminate. This mechanism, if it is possible, will reduce the cost of loading and writing vertex/walker data directly to zero.

3 DESIGN OF NOSWALKER

To take advantage of the above two properties, NosWalker proposes a novel decoupled architecture that enables walker oriented scheduling. In this section, we first introduce the architecture and workflow of NosWalker, which is designed based on the key idea of keeping walkers always moving. In other words, besides trying to load more and faster graph data, NosWalker tries its best to make sure that there are always enough data in memory to move a few walkers forward. Then, we present the programming model of NosWalker and demonstrate with two examples how it can be used to implement real-world random walk applications. Finally, we present some optimization details that further improve the performance.

Algorithm 1: Workflow of NosWalker.

```
1  Function BackgroundBlockLoad():
2      $b \leftarrow AllBlocks$.Unloaded().MaxNumWalker();
3      $BlockBuffer$.Insert(LoadBlock($b$));
4  Function Processing():
5      $n \leftarrow 0$
6      while $n < TotalNumWalker$ or not $Walkers$.Empty() do
7          While $n < TotalNumWalker$ and not $Walkers$.Full() do
8              $Walkers$.Insert(GenerateWalker($n$++));
9          foreach $block$ in $BlockBuffer$ do
10             PreSample($block$);
11         If exists walkers that can be moved then
12             MoveWalkers($Walkers$);
13 Function MoveWalkers(Walker $walkers[\,]$) :
14     foreach $w$ in $walkers$ do
15         If $PEbuffer[w.location]$.Empty() then continue;
16         If not Active($w$) then $walkers$.Remove($w$);
17         If Action($w$, $PEbuffer[w.location]$.Top()) then
18             $PEbuffer[w.location]$.Pop();
19 Function PreSample(Block $block$) :
20     foreach $v$ in $block.vertices$ do
21         While not $PEbuffer[v.id]$.Full() do
22             $PEbuffer[v.id]$.Insert(Sample($v$));
```

3.1 Architecture and Workflow

Figure 6 depicts the decoupled architecture of NosWalker. The corresponding high-level workflow is presented in Algorithm 1. As we can see, there are two parts of the architecture that are decoupled by the pre-sampled edge buffers (*PEbuffer* in Algorithm 1), which is the center of our new architecture. In NosWalker, the system allocates only a small number of block buffers to support uninterrupted graph loading via a background I/O thread (lines 1-3 in Algorithm 1). The rest part of the memory is allocated for the walker pools and the pre-sampled edge buffers.

The workflow of the walker processing threads is presented in the *Processing()* function. Instead of generating all the walkers at once, NosWalker continuously generates new walkers without exceeding the memory limit (line 7). It ensures that there is no need of walker states swapping. Moreover, rather than caching the outgoing edges, NosWalker pre-samples the edge block (lines 9-10) and reserves only the pre-sampled results in memory (lines 19-22). As a result, the system can keep the walkers moving by using the pre-sampled edges (lines 17-18) even if the blocks are evicted. The key advantage of this mechanism is two fold: *1)* the size of the pre-sampled edges is much smaller than the original outgoing edge set of the vertex; *2)* the distribution of the buffered pre-sample results is not limited by the block partitioning.

A detailed example is demonstrated in Figure 5(b), which processes the same random walk task as Figure 3. NosWalker partitions the graph into smaller blocks than GraphWalker. In this example, this mechanism is demonstrated by loading one vertex per epoch. After loading all the edges of v_0, NosWalker extracts three samples from them and only one is used to move w_2 to v_2. In the second epoch, NosWalker loads all the edges of v_4 and, similarly, extracts three samples from them and uses one to move w_0. At this time, after moving w_0 to v_0, we find that although the edges of v_0 have been evicted from the memory, there are still two pre-sample results reserved in the pre-sample buffer. Thus, the pre-sample is used to move w_0 further to v_3. The following steps are also depicted in

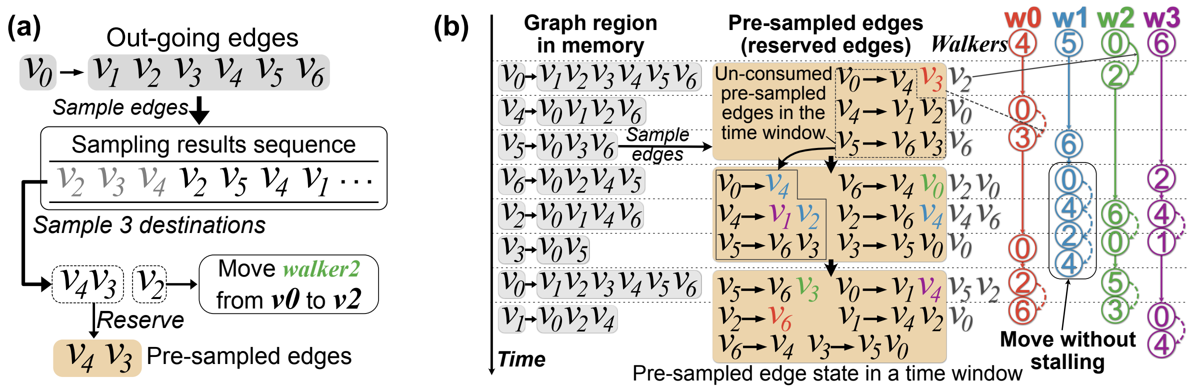
(a)
Out-going edges
Sample edges
Sampling results sequence
Sample 3 destinations
Move walker2 from v0 to v2
Reserve
Pre-sampled edges
(b)
Graph region in memory
Pre-sampled edges (reserved edges)
Walkers
w0 w1 w2 w3
Un-consumed pre-sampled edges in the time window
Sample edges
Time
Move without stalling
Pre-sampled edge state in a time window

Figure 5: (a) Pre-sampling and (b) execution process in NosWalker.

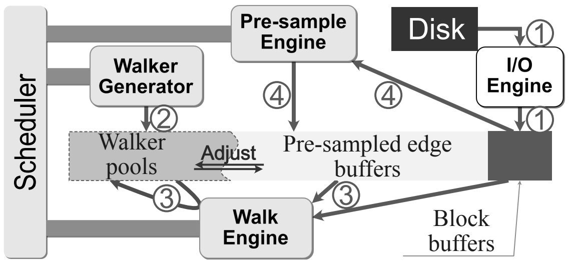

Figure 6: Architecture of NosWalker. ① Loading block to block buffer. ② Adaptively generating walkers. ③ Moving walkers. ④ Building or refilling pre-sampled edge buffers.

the figure. The leftmost part of the figure is the currently loaded vertex and its all outgoing edges, which will be discarded soon after pre-sampling. The center yellow part demonstrates the reserved pre-sample results. An outstanding example is the second step of w_1. After moving it to v_0, the system successively moves it for three steps by using only the reserved pre-samples, without the need to load any edge data. In summary, NosWalker loads only 32 edges in total to finish the same task shown in Figure 3.

3.2 Programming Model

Algorithm 2: Weighted random walk example.

```
Function GenerateWalker(Int n) : Walker
    return Walker{index: n, location: n, step: 0};
Function Sample(Vertex v) : VertexID
    return WeightedSample(v.edges, v.edgeWeights);
Function Active(Walker w) : Bool
    return $w.step = L$;
Function Action(Walker w, VertexID next) : Bool
    $w.location \leftarrow next$;
    $w.step$++;
    return True;
```

NosWalker provides four APIs for users to implement random walk applications, which are also called in Algorithm 1. As described above, **GenerateWalker** dynamically generates new walkers, which is similar to the initialization procedure of existing systems. The **Sample** function defines how to sample an edge from the outgoing edges of a vertex. This function, which describes the core logic of different random walk applications, is widely used in existing random walk systems [34, 39, 61, 68, 72, 73]. The return value of **Active** indicates whether a walker has been terminated. NosWalker calls it before moving a walker (line 16). Finally, the **Action** called in line 17 is used to implement the specific step-moving logic of random walk. It returns True to indicate that the pre-sampled edge passed in is consumed. Algorithm 2 depicts a random walk task on a weighted graph as an example. Every edge in the graph has a weight property. It issues a walker starting from each vertex with walk length L. The **GenerateWalker** is called to generate the n-th walker starting from at the vertex whose ID is n. The user implements the **Sample** to sample an edge with weight from the outgoing edges of v based on their weights. If the step is equal to L, **Active** returns True to indicate that the walker is terminated. The **Action** moves the walker w to the destination (next) of the pre-sampled edge. Each time it moves a walker, the step is increased by one and the location is updated accordingly (lines 8-9).

3.3 Implementation Challenges and Optimizations

However, the realization of the architecture change is also not straightforward. Many implementation challenges need to be solved to reduce the possibility of stalling that occurs when suitable pre-samples run out. Therefore, a compact data structure is designed to reserve as many pre-samples as possible in memory (§3.3.2) and to use the loaded data blocks to move more walkers (§3.3.5). Another important challenge is the long tail problem depicted in Figure 4. Near the end of execution, the sparsity of walkers is very high, and thus the possibility of stalling without optimizations is high. To mitigate this problem, we design several mechanisms to adaptively adjust the behavior of NosWalker, including adaptive block granularity (§3.3.1), recycling memory from finished walkers to reserve more pre-sampled edges (§3.3.3), and prioritizing the allocation of pre-sampled edges for frequently visited vertices (§3.3.2). In addition, NosWalker optimizes the inefficiency of reserving pre-sampled edges for low-degree vertices (§3.3.4).

3.3.1 Adaptive Block Granularity. Modern SSD can achieve both high throughput and high IOPS, but not simultaneously. Our benchmark shows that, for Intel SSD P4618, the sequential read bandwidth can achieve up to about 3.1 GiB/s for large block read and the IOPS can be up to more than 600k IOPS for 4 KiB random read (2.4 GiB/s).

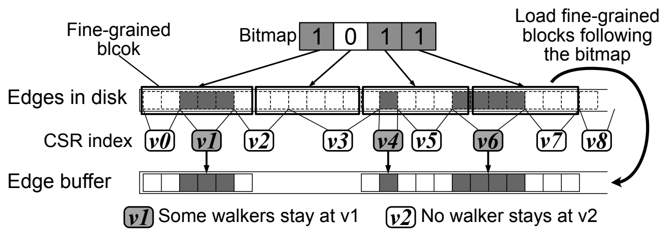
Fine-grained blcok
Bitmap 1 0 1 1
Load fine-grained blocks following the bitmap
Edges in disk
CSR index
v0 v1 v2 v3 v4 v5 v6 v7 v8
Edge buffer
v1 Some walkers stay at v1
v2 No walker stays at v2

Figure 7: Mark and load the fine-grained blocks.

It is desirable to make the best use of both these two sides of the SSD.

In random walk applications, when there are plentiful active walkers, the bottleneck of the system will be read throughput because most of the edge blocks will carry many active walkers. In contrast, as active walkers become sparse, there will be very few of them even in the hottest block. As discussed in §2.2, this long tail problem is one of the important problems that reduce the efficiency of existing out-of-core random walk systems. Thus, it is natural to consider that we can adaptively shrink the size of edge blocks according to the sparsity of walkers. As a result, we can take advantage of the high IOPS feature of modern SSD and improve I/O utilization at the same time.

In particular, when the active walkers are dense enough compared to the graph, NosWalker uses the coarse-grained block mode, which continuously loads the hottest coarse-grained block. With the help of the high-performance Linux-native asynchronous I/O access library [4], a single thread is sufficient to achieve the peak sequential read throughput of the SSD. When the active walkers become sparse, only a fraction of the data in a large block is needed. In this case, NosWalker switches to fine-grained block mode. In this mode, NosWalker tries to identify which fine-grained blocks should be loaded and launch precise I/O requests targeting those identified blocks. Since the underlying hardware of SSDs typically sets 4 KiB (one SSD page) as the smallest unit that can be read in an I/O operation, NosWalker conceptually divides each coarse-grained block into 4 KiB fine-grained blocks and issues I/O operations at 4 KiB-block-granularity. This ensures that the high IOPS of SSDs can be fully utilized while bypassing the unrequired data as accurately as possible.

Then, we introduce some implementation details to achieve the above design. We use S_G, $|W_a|$, and α to represent the size of the graph data, the number of active walkers, and the unevenness factor of the walker distribution, respectively. Since the fine-block size is 4 KiB, NosWalker switches to the fine-grained block mode when $\alpha|W_a| \cdot 4\ KiB < S_G$. We observe that the active walker density of the hottest block is usually twice the average, and walkers also distribute unevenly across the block. Therefore, the α is set to 4 by default. Since the number of walkers is monotonically decreasing, once NosWalker switches to the fine-grained block mode, it will stay in this mode until the task is completed.

In addition, to support the fine-grained block mode, NosWalker uses a bitmap to indicate whether a fine-grained block should be loaded. When the pre-sampled edges of a vertex are all consumed, some walkers may stall at these vertices and thus cannot be moved. For example, in Figure 7, the 1st block is marked due to v_1, and the 3rd and 4th blocks are marked due to v_4 and v_6. NosWalker will load the marked blocks in the future according to the bitmap. The other unmarked blocks, like the 2nd block, are therefore skipped to save I/O. As depicted in the figure, these fine-grained blocks are loaded into the corresponding addresses in the edge buffer. NosWalker can thus access the edges through the CSR index as usual. This simplifies the system implementation and improves the efficiency of edge access. NosWalker uses fine-grained block mode only when active walkers are sparse. In this case, the walker pools that initially occupy most of the memory have released a lot of memory. Thus, NosWalker can utilize this memory to accommodate the bitmap arrays and keep the CSR index in memory.

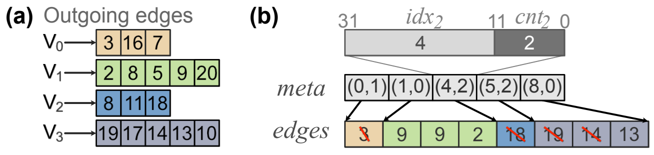

Figure 8: A pre-sampled edge buffer for the four consecutive vertices.

3.3.2 Compact Data Structure for Pre-Sample Results. In our decoupled architecture, a walker can keep moving until the pre-sampled edges of the vertex where it locates are consumed up. While the number of pre-sampled edges is limited by memory, a special data structure is needed to support more steps of walker moving. Since the vertices tend to be visited unevenly by walkers, we design a compact data structure to sample more edges for the vertex that will be visited more frequently. Here, the future frequency of visits is estimated by the number of historical visits. And this is a general method for different random walk applications even with different probability distributions.

Now we introduce the compact data structure. In NosWalker, a certain number of consecutive vertices correspond to a pre-sampled edge buffer. The buffer is similar to the CSR format as depicted in Figure 8(b). It has two arrays: `meta` and `edges`. The pre-sampled edges of a vertex are continuously stored in the `edges`. Each vertex has a metadata in the `meta` to track the status of the pre-sampled edges and the visiting history. It has two regions: idx_v(the starting position of the pre-sampled edges of v) and cnt_v (counting the consumed edges), and is represented as (idx_v, cnt_v). The pre-sampled edges of v are stored at the position between idx_v and idx_{v+1} of `edges`. This data structure compactly supports storing different numbers of pre-sampled edges for each vertex for better allocation.

Then we introduce how it works. Every time a pre-sampled edge is consumed, NosWalker increases cnt_v by one (Algorithm 1 lines 17-18) to record the consumption. And before moving a walker, NosWalker needs to ensure that the pre-sampled edges are not consumed (**Empty** method in Algorithm 1 line 15) by checking whether $idx_v + cnt_v$ is less than idx_{v+1}. If ensured, the edge stored in `edges`$[idx_v + cnt_v]$ can be the moving direction (get by **Top** method Algorithm 1 line 17) of the walker's next step. If not, NosWalker also increases cnt_v by one to record the visit. Thus, cnt_v is an estimate of the number of historical visits.

When the pre-sampled edge buffer is refilled (Algorithm 1 line 10), NosWalker reallocates the number of pre-sampled edges for each vertex based on cnt_v. Specifically, it uses the value of cnt_v as the weight to reassign the number of pre-sampled edges for the vertices.

In fact, the number of pre-sampled edges of v is approximately proportional to cnt_v. Since cnt_v can be used to approximate how often a vertex is visited, NosWalker achieves the goal of prioritizing the allocation of pre-sampled edges for frequently visited vertices. Thus, the limited buffer space can be used efficiently to move more steps for walkers and reduce the walking stalls.

Both the on-disk and in-memory edge list/pre-samples data structures used by NosWalker are based on CSR, which is a static format to which new edges cannot be added directly. Thus, the walk engine consumes (and then deletes) the pre-sampled edge simply by in-place updating the *cnt* region of the metadata. And, NosWalker will generate a new pre-sampled edge buffer according to the adaptive algorithm mentioned above when it should sample new edges. The original buffer will be released, even if there are still some unconsumed pre-sampled edges (instead of appending new samples to the original buffer). The cost is acceptable since the size of the pre-sampled edges is much smaller than the size of the graph, and the buffer manages the pre-samples for only a part of the vertices (more details in §3.3.3).

3.3.3 Manage the Dynamic Memory Space for Pre-Sampled Edges. NosWalker uses one pre-sampled edge buffer for each coarse-grained block to manage the pre-samples. And one buffer manages the pre-samples for one block of vertices. It ensures that a few bits for the *idx* region of metadata is enough to manage pre-samples, which saves memory especially when the number of vertices is large. Since the memory will be released from the finished walkers, NosWalker can recycle it to reserve more pre-samples.

Instead of initially allocating all the buffers with a small size and gradually enlarging them, NosWalker supports dynamically growing pre-sample buffers by first making the buffer size large enough and fixed, and then allocating new buffers one by one. This design has two advantages. 1) It avoids the metadata of the small buffers that occupy most of the memory. 2) When a part of the memory is released, only one memory allocation is made to allocate a new buffer, instead of reallocating memory space for each buffer.

3.3.4 Pre-Sampling on Low-Degree Vertices. Natural graphs usually have skewed power-law degree distribution [23, 30]. Thus, most vertices in these graphs have relatively few neighbors. It is not cost-effective to pre-sample edges for these low-degree vertices. For example, pre-sampling for a one-degree vertex violates the original purpose of pre-sampling, which is to make the pre-sampled edges a succinct representation of the original edge data. However, the edges of the low-degree vertices are usually only a very small part of the whole graph. For example, there are about 9% of vertices with a degree of 1 in Kron30 [3], and these vertices have only about 0.3% of the edges. This phenomenon is also observed by [72]. Therefore, NosWalker directly reserves the edges of low-degree (1 to 4 determined by the size of the graph) vertices in memory when it should pre-sample edges from them. It alleviates the drawback of the pre-sampling technique on low-degree vertices.

3.3.5 Use Loaded Edges as Pre-Sampled Edges. The pre-sampled edges are the key resources that keep the walkers always moving. However, the memory allocated for the pre-sampled edge buffers is limited. In practice, NosWalker also makes use of the loaded edge data in the block buffers to act as pre-sampled edges by prioritizing moving the walkers staying at these blocks before their eviction (just like existing out-of-core systems). Since NosWalker always loads the hottest block similar to GraphWalker, a large number of walkers can be moved without consuming the pre-sampled edges retained in memory. As discussed above, a key advantage of pre-sampling is that its distribution is not limited by block partitioning. With this optimization, the pre-sample results are only consumed if the corresponding edge block is not currently loaded, presumably avoiding this costly loading.

Table 1: Statistics of Datasets.

Dataset	$\|V\|$	$\|E\|$	CSR Size
Twitter (TW) [38]	61.6M	1.5B	6.2 GiB
YahooWeb (YH) [71]	1.4B	6.6B	37.6 GiB
Kron30 (K30) [3]	1B	32B	136 GiB
Kron31 (K31) [3]	2B	64B	272 GiB
CrawlWeb (CW) [1]	3.5B	128B	540 GiB
Weighted Kron 30 (K30W)	1B	32B	384 GiB
G12	2.7B	33B	144 GiB
α2.7	4.2B	27B	134 GiB

4 EVALUATION

In this section, we evaluate NosWalker on several real-world and synthetic graph datasets with up to more than one hundred billion edges. First, we compare the performance of NosWalker with state-of-the-art out-of-core graph random walk systems on a set of representative real-world random walk applications. We then use several micro-benchmarks to demonstrate the impact and analyze the reasoning behind our optimizations. Different settings, such as the number of walkers, the length of walkers, the memory budget, multiple SSDs, and higher-order random walks are all evaluated in the experiments.

4.1 Experiment Settings

Testbed. All experiments are performed on a machine with 2×24 Intel(R) Xeon(R) Gold 6240R @ 2.40 GHz processors. The memory budgets of all the evaluated systems are set to 64 GiB, which is about 12% of the largest graph we evaluate. We use *cgroups* [2] to force the sum of the memory used by the application and the page cache [5] within our limit. All graph data are stored on Intel(R) SSD DC P4618 Series 3.2 TB NVMe SSD unless otherwise noted.

Dataset. Table 1 lists the datasets we used. TW, YH and CW are real-world graphs. K30 and K31 are two synthetic graphs generated by the kronecker algorithm in Graph500 [3], which are widely used to evaluate graph analysis systems [29, 36, 41, 45, 48, 52, 68]. Among them, TT and YW are two graphs of small size to evaluate the in-memory performance of the systems, while K30, K31, and CW, whose size largely exceeds the memory budget in our testbed, can help to evaluate the scalability on different graphs. We randomly generate the weight property for each edge in K30 to conduct the weighted random walk experiments in §4.4. It also includes a pre-generated alias table [11, 37] for each vertex to replace the adjacent edge list, which is widely adopted in many random walk systems [61, 67, 72, 73] for a higher performance. The total weighted graph data (K30W) is about 384 GiB.

Both the real-world graphs (TW, YH, CW) and the synthetic graphs (K30, K31, K30W) listed above exhibit strong power-law characteristics with highly skewed vertex degree distributions. Therefore, to explore the sensitivity of the system performance on different graph structures, we further generate two synthetic flatter graphs, α2.7 and G12, with different characteristics. α2.7 is an approximate power-law distribution graph generated by the Configuration model [14, 50]. Its power-law constant α is set to 2.7. This setting makes its vertex degree distribution much flatter than the common power-law graphs including the six graphs above (the α of natural graphs is usually around 2 [30]). G12 is a uniform graph with each vertex connected by 12 edges.

4.2 Real-World Random Walk Applications

DrunkardMob [39] and GraphWalker [68] are the state-of-the-art out-of-core graph random walk systems. To evaluate the speedup of NosWalker against them, we use four representative and impactful real-world random walk applications: *1)* Personalized PageRank (PPR) [25, 46] is one of the most important graph analysis algorithms that measure the importance of each vertex. In our evaluation, we run 2000 random walks with length 10, which is considered to be sufficient to ensure accuracy, starting from each query source vertex to approximate the PPR; *2)* SimRank (SR) [35] is a general graph similarity measure and is used in many important data mining tasks [15, 32, 43]. The pair-wise vertex similarity $sim(a, b)$ can be interpreted as a measure of the time for two random walkers expected to meet at the same vertex if they start at vertices a and b. For each of the two vertices in a queried pair, we start 2000 random walks with length 11 to compute the expected meeting time; *3)* Random Walk Domination (RWD) [44] finds a vertex set with the maximum influence diffusion. We start a walker with length 6 from each vertex in the graph to collect the vertex visit statistics; *4)* Graphlet Concentration (GC) [54, 55] estimates the ratio of a type of graphlet in the graph. We use the graphlet triangle as a study case. We randomly start $|V|/100$ walkers of length 3 to estimate it, where $|V|$ is the vertex number of the graph.

The results of our evaluation are depicted in Figure 9. Since the reachable vertices for some vertices can be very few, there is a huge difference in the time of a random walk starting from different vertices. We measure the completion time of 1000 random selected sources in the PPR experiments and 1000 randomly selected vertex pairs in the SR experiments. DrunkardMob cannot process the large K31 and CW graphs, due to the limitations of holding all the vertex data in memory. As we can see from the figure, NosWalker achieves 6× to 64× speedup over with GraphWalker on the large datasets, K30, K31 and CW. It also achieves 3.6× to 7.9× speedup on TW and YH. The result that a larger graph leads to a larger speedup proves the scalability of NosWalker. We also measure the total disk I/O of each system, and the results show that NosWalker requires much less I/O than the other systems (Figure 2). This indicates that our optimizations achieve a better I/O utilization, which is the main reason for NosWalker's speedup.

4.3 Different Settings

In order to demonstrate the different aspects of NosWalker's performance, we also evaluate the basic random walk kernel and compare NosWalker with the state-of-the-art systems in different settings.

Different Number of Walkers. Figure 10 shows the performance with different numbers of walkers, with the walk length fixed at 10. Due to the memory limitation, DrunkardMob cannot support 10 billion walkers or large graphs such as K31 and CW. According to our evaluation, when the number of walkers is less than 10^9 in these experiments, the main bottleneck becomes graph loading, so there is no significant change in the time cost of GraphWalker and DrunkardMob. They inevitably load most parts of the graph many times. In contrast, NosWalker can use pre-sampled edges to keep walkers moving and adaptively shrink the block size to significantly reduce I/O. This explains why the speedup of NosWalker achieves up to two orders of magnitude when the number of walkers decreases.

Different Walk Lengths. Figure 11 shows the performance with different walk lengths when the number of walkers is fixed to 10^6. When the graph is smaller than the memory, for example TW and YH, the performance of NosWalker can beat others because of the more efficient walker management. When the graph is larger than the memory, e.g. K30, K31 and CW, the execution time of all three systems increases almost linearly with the walk length. But NosWalker is always 30× to 95× faster than GraphWalker. Even when the walk length is 512 for K31, NosWalker can still finish within a reasonable time while DrunkardMob and GraphWalker need several hours or even more than a day. The experiments also demonstrate the scalability of NosWalker to support long random walks with millions of walkers.

Different Memory Budget. To demonstrate the capability of NosWalker to work with low memory budgets, we conduct experiments under different memory budgets on the K30 dataset and compare them with GraphWalker. As depicted in Figure 12(a), the speedup of NosWalker has a notable improvement when the budget varies from 10% to 20%. It is because little memory can be allocated for pre-sampled edge buffers when the budget is only 10% of all states. Performance is more sensitive to the memory budget when the number of walkers is larger. In these cases, if there is enough memory to manage a large number of walkers, NosWalker can achieve a larger speedup as the number of walkers increases. Even with very limited memory budgets, NosWalker can still achieve significant performance improvements.

Performance on RAID. We also study the impact of storage devices by running experiments on RAID-0 consisting of seven Intel SSD D3 S4610 1.92 TB. Its sequential read-throughput is about 3.4 GiB/s. But the IOPS is only about 150k for 4 KiB random read. Figure 12(b) and (c) depict the experiment results on K30. Compared to the experiments on NVMe SSD, the low IOPS leads to some performance loss. However, NosWalker still achieves about 15.2× to 41× speedup on the fixed walk length experiments (Figure 12(b)) and about 17.3× to 27.4× speedup on the fixed number of walkers experiments (Figure 12(c)). It validates the broad applicability of NosWalker on different hardware configurations.

Sensitivity to the Graph Structure. We also conduct experiments on non-power-law graphs (G12 and α2.7) to evaluate the sensitivity to the graph structure. As depicted in Figure 13, there is some decrease in the speedup of NosWalker on non-power-law graphs. In the Basic-RW, PPR, and SR tasks, the number of walkers is independent of the number of vertices in the graph. The speedup of NosWalker decreases from 18× to 8×, 35× to 20× and 25× to

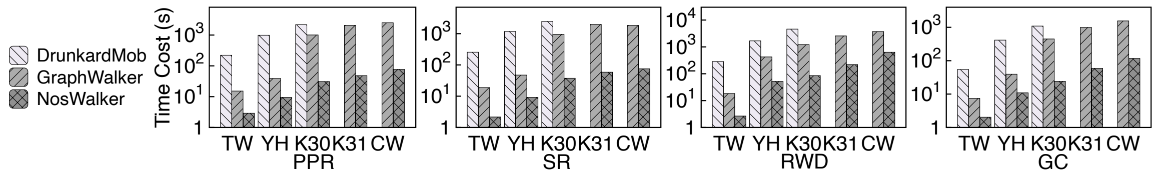

Figure 9: Performance of random walk based applications.

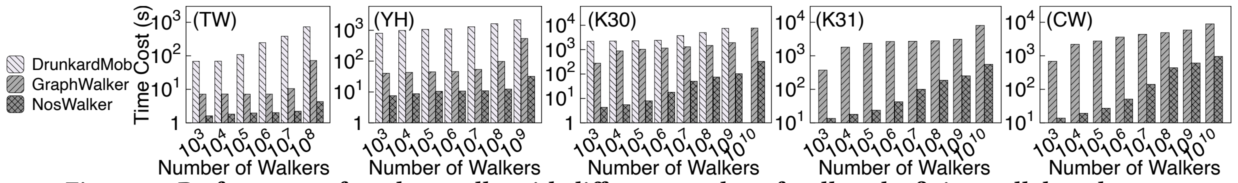

Figure 10: Performance of random walks with different number of walkers by fixing walk length as 10.

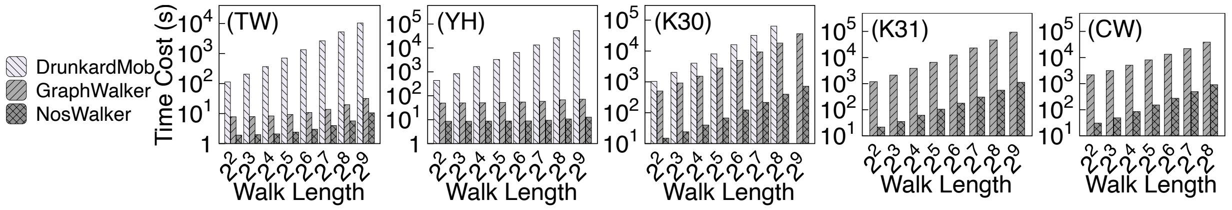

Figure 11: Performance of random walks with different walk lengths by fixing the number of walks as 1 million.

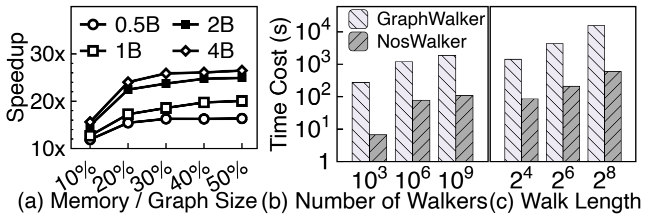

Figure 12: (a) The speedup of NosWalker relative to GraphWalker under different memory budgets with 0.5B, 1B, 2B and 4B walkers. (b) Time cost with 10 walk length on RAID-0 devices. (c) Time cost with 1 million walkers on RAID-0 devices.

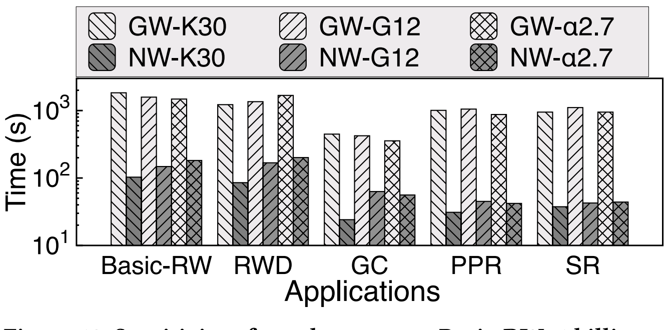

Figure 13: Sensitivity of graph structure. Basic-RW: 1 billion walkers with 10 length. Systems: GW (GraphWalker) and NW (NosWalker).

21× respectively. This is because the lower average degree makes the pre-sampling technique less effective. However, the overall speedups are still notable, because the long tail problem is still severe in the G12 and α2.7, so the effectiveness of "shrink block size" mentioned in §3.3.1 is still high. More details can be found in the following breakdown analysis (§4.4). As for applications RWD and GC, the speedup of NosWalker drops more on datasets G12 and α2.7 compared with it on K30. This is because the number of walkers is determined by the number of vertices in these applications, and the number of vertices on G12 (2.7B) and α2.7 (4.2B) is significantly higher than on K30 (1B). In these tasks with a large number of walkers, the long tail problem is not so severe. Thus, the speedup from the pre-sampling technique dominates the overall acceleration.

4.4 Optimizations Breakdown

We also conduct some experiments to measure the effectiveness of our main optimizations, namely dynamically generating walkers to support in-memory *Walker Management*, adaptively *Shrinking Block Size*, and decoupling edge sampling with moving walkers (*PreSample Edges*). In addition, we further use the weighted random walk to evaluate the efficiency of these optimizations on the weighted graph (K30W).

The three optimizations are added to the base implementation one by one, and their performance is evaluated on several applications. The base implementation has a degraded workflow similar to GraphWalker's, but is faster than GraphWalker. This mainly because NosWalker uses asynchronous I/O to replace the original buffered I/O of GraphChi and overlaps the disk I/O with computation. The disk bandwidth utilization in NosWalker is about 70% ~ 90% while it is about 20% ~ 30% in GraphWalker. Thus, using this special version of NosWalker with optimization knobs in the breakdown analysis precludes the impact of the underlying implementation details. I/O reduction breakdown analyses are also performed to cross-validate the source of the performance gain brought by these optimizations.

As we can see from the experiment results depicted in Figure 14, the performance gain of NosWalker is mainly from its ability to

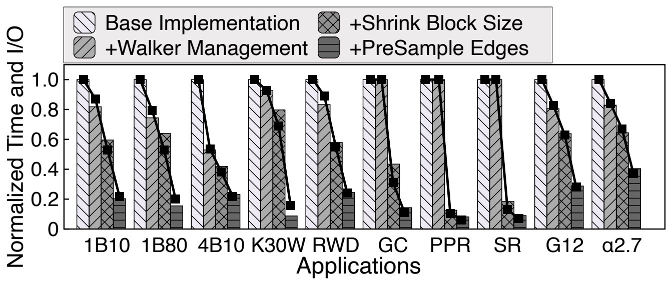

Figure 14: Effectiveness of optimizations. Bars: normalized time. Lines: normalized I/O volume. Applications: basic random walks which are 1 billion walkers with 10 length (1B10), 1 billion walkers with 80 length (1B80) and 4 billion walkers with 10 length (4B10), weighted random walk with 1 billion walkers and 80 length (K30W), four real-world random walk applications mentioned in §4.2, and 1 billion walkers with 10 length on two synthetic flatter graphs, G12 and α2.7.

reduce I/O. For example, the collected "Total I/O" results for the 1B10 workload, is decreased from 1263 GB to 1098 GB/669 GB/274 GB after adding the three major optimizations one by one. Thus, the normalized I/O is 1/0.86/0.52/0.21 for each case and it is very similar to the normalized time 1/0.81/0.60/0.20 presented in the figure.

(1) Walker Management. The baseline implements a swapping mechanism for walker states that is similar to GraphWalker's implementation. Our optimization of dynamically generating walkers is able to reduce this swapping cost and thus improve the performance. This optimization is almost independent of the graph structure, so it has a similar effect on non-power-law graphs such as G12 and α2.7. And it is natural to find that a larger number of walkers leads to a higher speedup of this optimization. In task 4B10, applying this optimization alone achieves about 1.9× speedup.

(2) Shrink Block Size. The third bar in each group of Figure 14 depicts the effect after adding the optimization of adaptively *Shrinking Block Size* described in §3.3.1. It is related to the long tail problem mentioned above. As demonstrated in Figure 4, in GraphWalker, the last 30% time of execution is used to execute the last 3% of walkers. Thus, in the task 1B10, NosWalker can achieve 35% performance improvement after just by enabling this optimization. And in contrast to the first optimization, this optimization achieves better speedup when the number of walkers becomes sparse. As we can see from the figure, the optimization achieves a significant speedup in the applications GC, PPR and SR, where the total number of walkers is in the order of million. In the PPR application, enabling this optimization achieves about 7.9× speedup. In addition, this optimization has a similar effect in non-power-law graphs. Compared to simply enabling the first optimization, it reduces about 26% and 28% I/Os in G12 and α2.7, respectively, which are close to the I/O reduction (37%) in K30.

(3) PreSample Edges. Finally, we enable pre-sampling edges and, as we can see, the performance of NosWalker is further significantly improved. The optimization is caused by the skipped I/O through buffered pre-samples. As an example, about 18% of the total steps are moved by pre-samples in the 1B10 workload, and hence the corresponding I/O is skipped. According to our further investigation, the reason why an 18% skipping in steps leads to a 60% reduction in I/O is related to the read amplification caused by block I/O (read an entire block for a single vertex) and the power-law semantics of the graph. The average degree of the vertex where the skipped steps are located is about 382, which is much higher than the average degree (32) of the whole graph, and thus leads to a higher I/O reduction. This is because the walker will visit the high-degree vertex more frequently.

As a result, for non-power-law graphs such as G12 and α2.7, the effect of pre-sampling optimization is weaker. We perform an additional evaluation on the α2.7 (about 6.4 edges per vertex). The results demonstrate that the pre-samples lead to a 32% skipping of steps but only about a 44% I/O reduction. In addition, we conduct experiments on a graph G2.5 with an even lower average degree (about 2.5), which is close to the real-world road graphs. The similar phenomenon is more evident due to the low degree. The pre-sampling technique leads to only about a 9% I/O reduction. And all the three optimizations result in a speedup of about 2× compared to the base implementation.

For the heavy weight tasks, where the number of walkers is more than a billion, NosWalker can achieve a total speedup of about 6.5× (1B80). Even for the light-weight tasks which have been greatly optimized by the first two optimizations, NosWalker achieves about 22× speedup in total. It is because our decoupling architecture and pre-sampling mechanism is a generic optimization. In the weighted random walk task, the size of the graph data is significantly larger than the original graph. Just applying the first two optimizations achieves only about 20% improvement. But NosWalker achieves a total speedup of about 11.4× after applying this optimization. This is a more significant performance improvement than it is in experiment group 1B80. It is because the pre-sampled edges stored in memory are notably smaller than the entire graph with edge properties. This optimization is particularly effective for tasks where there are properties attached to the graph.

The decoupled architecture based on walker oriented scheduling makes NosWalker achieve a high performance improvement. It can keep walkers always moving even when the blocks are not in memory, as long as the succinct pre-sample results are reserved in memory. In this way, NosWalker can make the best use of the CPU, memory, and I/O resources.

4.5 Second-Order Random Walk

Previous experiments concentrate on first-order random walk applications, which are most prevalently used in practice and assume that the sampling of the next step only relies on the information of the current vertex. To model higher-order structures in the data, researchers have also proposed second-order random walks, which select the next step based on more historical information. Taking the random walk generation of Node2Vec [31] as an example, on a given undirected graph, the edge weight for a walker w_u^v is defined as

$$\alpha_{vx} = \begin{cases} 1/p & \text{if } d_{ux} = 0 \\ 1 & \text{if } d_{ux} = 1 \\ 1/q & \text{if } d_{ux} = 2 \end{cases} \tag{1}$$

where u and v are the vertices on which w_u^v locates in the previous step and the current step respectively, p and q are hyperparameters

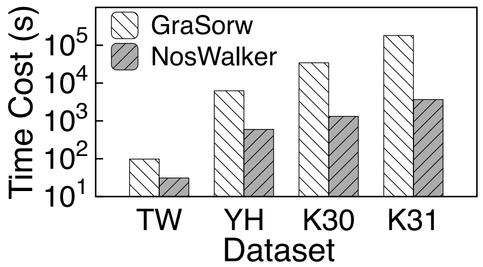

Figure 15: Performance comparing with GraSorw [42] on Node2Vec tasks.

of the model, and d_{ux} is the distance between u and x. So, in second-order random walks, the edge sampling weights depend not only on the current vertex, but also on the vertex visited by the walker in the previous step.

GraSorw [42] is a state-of-the-art disk-based system that supports second-order random walk. We compare NosWalker to it to verify the superiority of the decoupled architecture. Its walk engine is based on GraphWalker [68]. Thus, it achieves little performance improvement over GraphWalker in the first-order tasks as described in [42]. For second-order tasks, GraSorw develops a new triangular bi-block scheduling strategy, bucket-based walk management, and skewed walk storage to convert random I/Os into sequential I/Os. It also designs a learning-based block loading model to improve the I/O utilization. Compared to a naive extension from GraphWalker for the second-order task, GraSorw achieves a significant performance improvement.

We extend NosWalker to handle the second-order random walk. Using the rejection sampling method [64], NosWalker is able to decouple edge sampling from determining the direction of walkers' movement. Specifically, (1) NosWalker samples edges uniformly into the pre-sampled edge buffer. Then (2) it consumes a pre-sampled edge/neighbor x in the buffer as a candidate direction/vertex for w_v^u, and randomly generates a value $h \in [0, \max\{1/p, 1, 1/q\}]$. Finally, (3) when the outgoing edges of x are loaded in memory, it calculates the distance between u and x to decide to reject/accept this direction x based on h and Equation 1. Steps (1) and (2) together are equivalent to generating random horizontal coordinates in the rejection sampling method. It ensures the correctness of NosWalker in the second-order scenario. More details are described in Appendix A.

We conduct the random walk generation of Node2Vec [31] on 4 datasets. These datasets are converted into undirected graphs to satisfy the requirements of Node2Vec. Following the original work [31], we start 10 walkers from each vertex, and set p and q to 2 and 0.5, respectively. To get the performance results in a reasonable time, we set the walk length to 10. The evaluations on the CW dataset are removed because GraSorw takes too long to complete the task under our memory limitation. The performance comparison between NosWalker and GraSorw is depicted in Figure 15. NosWalker achieves about 3× speedup in TW due to its highly efficient walker management. And NosWalker achieves about 10× to 49× speedup in the graphs (YH, K30 and K31) whose sizes are larger than memory. It is because NosWalker additionally benefits from the decoupling between edge sampling and moving walkers.

5 RELATED WORK

5.1 Out-of-Core Graph Systems

The explosion of graph size, in contrast with expensive and precious DRAM resources, leads to the birth of many out-of-core graph systems. Most of these systems adopt an iterative computation model that repeatedly processes the input graph until convergence [18, 33, 45, 47, 65, 77, 81]. To make better use of sequential disk I/O, these systems will partition the graph into pre-defined blocks at the beginning, and then either completely or selectively load into memory in each iteration. This paradigm can be considered a block-centric design. For example, GraphChi [40], X-Stream [57], GridGraph [83] and AsyncStripe [18] all follow this paradigm. Recent systems, such as CLIP [12] and Wonderland [77], develop optimizations such as re-entry, abstract and asynchronous processing to speed up the convergence of computation, but they can also be considered to follow the block-centric design that provides graph oriented scheduling. In contrast, NosWalker proposes a novel decoupled architecture that is more suitable for random walk workloads, and hence it is much faster than existing out-of-core random walks systems [39, 68] that are built upon general graph processing frameworks.

As we have mentioned before in §2.2, several recent designs attempt to optimize the I/O utilization of existing general out-of-core processing frameworks without radically changing their architectures. Some of these optimizations are not applicable to random walk applications. For example, the dynamic partitioning mechanism in DynamicShards [66] is not helpful for graph algorithms that have non-distributive gather and apply functions, and thus random walk cannot get benefit from it. Some others have already been adopted in existing out-of-core random walk systems. LUMOS [65] proactively propagates values across iterations while simultaneously providing synchronous processing guarantees. In random walk tasks, the condition under which it can propagate values across iterations is the same as in Clip [12]. And the latter is adopted in GraphWalker[68]. The rest is orthogonal to NosWalker's optimizations.

As an illustration, Graphene [45] attempts to improve the I/O utilization by proposing an on-demand I/O strategy that dynamically adjusts the loaded graph block layout and skips loading blocks that do not contain any walkers. Figure 16 shows the performance of Graphene and NosWalker in different numbers of walkers with walk length as 10. As we can see from the figure, NosWalker achieves up to 80× speedup over Graphene. Although Graphene can dynamically adjust the loaded graph blocks, it only iterates through the loaded data in the order in which they are stored on the disks. It leads to a low I/O utilization for random walk applications, which has also been discussed in [68].

5.2 In-Memory Random Walk Systems

As random walk attracts increasing interest from both academia and industry, a bunch of works have been proposed to speed up its processing. ThunderRW [61], FlashMob [72], and [59] focus on optimizing random walk processing in the shared-memory system. FlashMob also uses pre-sampling to maximize cache utilization and avoid high memory access latency. But its sampling policy and objective are different from NosWalker. C-SAW [52], NextDoor [34],

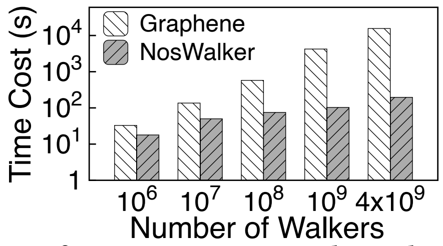

Figure 16: Performance comparing with Graphene [45] on dataset Kron30 by fixing walk length as 10.

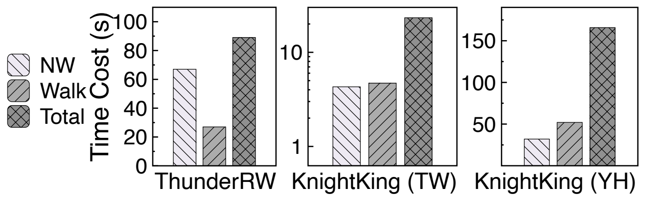

Figure 17: Comparing NosWalker (NW) with ThunderRW in Kron30 and KnightKing on Twitter and Yahoo. The bars marked with *Walk* and *Total* depict the computation time and the total time (including the data loading time) respectively.

and SkyWalker [67] accelerate random walk with GPUs, taking advantage of their high memory bandwidth and parallelism. These systems provide fast in-memory processing speed, but cannot handle large graphs beyond the memory capacity of a single server. Since ThunderRW is more memory efficient than FlashMob, we select it for comparison so that larger graphs can be evaluated in memory. The experiments are conducted on a machine with more than 360 GB memory. The results of issuing 1 billion walkers with walk length of 10, are depicted in Figure 17. As we can see from the figure, ThunderRW is about 1.5× faster than NosWalker if only the computation time is counted. On the other hand, if the data loading time is also included in the execution time of ThunderRW, the total time of ThunderRW is about 32% slower than NosWalker. It is because that about 75% of ThunderRW's time is graph loading while NosWalker can pipeline graph loading with moving walkers. These evaluation results demonstrate the efficiency of NosWalker's computation.

There are also some distributed random walk systems that require a distributed cluster connected by a high bandwidth network to hold the large graph. All these systems require to hold all the graph data in memory and therefore are less economical than out-of-core approaches. For example, KnightKing [73] is the latest distributed random walk system. We evaluate KnightKing on TW and YH. For all the datasets we have, the latter is the largest graph that KnightKing can handle on our 4-node cluster with 10 Gbps interconnection. We set 10^8 and 10^9 walkers, according to the graph scale, with walk length 10 in the experiments on datasets TW and YH, respectively. The results are depicted in Figure 17. Since there is no network communication overhead, NosWalker is about 10% and 63% faster than KnightKing, respectively, if only the computation time is counted. When the data loading time is also included in the execution time, NosWalker can achieve about 5.4× and 5.2× speedup compared to KnightKing on TW and YH respectively. According to our further experiments, when the number of nodes increases to 8, the computation performance of KnightKing is equal to (on TW) or very slightly faster than (on YH) NosWalker. The evaluation results prove the resource friendliness of NosWalker.

5.3 Works on Concurrent Query Processing

There are some works that focus on concurrent query processing. CGraph [78] and GraphM [79] are two representative out-of-core systems designed for concurrent iterative graph processing jobs. CGraph proposes a correlations-aware execution model and a core-subgraph-based scheduling algorithm to enable the jobs to efficiently share the graph structure data in cache/memory and the data accesses to the graph. GraphM achieves a similar effect by regularizing the traversal order of the graph partitions and concurrently processing the related jobs in a novel fine-grained synchronization. MultiLyra [49] is a distributed framework designed for efficient batched graph query evaluation. It optimizes query evaluation by amortizing the communication and synchronization costs between multiple queries. SimGQ+ [70] optimizes the simultaneous evaluation of a group of vertex queries originating from different source vertices on a single multicore shared-memory machine. It proposes batching techniques to amortize runtime overheads and sharing techniques to use the shared results to accelerate the query evaluation within the batch.

The systems mentioned above process a few tens to thousands of graph queries at a time. However, the footprint of each walker is much smaller than in other kinds of graph applications, so the benefit of concurrently executing a large number of walkers and sharing the overlapped I/O is limited. In contrast, NosWalker only tries its best to keep a small number of walkers always moving with the pre-samples.

6 CONCLUSION

In this paper, we present NosWalker, a novel out-of-core graph random walk system that exploits several unique properties of random walk applications. NosWalker proposes a novel walker oriented scheduling leading to an architecture that decouples the graph loading and pre-sampling process from walker processing. It enables the system to adaptively generate walkers and flexibly adjust the distribution of reserved sampling results in memory. These optimizations substantially reduce the disk I/O for vertex data and edge data, respectively, resulting in much higher I/O utilization. According to our evaluation, NosWalker can achieve up to two orders of magnitude speedups compared to the state-of-the-art out-of-core random walk systems, including DrunkardMob [39], GraphWalker [68], and GraSorw [42].

ACKNOWLEDGMENTS

We thank our shepherd and all the reviewers for their valuable comments and suggestions. This work is supported by National Key Research & Development Program of China (2022YFB2404202), Natural Science Foundation of China (62141216, 61877035), Tsinghua University Initiative Scientific Research Program, Young Elite Scientists Sponsorship Program by CAST (2022-2024), Beijing HaiZhi XingTu Technology Co., Ltd. and Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China.

A EXTEND NOSWALKER TO SECOND-ORDER RANDOM WALK

In this appendix, we elaborate on our implementation that extends NosWalker to support second-order random walk applications.

A.1 Second-Order Random Walk

With the popularity of random walks, the sampling logic has become more complex. To model higher-order structures in the data, researchers have also proposed the higher-order random walk which involves dynamic sampling. In higher-order random walks, the sampling of the next step may rely not only on the current vertex where the walker locates but also on the previous vertex (or vertices) visited by the walker. As a result, the complex random walk algorithms utilize more information about the graph topology at the cost of sampling complexity.

Node2Vec [31] is one of the representatives of second-order random walk models. We take it as an example. On a given undirected graph, the edge weights for a walker w_u^v in Node2Vec are defined as

$$\alpha_{vx} = \begin{cases} 1/p & \text{if } d_{ux} = 0 \\ 1 & \text{if } d_{ux} = 1 \\ 1/q & \text{if } d_{ux} = 2 \end{cases} \tag{2}$$

where u and v are the vertices on which w_u^v locates in the previous step and the current step, respectively, p and q are the hyper-parameters of the model, d_{ux} is the distance between u and x. Figure 18(a) depicts a sample graph segment in a Node2Vec task where the parameters p and q are set to 2 and 0.5 respectively, and u and v have the same definition as in Equation 2. The edge weights of v's outgoing edges are computed dynamically after the walker is moved from u to v and are labeled in the figure.

Since the edge weights for weighted sampling are computed dynamically based on the walking history of the walker, we cannot pre-generate the alias tables [11, 37] for each vertex to achieve efficient sampling as analyzed in [73]. The second-order random walk increases the computational complexity of sampling. Even worse, the computation of the d_{ux} in Equation 2 requires the neighbor information (outgoing edges) of vertex u or vertex x. It poses a challenge for designing the out-of-core random walk system to reduce random I/Os.

A.2 Rejection Sampling for Second-Order Random Walk in Out-of-Core Scenario

To overcome the problem, we leverage the rejection sampling method [64], which is a general method to sample from an arbitrary probability distribution. KnightKing [73] first applies rejection sampling in a distributed random walk system to improve the computational efficiency of sampling for second-order tasks. NosWalker applies it to extend the out-of-core random walk system to efficiently handle second-order tasks. Next, we introduce how the rejection sampling method is applied in second-order random walk based on the Node2Vec model [31].

We assume that a walker has walked from vertex u to vertex v and is deciding where to go next in Figure 18(a). As depicted in Figure 18(b), the rejection sampling method obtains the weighted sampling results by generating a randomly distributed two-dimensional

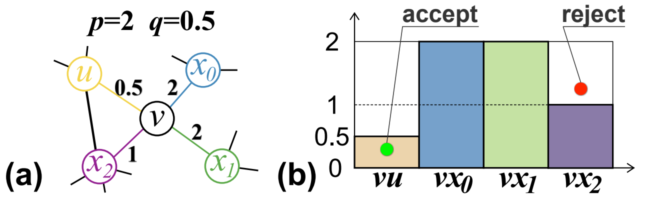

Figure 18: (a) A sample graph segment in Node2Vec [31] task. (b) Rejection sampling for Node2Vec.

coordinate (X, Y), where X is an integer corresponding to an outgoing edge of v, and Y is a floating-point number uniformly distributed in $[0, \max\{1/p, 1, 1/q\}]$. We can calculate the weight of the edge corresponding to the value of X by Equation 2. And the weight corresponds to the height of the colored rectangle of the edge in the figure. If the value Y is less than the weight, it means that the coordinate falls within the colored rectangle as the green dot in the figure depicts. The coordinate is accepted here. And the sampled edge is the one corresponding to X. On the contrary, in the case depicted by the red dot, the coordinate is rejected, and we regenerate a coordinate according to the above rules until it is accepted.

The computational cost for each attempt to generate and verify (accept/reject) a generated coordinate is $O(1)$. And $\mathbb{E}$, the average number of attempts needed to sample an edge, can be calculated as follows:

$$\mathbb{E} = \frac{\max\{1/p, 1, 1/q\} \cdot |E_v|}{\Sigma_{e \in E_v} \alpha(e)} \tag{3}$$

where E_v is the outgoing edge set of v, and $\alpha(e)$ is the edge weight of e computed by Equation 1. In the Node2Vec task, $\mathbb{E}$ can be small even with a huge graph and the total computational cost is also small.

The key advantages of the rejection sampling method for the out-of-core random walk system are not limited to the fact that the number of trials $\mathbb{E}$ is small. In particular, we can generate the coordinate in advance to obtain a candidate destination vertex c only based on v's outgoing edges, and then decide whether to accept c as the vertex of the next step when c's outgoing edges are loaded. It is not necessary that the outgoing edges of v and c are loaded at the same time. Thus, the method gives us the opportunity to avoid a lot of random I/Os. And since the coordinate is randomly generated, the process of obtaining the candidate destination vertex is independent of the edge weights that are dynamically computed by Equation 1. Thus, the pre-sampling technique in NosWalker can also be used to get the candidate destination vertex. It means that we can continue to decouple edge sampling with moving walkers.

A.3 Workflow and Programming Model

The workflow of NosWalker for second-order random walk tasks is mostly the same as it for first-order. We present the different parts in Algorithm 3. As we can see, when extended to support second-order tasks, NosWalker adds a process *RejectionProcess()* in line 7 to determine whether to accept or reject the candidate destination vertices of walkers. When the block is loaded into memory, the *RejectionProcess()* processes all the walkers whose *candidates* belong to the block (lines 11-12).

Algorithm 3: Workflow of NosWalker for second-order random walk.

```
1  Function Processing():
2      while $n < N$ or not $Walkers$.Empty() do
3          While $n < N$ and not $Walkers$.Full() do
4              $Walkers$.Insert(GenerateWalker($n$++));
5          foreach $block$ in $BlockBuffer$ do
6              PreSample($block$);
7              RejectionProcess($block$);
8          If exists walkers that can be moved then
9              MoveWalkers($Walkers$);
10 Function RejctionProcess(Block $block$) :
11     foreach $w$ in $walkers$ and $w.candidate \in block.vertices$ do
12         Rejection($w$, $block.vertices$);
```

Algorithm 4: Node2Vec based random walk example.

```
1  Function GenerateWalker(Int $n$) : Walker
2      return Walker{prev: null, curv: $n/10$, candidate: null, step: 0};
3  Function Sample(Vertex $v$) : VertexID
4      return RandomSample($v.edges$);
5  Function Active(Walker $w$) : Bool
6      return $w.step = L$;
7  Function Action(Walker $w$, VertexID $vid$) : Bool
8      If $w.candidate \neq null$ then
9          return False;
10     $w.candidate \leftarrow vid$;
11     $w.h \leftarrow$RandomFloat(0, max{$1/p$, 1, $1/q$});
12     return True;
13 Function Rejection(Walker $w$, Vertex $vertices[\,]$) :
14     If $w.prev = w.candidate$ then
15         $weight \leftarrow 1/p$;
16     elseIf $w.prev$ in $vertices[w.candidate].edges$
17         $weight \leftarrow 1$;
18     else
19         $weight \leftarrow 1/q$;
20     If $w.h \leq weight$ then
21         $w.prev \leftarrow w.curv$;
22         $w.curv \leftarrow w.candidate$;
23         $w.step$++;
24     $w.candidate \leftarrow$ null;
```

In addition to the four APIs mentioned in §3.2, NosWalker adds a **Rejection** API, which is also called in Algorithm 3 line 12, for second-order tasks. We continue to introduce these APIs with the Node2Vec model as an example in Algorithm 4. The **GenerateWalker** here dynamically generates new walkers. After being called multiple times, it will generate 10 walkers for a vertex. The $w.prev$ is set to *null* to ensure that the first step of w is uniformly distributed. The functions of **Sample** and **Active** are the same as those described in §3.2. The **Action** is called to set a candidate destination vertex $w.candidate$ for walker w. And the value $w.h$ (line 11) combined with the $w.candidate$ together form the two-dimensional coordinate, which is mentioned in §A.2, in the rejection sampling method. The **Rejection** first gets the weight *weight* corresponding to the $w.candidate$ by Equation 2 in lines 14-19. Then, it decides whether to accept the candidate destination vertex (line 20). If it accepts, it updates the w to move the walkers (lines 21-23).

NosWalker, which is designed according to the above workflow, has almost no random I/Os due to the second-order nature of walking. And since the rejection sampling method provides the opportunity to continue decoupling edge sampling and moving walkers, NosWalker can efficiently handle the second-order random walk tasks.

REFERENCES

[1] 2022. The 2012 common crawl graph. http://webdatacommons.org/.

[2] 2022. cgroups(7) Linux manual page. https://man7.org/linux/man-pages/man7/cgroups.7.html.

[3] 2022. Graph 500. https://graph500.org/.

[4] 2022. Linux-native asynchronous I/O access library. https://pagure.io/libaio.

[5] 2022. Linux System Administrators Guide: Chapter 6. Memory Management. https://tldp.org/LDP/sag/html/buffer-cache.html.

[6] 2022. Node2vec on Spark. https://github.com/aditya-grover/node2vec.

[7] 2022. Random walk. https://en.wikipedia.org/wiki/Random_walk.

[8] 2023. Price of ECC Unbuffered Memory. https://www.amazon.com/Tech-Unbuffered-Memory-PowerEdge-Server/dp/B07NQS57WZ?th=1.

[9] 2023. Price of Intel SSD D7-P5510 Series. https://www.amazon.com/Intel-SSD-D7-P5510-Series-7-68TB/dp/B08R3YQN2V.

[10] Khushbu Agarwal, Tome Eftimov, Raghavendra Addanki, Sutanay Choudhury, Suzanne Tamang, and Robert Rallo. 2019. Snomed2Vec: Random Walk and Poincaré Embeddings of a Clinical Knowledge Base for Healthcare Analytics. *arXiv preprint arXiv:1907.08650* (2019).

[11] Joachim H Ahrens and Ulrich Dieter. 1989. An alias method for sampling from the normal distribution. *Computing* 42, 2-3 (1989), 159–170. https://doi.org/10.1007/BF02239745

[12] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. 2018. Clip: A disk I/O focused parallel out-of-core graph processing system. *IEEE Transactions on Parallel and Distributed Systems* 30, 1 (2018), 45–62. https://doi.org/10.1109/TPDS.2018.2858250

[13] Michael J Anderson, Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Theodore L Willke, and Pradeep Dubey. 2016. Graphpad: Optimized graph primitives for parallel and distributed platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 313–322. https://doi.org/10.1109/IPDPS.2016.86

[14] Béla Bollobás. 1980. A probabilistic proof of an asymptotic formula for the number of labelled regular graphs. *European Journal of Combinatorics* 1, 4 (1980), 311–316. https://doi.org/10.1016/S0195-6698(80)80030-8

[15] Yuanzhe Cai, Pei Li, Hongyan Liu, Jun He, and Xiaoyong Du. 2008. S-simrank: Combining content and link information to cluster papers effectively and efficiently. In *International Conference on Advanced Data Mining and Applications*. Springer, 317–329. https://doi.org/10.1007/978-3-540-88192-6_30

[16] Mo Chen, Jianzhuang Liu, and Xiaoou Tang. 2008. Clustering via Random Walk Hitting Time on Directed Graphs.. In *Proceedings of the 23rd national conference on Artificial intelligence-Volume 2*. 616–621.

[17] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–15. https://doi.org/10.1145/3298989

[18] Shuhan Cheng, Guangyan Zhang, Jiwu Shu, and Weimin Zheng. 2016. Asyncstripe: I/o efficient asynchronous graph computing on a single server. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. 1–10. https://doi.org/10.1145/2968456.2968473

[19] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1804–1815. https://doi.org/10.14778/2824032.2824077

[20] Fan Chung and Wenbo Zhao. 2010. PageRank and random walks on graphs. In *Fete of combinatorics and computer science*. Springer, 43–62. https://doi.org/10.1007/978-3-642-13580-4_3

[21] Nick Craswell and Martin Szummer. 2007. Random walks on the click graph. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*. 239–246. https://doi.org/10.1145/1277741.1277784

[22] Wendy Ellens, Floske M Spieksma, Piet Van Mieghem, Almerima Jamakovic, and Robert E Kooij. 2011. Effective graph resistance. *Linear algebra and its applications* 435, 10 (2011), 2491–2506. https://doi.org/10.1016/j.laa.2011.02.024

[23] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. 1999. On power-law relationships of the internet topology. *ACM SIGCOMM computer communication review* 29, 4 (1999), 251–262. https://doi.org/10.1145/316194.316229

[24] Siying Feng, Jiawen Sun, Subhankar Pal, Xin He, Kuba Kaszyk, Dong-hyeon Park, Magnus Morton, Trevor Mudge, Murray Cole, Michael O'Boyle, et al. 2021. CoSPARSE: A Software and Hardware Reconfigurable SpMV Framework for Graph Analytics. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 949–954. https://doi.org/10.1109/DAC18074.2021.9586114

[25] Dániel Fogaras, Balázs Rácz, Károly Csalogány, and Tamás Sarlós. 2005. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics* 2, 3 (2005), 333–358. https://doi.org/10.1080/15427951.2005.10129104

[26] Santo Fortunato and Darko Hric. 2016. Community detection in networks: A user guide. *Physics reports* 659 (2016), 1–44. https://doi.org/10.1016/j.physrep.2016.09.002

[27] Francois Fouss, Alain Pirotte, Jean-Michel Renders, and Marco Saerens. 2007. Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation. *IEEE Transactions on knowledge and data engineering* 19, 3 (2007), 355–369. https://doi.org/10.1109/TKDE.2007.46

[28] Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. 1983. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and systems (TOPLAS)* 5, 1 (1983), 66–77. https://doi.org/10.1145/357195.357200

[29] Prasun Gera, Hyojong Kim, Piyush Sao, Hyesoon Kim, and David Bader. 2020. Traversing large graphs on GPUs with unified memory. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1119–1133. https://doi.org/10.14778/3384345.3384358

[30] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *the Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA, 17–30. https://doi.org/10.5555/2387880.2387883

[31] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 855–864. https://doi.org/10.1145/2939672.2939754

[32] Masoud Reyhani Hamedani and Sang-Wook Kim. 2016. SimRank and its variants in academic literature data: measures and evaluation. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. 1102–1107. https://doi.org/10.1145/2851613.2851811

[33] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 77–85. https://doi.org/10.1145/2487575.2487581

[34] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. 2021. Accelerating graph sampling for graph machine learning using GPUs. In *Proceedings of the 16th European Conference on Computer Systems*. ACM, 311–326. https://doi.org/10.1145/3447786.3456244

[35] Glen Jeh and Jennifer Widom. 2002. Simrank: a measure of structural-context similarity. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. 538–543. https://doi.org/10.1145/775047.775126

[36] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, et al. 2018. GraFBoost: Using accelerated flash storage for external graph analytics. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 411–424. https://doi.org/10.1109/ISCA.2018.00042

[37] Richard A Kronmal and Arthur V Peterson Jr. 1979. On the alias method for generating random variables from a discrete distribution. *The American Statistician* 33, 4 (1979), 214–218. https://doi.org/10.1080/00031305.1979.10482697

[38] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World Wide Web*. ACM, 591–600. https://doi.org/10.1145/1772690.1772751

[39] Aapo Kyrola. 2013. Drunkardmob: Billions of random walks on just a PC. In *Proceedings of the 7th ACM conference on Recommender systems*. 257–264. https://doi.org/10.1145/2507157.2507173

[40] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-scale graph computation on just a {PC}. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 31–46. https://doi.org/10.5555/2387880.2387884

[41] Kartik Lakhotia, Rajgopal Kannan, and Viktor Prasanna. 2018. Accelerating {PageRank} using {Partition-Centric} Processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 427–440. https://doi.org/10.5555/3277355.3277397

[42] Hongzheng Li, Yingxia Shao, Junping Du, Bin Cui, and Lei Chen. 2022. An I/O-efficient disk-based graph system for scalable second-order random walk of large graphs. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1619–1631. https://doi.org/10.14778/3529337.3529346

[43] Pei Li, Hongyan Liu, Jeffrey Xu Yu, Jun He, and Xiaoyong Du. 2010. Fast single-pair simrank computation. In *Proceedings of the 2010 SIAM International Conference on Data Mining*. SIAM, 571–582. https://doi.org/10.1137/1.9781611972801.50

[44] Rong-Hua Li, Jeffrey Xu Yu, Xin Huang, and Hong Cheng. 2014. Random-walk domination in large graphs. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 736–747. https://doi.org/10.1109/ICDE.2014.6816696

[45] Hang Liu and H Howie Huang. 2017. Graphene: Fine-grained {IO} management for graph computing. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*. 285–300. https://doi.org/10.5555/3129633.3129659

[46] Qin Liu, Zhenguo Li, John CS Lui, and Jiefeng Cheng. 2016. Powerwalk: Scalable personalized pagerank via random walks with vertex-centric decomposition. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 195–204. https://doi.org/10.1145/2983323.2983713

[47] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*. 527–543. https://doi.org/10.1145/3064176.3064191

[48] Daniel Margo and Margo Seltzer. 2015. A scalable distributed graph partitioner. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1478–1489. https://doi.org/10.14778/2824032.2824046

[49] Abbas Mazloumi, Xiaolin Jiang, and Rajiv Gupta. 2019. Multilyra: Scalable distributed evaluation of batches of iterative graph queries. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 349–358. https://doi.org/10.1109/BigData47090.2019.9006359

[50] Michael Molloy and Bruce Reed. 1995. A critical point for random graphs with a given degree sequence. *Random structures & algorithms* 6, 2-3 (1995), 161–180. https://doi.org/10.1002/rsa.3240060204

[51] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[52] 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 *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15. https://doi.org/10.1109/SC41405.2020.00060

[53] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 701–710. https://doi.org/10.1145/2623330.2623732

[54] Nataša Pržulj. 2007. Biological network comparison using graphlet degree distribution. *Bioinformatics* 23, 2 (2007), e177–e183. https://doi.org/10.1093/bioinformatics/btl301

[55] Natasa Pržulj, Derek G Corneil, and Igor Jurisica. 2004. Modeling interactome: scale-free or geometric? *Bioinformatics* 20, 18 (2004), 3508–3515. https://doi.org/10.1093/bioinformatics/bth436

[56] Bruno Ribeiro and Don Towsley. 2010. Estimating and sampling graphs with multidimensional random walks. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. 390–403. https://doi.org/10.1145/1879141.1879192

[57] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 472–488. https://doi.org/10.1145/2517349.2522740

[58] Semih Salihoglu and Jennifer Widom. 2013. *Computing strongly connected components in pregel-like systems.* Technical Report. Technical report, Stanford University.

[59] Yingxia Shao, Shiyue Huang, Xupeng Miao, Bin Cui, and Lei Chen. 2020. Memory-aware framework for efficient second-order random walk on large graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1797–1812. https://doi.org/10.1145/3318464.3380562

[60] Peng Sun, Yonggang Wen, Ta Nguyen Binh Duong, and Xiaokui Xiao. 2017. Graphmp: An efficient semi-external-memory big graph processing system on a single machine. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 276–283. https://doi.org/10.1109/ICPADS.2017.00045

[61] Shixuan Sun, Yuhang Chen, Shengliang Lu, Bingsheng He, and Yuchen Li. 2021. ThunderRW: An in-memory graph random walk engine.. In *Proc. VLDB Endow.*, Vol. 14. 1992–2005. https://doi.org/10.14778/3476249.3476257

[62] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. 2006. Fast random walk with restart and its applications. In *Sixth international conference on data mining (ICDM'06)*. IEEE, 613–622. https://doi.org/10.1109/ICDM.2006.70

[63] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. 2008. Random walk with restart: fast solutions and applications. *Knowledge and Information Systems* 14, 3 (2008), 327–346. https://doi.org/10.1007/s10115-007-0094-2

[64] John Von Neumann. 1951. 13. various techniques used in connection with random digits. *Appl. Math Ser* 12, 36-38 (1951), 3.

[65] Keval Vora. 2019. {LUMOS}: Dependency-Driven Disk-based Graph Processing. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 429–442. https://doi.org/10.5555/3358807.3358844

[66] Keval Vora, Guoqing Xu, and Rajiv Gupta. 2016. Load the Edges You Need: A Generic {I/O} Optimization for Disk-based Graph Processing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 507–522. https://doi.org/10.5555/3026959.3027006

[67] 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 *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 304–317. https://doi.org/10.1109/PACT52795.2021.00029

[68] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John C. S. Lui. 2020. Graph-Walker: An I/O-efficient and resource-friendly graph analytic system for fast and scalable random walks. In *2020 USENIX Annual Technical Conference (USENIX*

ATC 20). 559–571. https://doi.org/10.5555/3489146.3489184

[69] Wanjing Wei, Yangzihao Wang, Pin Gao, Shijie Sun, and Donghai Yu. 2020. A distributed multi-GPU system for large-scale node embedding at Tencent. *arXiv preprint arXiv:2005.13789* (2020).

[70] Chengshuo Xu, Abbas Mazloumi, Xiaolin Jiang, and Rajiv Gupta. 2022. SimGQ+: Simultaneously evaluating iterative point-to-all and point-to-point graph queries. *Journal of parallel and distributed computing* 164 (2022), 12–27. https://doi.org/10.1016/j.jpdc.2022.01.007

[71] Yahoo! 2002. Yahoo! AltaVista Web Page Hyperlink Connectivity Graph. https://webscope.sandbox.yahoo.com/catalog.php?datatype=g

[72] Ke Yang, Xiaosong Ma, Saravanan Thirumuruganathan, Kang Chen, and Yongwei Wu. 2021. Random Walks on Huge Graphs at Cache Efficiency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*. 311–326. https://doi.org/10.1145/3477132.3483575

[73] Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. 2019. KnightKing: A fast distributed graph random walk engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 524–537. https://doi.org/10.1145/3341301.3359634

[74] Luh Yen, Denis Vanvyve, Fabien Wouters, François Fouss, Michel Verleysen, Marco Saerens, et al. 2005. clustering using a random walk based distance measure.. In *ESANN*. 317–324.

[75] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2020. Speeding up SpMV for power-law graph analytics by enhancing locality & vectorization. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15. https://doi.org/10.1109/SC41405.2020.00090

[76] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. [n. d.]. AGL: A Scalable System for Industrial-purpose Graph Machine Learning. *Proceedings of the VLDB Endowment* 13, 12 ([n. d.]). https://doi.org/10.14778/3415478.3415539

[77] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. 2018. Wonderland: A novel abstraction-based out-of-core graph processing system. 53, 2 (2018), 608–621. https://doi.org/10.1145/3296957.3173208

[78] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. 2018. CGraph: A correlations-aware approach for efficient concurrent iterative graph processing. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 441–452. https://doi.org/10.5555/3277355.3277398

[79] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, Haikun Liu, and Yicheng Chen. 2019. GraphM: an efficient storage system for high throughput of concurrent graph processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. https://doi.org/10.1145/3295500.3356143

[80] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. 2022. ByteGNN: efficient graph neural network training at large scale. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1228–1242. https://doi.org/10.14778/3514061.3514069

[81] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. 2015. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *13th {USENIX} conference on file and storage technologies ({FAST} 15)*. 45–58. https://doi.org/10.5555/2750482.2750486

[82] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. 2008. Large-scale parallel collaborative filtering for the netflix prize. In *International conference on algorithmic applications in management*. Springer, 337–348. https://doi.org/10.1007/978-3-540-68880-8_32

[83] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC '15 Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. 375–386. https://doi.org/10.5555/2813767.2813795

Received 2022-10-20; accepted 2023-01-19