An I/O-Efficient Disk-based Graph System for Scalable Second-Order Random Walk of Large Graphs

Hongzheng Li[⋆], Yingxia Shao[⋆*], Junping Du[⋆], Bin Cui[§♭], Lei Chen[#]
[⋆]School of Computer Science (National Pilot Software Engineering School), BUPT
[§]School of CS & Key Laboratory of High Confidence Software Technologies (MOE), Peking University
[♭]Institute of Computational Social Science, Peking University (Qingdao), China
[#]Department of Computer Science and Engineering, Hong Kong University of Science and Technology
{Ethan_Lee,shaoyx}@bupt.edu.cn,junpingdu@126.com,bin.cui@pku.edu.cn,leichen@cse.ust.hk

ABSTRACT

Random walk is widely used in many graph analysis tasks, especially the first-order random walk. However, as a simplification of real-world problems, the first-order random walk is poor at modeling higher-order structures in the data. Recently, second-order random walk-based applications (e.g., Node2vec, Second-order PageRank) have become attractive. Due to the complexity of the second-order random walk models and memory limitations, it is not scalable to run second-order random walk-based applications on a single machine. Existing disk-based graph systems are only friendly to the first-order random walk models and suffer from expensive disk I/Os when executing the second-order random walks. This paper introduces an I/O-efficient disk-based graph system for the scalable second-order random walk of large graphs, called GraSorw. First, to eliminate massive light vertex I/Os, we develop a bi-block execution engine that converts random I/Os into sequential I/Os by applying a new triangular bi-block scheduling strategy, the bucket-based walk management, and the skewed walk storage. Second, to improve the I/O utilization, we design a learning-based block loading model to leverage the advantages of the full-load and on-demand load methods. Finally, we conducted extensive experiments on six large real datasets as well as several synthetic datasets.. The empirical results demonstrate that the end-to-end time cost of popular tasks in GraSorw is reduced by more than one order of magnitude compared to the existing disk-based graph systems.

PVLDB Reference Format:
Hongzheng Li, Yingxia Shao, Junping Du, Bin Cui, and Lei Chen. An I/O-Efficient Disk-based Graph System for Scalable Second-Order Random Walk of Large Graphs. PVLDB, 15(8): 1619 - 1631, 2022.
doi:10.14778/3529337.3529346

PVLDB Artifact Availability:
The source code has been made available at https://github.com/DuoLife-QNL/GraSorw.

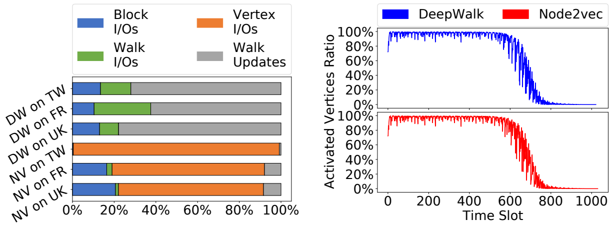
Proceedings of the VLDB Endowment, Vol. 15, No. 8 ISSN 2150-8097.
doi:10.14778/3529337.3529346

1 INTRODUCTION

Random walk has been successfully used in a variety of graph analysis tasks [7, 11, 13, 16, 17, 25, 32, 33, 35, 42, 48]. Most of the existing tasks adopt first-order random walk models [21, 33], which assume that the next vertex of a walk only relies on the information of the current vertex. However, as a simplification of real-world problems, the first-order random walk is poor at retaining historical information. Previous studies [47] show that higher-order random walk models can provide better support for graph analysis tasks by selecting the next vertex based on more historical information. Node2vec [15] is one of the most successful applications of the second-order random walk model, and for the graph embedding task, it has better performance than DeepWalk [33], which uses the first-order random walk model. For the graph proximity measurements, PageRank [18] and SimRank [17] are two popular metrics. CoSimRank [36] is proposed to reduce the computation cost in SimRank. All of these metrics adopt the first-order random walk model. In recent years, Wu et.al. [47] proposed second-order random walk-based PageRank and SimRank, and Liao et.al. put forward the second-order CoSimRank [23]. They all demonstrated that the second-order approaches achieve better results compared to the standard ones through empirical studies. Second-order random walk-based models are also widely used in community detection tasks, such as overlapping community detection [10, 12] and arc-community detection [9]. Moreover, many other interesting applications adopt second-order random walk to model different complex systems. For example, in cloud services, ServiceRank [27] and CloudRanger [45] apply the second-order random walk to identify the culprit services which are responsible for cloud incidents. For the intelligent transportation systems, R. Besenczi et.al. [8] introduced a second-order random walk-based model on dual graph [34] to analyze the traffic flow on urban streets.

Nowadays, many real-world graphs occupy hundreds of Gigabytes in CSR format, which exceeds the size of the RAM for most commodity machines. Due to the limitation of memory, it is not scalable to run random walk models on large graphs with memory-based frameworks [39, 40] in a single machine. Many general disk-based graph systems [20, 28, 51] are proposed to conduct first-order random walks on large graphs. They originally partition the whole graph into several blocks, i.e., subgraphs. During execution, these systems load a block into memory, update all the activated vertices and edges in the current block, and repeat this operation until a certain termination condition is satisfied. DrunkardMob [19] is the first

*Yingxia Shao is the corresponding author.



(a) I/O-percentage (b) Activated vertex Ratio

Figure 1: The profile of running random walk tasks in SOGW.

random walk oriented disk-based graph system, which proposes a framework to start millions of random walks simultaneously, and GraphWalker [46] follows its idea while using an asynchronous walk update method to reduce the I/O overhead further.

However, none of the existing disk-based graph systems has considered the second-order random walk model to the best of our knowledge. In this paper, we aim to design a scalable disk-based graph system for executing second-order random walk models on large graphs. We also adopt the idea of processing the whole graph into blocks to address the insufficiency of memory. The main challenge of designing such a system is to deal with the extreme I/O overhead, which is twofold:

Massive Light Vertex I/Os. When generating a random walk in existing block-based graph systems, the walk is associated with the block which contains its current vertex. Since the block is loaded into memory before processing the walk, the current vertex and its neighbors are in the block, and it is efficient to update the first-order random walk without disk I/Os. However, when processing a second-order random walk that requires both the current vertex and previous vertex information, although the current vertex is easily retrieved from memory, the previous vertex might be retrieved from any other blocks in the disk, leading to a vertex I/O. These vertex I/Os are random and light and make it extremely I/O expensive to realize second-order random walk models on existing graph systems. Figure 1(a) visualizes the costs of running DeepWalk (i.e., first-order) and Node2vec (i.e., second-order) random walk in SOGW system (introduced in Section 6.1) with three large graph datasets, and we decompose the cost into block I/Os, walk I/Os, vertex I/Os and walk updating costs. It is clear to see that, in the second-order random walk task, the efficiency bottleneck is the cost of vertex I/Os.

Low Utilization of Block I/Os. Most of the existing block-based graph systems load the whole block at once. However, when the workload of a random walk task is light, or when the task is about to finish, the activated vertex might be just a small portion of the whole block, leading to a waste of block I/Os. Figure 1(b) visualizes the activated vertex ratio with regard to the time slots when running DeepWalk and Node2vec random walk tasks in SOGW system with LiveJournal dataset. We see that at the end of the tasks (i.e., about the last 20% time slots), the ratio is close to zero. To address low block I/O utilization, DynamicShards [44] and Graphene [24] dynamically adjust the layout of graph blocks to reduce the loading of useless data, but they do not consider the random walk features. GraphWalker [46] determines a proper block size according to the total number of random walks to improve the block I/O utilization, but such a solution is static and fails when tasks are about to finish and only few walks remain in the block.

To tackle the above two problems, we propose GraSorw, an *I/O-efficient disk-based graph system for scalable second-order random walk*, which is equipped with a bi-block execution engine and a learning-based block loading model to improve the I/O efficiency. The main techniques are as follows.

Bi-block Execution Engine. To solve the problem of massive light vertex I/Os, we propose a bi-block execution engine, which keeps two blocks (i.e., current block and ancillary block) in memory, guaranteeing the previous and current vertices are in memory as well. To realize an I/O-efficient bi-block execution engine, we need to schedule the execution sequence of blocks and organize the walk states properly, to reduce the block I/Os as much as possible. First, we theoretically analyze the hardness of block scheduling problems in the disk-based graph systems and discuss the influence between block scheduling strategies and the I/O cost. Then we introduce a *triangular bi-block scheduling strategy* which eliminates half block I/Os compared to the standard scheduling strategy. Furthermore, we develop a *bucket-based in-memory walk management* approach which merges random vertex I/Os into the sequence block I/Os, and a *skewed walk storage* to ensure the correctness of the new scheduling strategy.

Learning-based Block Loading Model. To improve the utilization of block I/Os, the challenge is to capture the dynamic workloads and estimate the costs of different block loading methods. In this work, we introduce two block loading methods – full load and on-demand load. The former is the traditional block loading method, and the latter only loads activated vertices. Then we build a learning-based block loading model in GraSorw, which dynamically selects proper block loading methods based on online statistics. The model uses the linear regression method to learn cost estimation models for the two block loading methods from historical data and derives a simple threshold-based selection criterion.

Finally, combining with the above technical contributions, we carefully implement GraSorw to efficiently process second-order random walk tasks on large graphs with a single machine. Experimental results on six large datasets show that GraSorw achieves efficiency improvement of more than one order of magnitude in common second-order random tasks such as random walk generation and PageRank query using Node2vec random walk model. To summarize, our contributions are as follows:

1) We identify the I/O inefficiency of running second-order random walk models on existing disk-based graph processing systems and propose an I/O-efficient system GraSorw.

2) We propose an efficient bi-block execution engine, which equips a triangular bi-block scheduling strategy, skewed walk storage, and bucket-based in-memory walk management to eliminate massive vertex I/Os.

3) We propose a learning-based block loading model to improve the block I/O utilization when a few walks remain in the bucket.

4) We compare our GraSorw with SOGW and SGSC on real-world and synthetic large graphs. The results show that GraSorw significantly reduces the end-to-end time of second-order random walk tasks and improves the I/O efficiency. We also demonstrate the effectiveness of GraSorw for the first-order random walk.

Table 1: The symbols frequently used in this paper.

Symbol	Description
$G = (V, E)$	Graph G with a set of vertices V and a set of edges E.
$e = (u, v)$	An edge from u to v.
a_{uv}	The weight of the edge (u, v).
$N(u)$	The set of neighbors of u.
$B(v)$	The ID of the block which vertex v belongs to.
B_i	The block whose ID is i .
b_i	The bucket whose ID is i .
N_B	The total number of partitioned blocks of graph G.
w, $\mathbb{W}$	A walk and a set of walks, i.e., $\mathbb{W} = \{w\}$.
w^v	A walk which currently resides on vertex v.
w^v_u	A walk whose current vertex is v, and previous vertex is u.
$\mathbb{A}$	The set of activated vertices.
t_f	Total time of block loading and executing stage with the full-load method.
t_o	Total time of block loading and executing stage with the on-demand load method.
η_0	The threshold of selecting block loading method.

2 PRELIMINARY

A graph $G = (V, E)$ is defined by a set of vertices V, and a set of edges E. Each edge is a pair of the form $e = (u, v), u, v \in V$, where u is the source vertex and v is the destination vertex of e, and a_{vz} represents the corresponding weight. If such $e = (u, v)$ exists, then v is a neighbor of u, and we use $N(u)$ to denote the set of neighbors of u. In disk-based graph systems, a graph is partitioned into several blocks, and we use $B(v)$ to denote the ID of the block which vertex v belongs to, and the ith block is denoted as B_i. Given a graph partition, we use N_B to represent the number of partitioned blocks. The notations frequently used in this paper are listed in Table 1.

2.1 Random Walk

A random walk w on graph $G = (V, E)$ starts from a vertex, and for each step, it selects the next vertex to visit following a transition probability distribution p. In first-order random walk models, $p = p(z|v)$, which means the selection of the next vertex z only depends on the vertex v that walk w currently resides on. A walk w currently residing on vertex v is denoted by w^v, and v is called the current vertex of w. In second-order random walk models, $p = p(z|uv)$, where u is the vertex that walk w previously resided on, and v is the current vertex of w. Such a walk is denoted by w^v_u and the corresponding distribution is called edge-edge distribution. Following the edge-edge distribution, selecting the next vertex z depends on both vertex u and v.

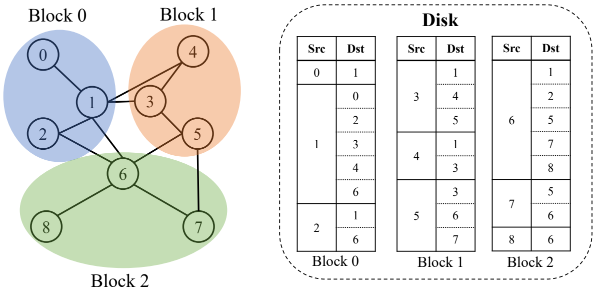
Next, we briefly review two popular random walk models.

DeepWalk model. In this paper, the DeepWalk model represents the first-order random walk model used by DeepWalk, a method of learning graph embeddings. The transition distribution in DeepWalk model is $p(z|v) = a_{vz}/Z_v$, where $Z_v = \sum_{t \in N(v)} a_{vt}$. The same distribution is used in most other first-order random walk models.

Node2vec model. In this paper, the Node2vec model represents the second-order random walk used by Node2vec, which is also a method of learning graph embeddings. In this model, for walk w^v_u, we define biased weight:

$$a'_{vz} = \begin{cases} \frac{a_{vz}}{p} & h_{uz} = 0 \\ a_{vz} & h_{uz} = 1 \\ \frac{a_{vz}}{q} & h_{uz} = 2 \end{cases} \qquad (1)$$

where $z \in N(v)$, $p, q \in \mathbb{R}^+$ are two hyperparameters, and h_{vz} is the shortest hops between v and z. For edges (v, u) and (u, z), the edge-edge transition distribution $p(z|uv) = a'_{vz}/Z'_v$, where $Z'_v = \sum_{t \in N(v)} a'_{vt}$.



Src	Dst
0	1
1	0
	2
	3
	4
	6
2	1
	6

Block 0

Src	Dst
3	1
	4
	5
4	1
	3
5	3
	6
	7

Block 1

Src	Dst
6	1
	2
	5
	7
	8
7	5
	6
8	6

Block 2

Figure 2: A partitioned graph and its storage in disk.

2.2 Disk-based Random Walk System

In disk-based random walk systems, a graph is partitioned into several blocks, and only one block is loaded into memory at a time while updating the corresponding random walks. Figure 2 illustrates a partitioned graph and its storage in disk. GraphWalker [46] is a state-of-the-art disk-based random walk system. It first loads a block into memory, then loads walks belonging to that block and updates them asynchronously. These walks are called the *current walks*, and the block loaded into memory is called the *current block*. It applies a state-aware block scheduling strategy, which chooses the block with most walks in it to be the current block. The updating of a walk stops either when it moves out of the current block or when it reaches the termination condition. For the former situation, the walk is associated with the new block where it currently moves into. After updating all walks in the current block, the system chooses the next current block and updates the corresponding walks in memory. Such a cycle is called a *time slot*.

As introduced in the Introduction, there is extreme I/O overhead when realizing second-order random walk models on existing disk-based random walk systems because of the massive light vertex I/Os and low utilization of block I/Os. In the next section, we will introduce GraSorw, which is an I/O-efficient disk-based graph system for the scalable second-order random walk over large graphs.

3 OVERVIEW OF GRASORW

GraSorw is an I/O-efficient disk-based graph system for the scalable second-order random walk. Similar to previous works, the graph and intermediate walks are stored on the disk. The graph is partitioned into blocks, and each block is associated with a walk pool storing the intermediate walks. The difference is as follows: to reduce the massive vertex and block I/Os, we design a *bi-block execution engine* and a *learning-based block loading model*. Figure 3 describes the high-level execution flow of GraSorw. During the execution, the bi-block execution engine iteratively selects a block as the current block, uses the learning-based block loading model to load an ancillary block into memory, and updates the intermediate walks associated with the current block. Next, we present the execution flow of GraSorw in a time slot in detail.

In each time slot, ① the engine uses the *bucket-based in-memory walk manager* to load the intermediate walks associated with the current block into memory and merges them with the one in the in-memory walk pool forming the *current walks*. ② Then the manager

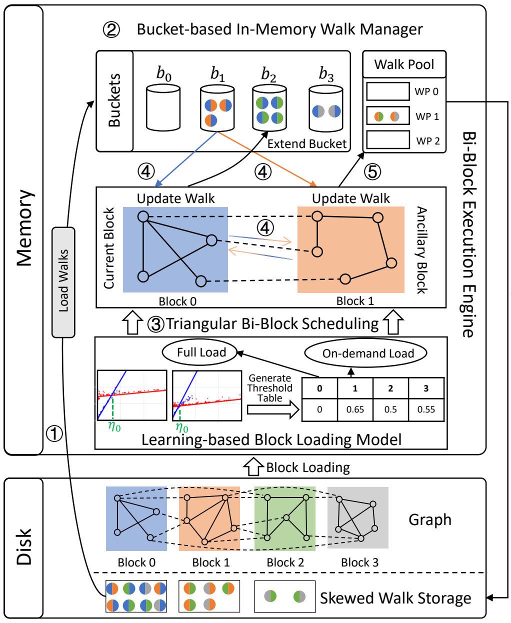


Figure 3: The execution flow of GraSorw. A semi-circle is a vertex in a walk, and the color of a semi-circle indicates the block that the vertex belongs to. Best viewed in color.

splits current walks into buckets, and each bucket stores the walks having the same block set, in which the pair of blocks contains their previous and current residing vertices. Such bucket-based in-memory walk management merges massive vertex I/Os into a single block I/O, and the details are described in Section 4.3.

Considering that the previous and current residing vertices are involved in two blocks, before processing a bucket, we need to load another block into memory, called the ancillary block. In GraSorw, each ancillary block is corresponded with a bucket, while the current block is shared among all buckets. ③ In each time slot, the bi-block execution engine uses *triangular bi-block scheduling* method to determine the loading sequence of ancillary blocks and also uses the learning-based block loading model to load the block.

After loading the blocks, ④ the engine asynchronously updates the walks in the bucket. The current vertex of walks in this bucket can be either in the current block or the ancillary block, as the blue and orange arrows show. Moreover, since there are some edges connecting the two blocks, walks can also be updated across the two blocks. The updating of a walk stops when it moves to any vertex not belonging to the blocks in memory or when the termination condition is reached. ⑤ For the former situation, walk persistence is needed to preserve the information of these intermediate walks and update them in future. Intermediate walks have two places to go. Most of them are stored in the in-memory walk pool with skewed walk storage (introduced in Section 4.3.1), and others may be moved into buckets, caused by the bucket-extending strategy introduced in Section 4.3.1. When the size of a walk pool reaches the pre-defined threshold, the in-memory walk pool is flushed to disk. After all walks in the bucket have terminated or been persisted, the next ancillary block is chosen with triangular bi-block scheduling, and the corresponding bucket is executed iteratively.

Note that the learning-based block loading model in GraSorw is proposed to improve the I/O utilization when the number of current walks is small. It uses a linear regression model to predict the cost by learning from historical logs, and on top of the model, we derive thresholds for each block and use the threshold to select the block loading method. The details are introduced in Section 5. In Figure 3, block 0 is fully loaded, and block 1 is loaded with the on-demand load method.

4 BI-BLOCK EXECUTION ENGINE

The basic idea of the bi-block execution engine is to keep two blocks (i.e., current block and ancillary block) in memory, thus guaranteeing the current and previous vertices are both in memory. To establish such an engine, we need to address the following two research questions: (RQ1) how do we schedule the two blocks to reduce block I/Os? and (RQ2) how do we manage the states of walks in disk and memory to support I/O-efficient block scheduling?

4.1 Block I/O Cost Analysis and Scheduling Strategy Revisit

Given a second-order random walk task and its input, the total number of block I/Os in GraSorw is related to two factors: the number of current block I/Os (i.e., time slots) and the number of ancillary block I/Os in each time slot. Minimizing the total number of block I/Os can be achieved by reducing the current block I/Os and the ancillary block I/Os separately. However, we show that obtaining the minimal number of current block I/Os is an NP-hard problem. In the next subsection, we present our triangular bi-block scheduling strategy, which focuses on reducing ancillary block I/Os.

Different block scheduling strategies incur different numbers of current block I/Os. We define the optimization problem of minimal current block I/Os as below:

Definition 1. ***(The minimal current block I/Os problem).*** *Given a graph $G = (V, E)$ which is partitioned into $N_{\mathbb{B}}$ blocks $\mathbb{B} = \{B_1, B_2, ..., B_{N_{\mathbb{B}}}\}$, and a task of the second-order random walk which generates M random walks, where each random walk w_i has a certain sequence of accessing current blocks, denoted by $\{B_{i0}, B_{i1}, ..., B_{ij}\}$, $0 < i \leq M$, $B_{ij} \in \mathbb{B}$. The goal is to find a global block scheduling strategy $\{B_1, B_2, ..., B_K\}$ so that all M random walks are generated and K is minimized, where K equals the number of current block I/Os.*

The following theorem presents the hardness of the problem.

Theorem 1. *The minimal current block I/Os problem is NP-hard.*

Proof. *The shortest common supersequence (SCS) problem [29, 38], which is an NP-Complete problem, can be reduced to the minimal current block I/Os problem in polynomial time. So this problem is an NP-hard problem. For more details we refer the reader to our technical report [22].*

Although there are several approximation algorithms for the SCS problem [43], they assume the sequences are known ahead. However, in our problem, the block access sequence for a second-order random walk is unknown, and we need to design an online algorithm to solve the above problem. As far as we know, most existing heuristic online solutions to the SCS problem have no (or poor) approximation error bound of the optimal solutions. Therefore, we empirically studied the different scheduling strategies [31, 46] for current blocks; and the results are reported in our technical report [22] because of the limited space. In short, the results show that no single method performs optimally on all datasets, and the performance of the same method on different datasets may vary widely. But in general, the Iteration-based method, which loads the block from B_0 to B_N iteratively, achieves the best result in most cases. With such observations, in this paper, we adopt the Iteration-based method to schedule the current block, and then focus on developing a new scheduling strategy, which optimizes the ancillary block I/Os.

4.2 Triangular Bi-Block Scheduling based Execution (RQ1)

As mentioned before, we use the Iteration-based method to schedule the current blocks, which sequentially loads the current blocks from B_0 to B_{N_B-1} iteratively into memory, and skips the loading of a current block if there is no intermediate walk in it. Due to the asynchronous walk updating method [46] in GraSorw, there is no walk whose previous vertex and current vertex are in the same block. Then for each current block, we at most process $N_B - 1$ ancillary blocks, which incurs $N_B - 1$ block I/Os in a time slot. In other words, the total block I/Os of processing the whole graph once is at most

$$N = N_B + N_B(N_B - 1) = N_B^2. \quad (2)$$

With the help of our skewed walk storage introduced in the next subsection, we can only load ancillary blocks whose ID is larger than the one of the current block. This is the new triangular bi-block scheduling strategy, and the total block I/Os is computed as follows:

$$N = N_B - 1 + \sum_{b=0}^{N_B-2} (N_B - 1 - b) = \frac{1}{2}(N_B + 2)(N_B - 1). \quad (3)$$

Compared to the Equation 2, the triangular bi-block scheduling strategy saves about 50% block I/Os.

Algorithm 1 illustrates the execution procedure on the basis of the triangular bi-block scheduling strategy. The current block ID b iterates from 0 to $N_B - 2$ (Line 2), and in each time slot the ancillary block ID iterates from $b + 1$ to $N_B - 1$ (Line 13). After choosing the current block, the associated walks are loaded into memory and collected into different buckets (Line 3). The details of bucket collection is described in Section 4.3.2. Finally, walks are processed in bucket id order (Line 16), and the update of walks in each bucket can be accelerated in parallel. Note that the correctness of Algorithm 1 is guaranteed by our skewed walk storage, which is introduced in Section 4.3.1. For more details about the correctness we refer the reader to our technical report [22], where we also discuss the space and time complexity of Algorithm 1.

Algorithm 1 Triangular Bi-Block Scheduling in GraSorw

```
1:  while has unfinished walk do
2:      for $b = 0 \to N_B - 2$ do
3:          $curWalks[\,] \leftarrow$ LoadWalks($b$)        ▷ From the skewed walk storage
4:          for $w \in curWalks[\,]$ do                  ▷ Collect bucket
5:              if PreBlockId($w$) = $b$ then
6:                  $p \leftarrow$ CurBlockId($w$)
7:              else
8:                  $p \leftarrow$ PreBlockId($w$)
9:              end if
10:             $bucket[p] \leftarrow bucket[p] \cup w$
11:         end for
12:         LoadSubGraph($b$)
13:         for $i = b + 1 \to N_B - 1$ do
14:             LoadSubGraph(i)
15:             for $w \in bucket[i]$ do
16:                 ProcessWalk($w, b, i, bucket$)        ▷ Algorithm 2
17:             end for
18:         end for
19:     end for
20: end while
```

4.3 Walk Management and Processing (RQ2)

In this subsection, we first describe the skewed walk storage, which supports the triangular bi-block scheduling strategy, and then introduce the bucket-based in-memory walk storage, which helps cluster the random vertex I/Os into blocks.

4.3.1 Skewed Walk Storage. Traditional walk storage methods associate a walk with the block to which its current vertex belongs. This brings limitations when updating walks under the triangular bi-block scheduling strategy. First, suppose that B_b is the current block and B_p is the previous block. With the traditional walk storage method, only walks currently in block B_b are loaded into memory, so only walks w_u^v such that $u \in B_p$, $v \in B_b$ get updated in the triangular bi-block scheduling strategy. The walks w_u^v such that $u \in B_b$, $v \in B_p$ are still in disk, and cannot utilize the ancillary block which has been loaded into memory more efficiently. Second, traditional walk storage cannot correctly support the triangular bi-block scheduling strategy. Because the walks currently in block B_b might have the ones of which the block ID of previous vertex is smaller than the ID of b_b, then these walks would never be updated with the triangular bi-block scheduling strategy. Therefore, we design a simple but effective skewed walk storage, which not only supports the triangular bi-block scheduling strategy but also helps update as many walks as possible in a time slot.

The skewed walk storage in GraSorw takes both the previous and current vertex of the walk into consideration to arrange the walks. Specifically, a walk w_u^v is associated with block B_i, where $i = min\{B(u), B(v)\}$. Compared to the traditional walk storage, the new storage splits the walks whose current vertices belong to the same block into two groups. One group contains the walks w_u^v such that $B(u) < B(v)$, and the other group contains the remaining walks. Consequently, in the context of the triangular bi-block scheduling strategy, the first group is processed when the corresponding $B(v)$, i.e., the blocks which their current vertices belong to, are loaded as the ancillary blocks, and the second group is processed when the corresponding $B(v)$ is loaded as the current block.

4.3.2 Bucket-based in-Memory Walk Management. As introduced in Section 3, to merge random vertex I/Os into block I/Os, the bucket-based in-memory walk manager splits the current walks into buckets, and each bucket stores the walks having the same block set in which the pair of blocks contain their previous and current residing vertices. Specifically, let $= \{b_i, 0 \le i < N_B\}$ be the set of buckets, then with the skewed walk storage, the current walks might also

Algorithm 2 Walk processing in GraSorw

```
    Parameters: walk: $w$, current block ID: $b$, ancillary block ID: $i$
 1: function ProcessWalk($w, b, i, bucket[\,]$)
 2:     $w' \leftarrow$ UpdateWalk($w, b, i$)
 3:     $cur \leftarrow$ CurBlockId($w'$)
 4:     if $cur < b$ then
 5:         AssociateWithBlock($w', cur$)
 6:     else if $b < cur < i$ then
 7:         if PreBlockId($w'$) = b then
 8:             AssociateWithBlock($w', b$)
 9:         else
10:             AssociateWithBlock($w', cur$)
11:         end if
12:     else if $cur > i$ then
13:         if PreBlockId($w'$) = b then
14:             $bucket[cur] \leftarrow bucket[cur] \cup w'$        ▷ Bucket-Extending
15:         else
16:             AssociateWithBlock($w', i$)
17:         end if
18:     end if
19: end function
20: function UpdateWalk($w, b, i$)
21:     while CurBlockId($w$) = $b$ or $i$ and walk not terminated do
22:         $w \leftarrow$ SampleDestVertex($w$)
23:     end while
24:     return $w$
25: end function
```

contain walks whose previous vertex belongs to the current block of the time slot. Let B_i be the current block, then a walk w_u^v is distributed into bucket: $\begin{cases} b_{B(v)} & \text{if} \quad u \text{ belongs to block } B_i, \\ b_{B(u)} & \text{if} \quad v \text{ belongs to block } B_i. \end{cases}$

That is to say, the bucket collection also relies on both the current vertex and the previous vertex of the walk. Furthermore, combined with the skewed walk storage, if the walk is collected to a bucket according to its previous vertex, then the ID of the block to which its current vertex belongs is smaller than that of the previous vertex, and vice versa. This walk management supports the triangular bi-block scheduling strategy.

4.3.3 *Walk Processing.*

Finally, we describe the procedure of walk processing by combining the techniques of triangular bi-block scheduling and bucket-based walk management in Algorithm 2. The association between updated walks and blocks follows the organization in the skewed walk storage, denoted by the function AssociateWithBlock, in which the walks are stored in the walk pool corresponding to the given block. In Function ProcessWalk in Algorithm 2, we first update the old walk w, and the new walk after updating is denoted as w'. Here we use pre and cur to denote the previous block ID and the current block ID of the new walk w', and b and i to represent the current block ID and ancillary block ID which are in memory now. For the new walk w', if $cur < b$ then it should be associated with block cur, since $(pre = b \text{ or } i) > cur$, as shown in Line 4 and Line 5. If cur is between b and i, the association depends on whether its previous vertex belongs to the current block or the ancillary block, as shown in Line 6 to 10. As Line 12 to 18 shows, when the new walk moves to the block whose ID is larger than the one of the ancillary blocks in memory (i.e., i), we associate it with the block where the new walk previously moves out from. An exception is shown in Line 14, where we develop a *bucket-extending* strategy to improve the efficiency further. Specifically, when the new walk is previously moved out from the current block b (i.e., pre=b), instead of associating the new walk w' with the current block b, we add it to the bucket corresponding to the current block cur of the new walk, which has not been executed as an ancillary block yet in this time slot. The bucket-extending strategy ensures new walks who meet the above condition are able to update as many steps as possible in a time slot. However, it also brings synchronization overhead when the updating of walks is executed in parallel. In our technical report [22], we describe an efficient implementation of bucket-extending.

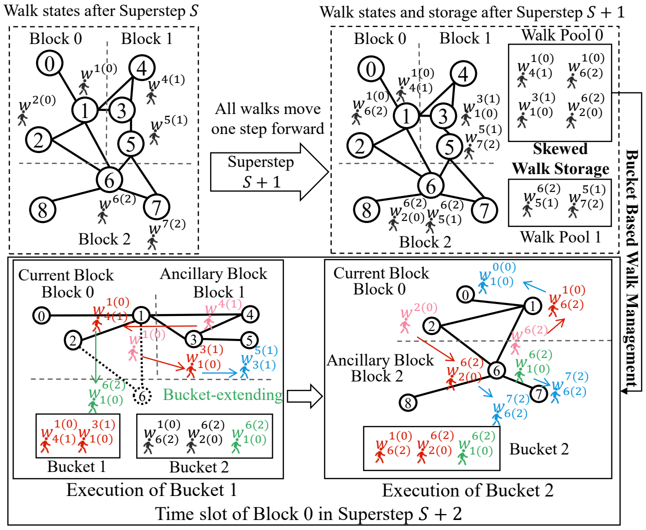


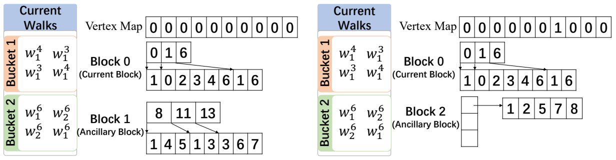
Figure 4: An example of walk processing.

During the updating, since there are two blocks in memory, the walks keep moving when they jump between the two blocks in memory, as Line 21 in Function UpdateWalk shown. Therefore, if two blocks are strongly connected (i.e., they has many edges across them), then walks can update much faster, without swapping the blocks in memory.

Figure 4 illustrates the key procedures of the walk processing with the skewed walk storage, bucket-based walk management, and the execution of buckets. Here we use $w_{u(pre)}^{v(cur)}$ to denote a walk, where $pre = B(u)$ and $cur = B(v)$. A Superstep shown in the figure represents the procedure in which all walks in the task move at least one step forward. In Superstep $S+2$, the walkers with red color are the ones being updated in their corresponding time slots, and the walkers in pink represent where the red walkers come from. The blue walkers represent where the red walkers are going to visit in the next step, and these walkers can be updated further in their time slots. The green walker is similar to the blue ones, but they have moved out the blocks in memory. In this example, the green walker satisfies the condition of the bucket extending strategy.

5 LEARNING-BASED BLOCK LOADING MODEL

The majority of block I/Os are caused by the ancillary block loading. When only a small portion of vertices in a block have walks residing



(a) Full-load of block 1.

(b) On-demand-load of block 2.

Figure 5: An example of different block loading methods.

on, it may lead to a waste of the block I/Os. To improve the I/O-utilization, we introduce two block loading methods in GraSorw, namely full load and on-demand load, and propose a learning-based model to automatically select a block loading method for ancillary block according to the run-time statistics.

5.1 Full load and On-Demand Load

Full-Load Method. This method means that a whole block is loaded into memory at once. In GraSorw, the slice of *Index File* and *CSR File* of the corresponding block is loaded into memory.

On-Demand-Load Method. This method means that only activated vertices in the corresponding block are loaded into memory. To load a block B with the on-demand-load method, we first check the current vertex and the previous vertex of each walk in the walk set $\mathbb{W}$, and record all the vertices which belong to the block B. These vertices are the activated vertices and will be used to update the walks. For each ancillary block, $\mathbb{W}$ is composed of the walks in the corresponding bucket. Then only the CSR segmentation related to the activated vertices is loaded. In GraSorw, the on-demand loading happens right before the execution of each bucket (as a replacement of Line 14 in Algorithm 1). Note that if any walk is able to update more than one step during execution and the information of the current vertex is not in memory, we should get its CSR segmentation solely from disk, which incurs few random vertex I/Os, and store it in memory as well.

Example. In Figure 5, we compare the I/O difference between full load and on-demand load through an example. Assume that there are eight walks being the current walks, and each value stored in the Index File and CSR File occupies 4 bytes in disk. The system decides to load block 2 with the on-demand-load method and load block 1 with the full-load method. The *Vertex Map* is used to record the activated vertices. Since block 1 is decided to be loaded with the full-load method as shown in Figure 5(a), the whole slice of the index file and the CSR file is loaded into memory, incurring 32 bytes I/O. After executing updates of walks in bucket 1, the memory for block 1 is freed. Before executing bucket 2, all walks in it are scanned to tally activated vertices for block 2. In the example shown in Figure 5(b), only the information of vertex 6 is needed, so the system only loads the CSR segmentation of vertex 6 into memory, and 20 bytes I/O is needed. In total, 52 bytes I/O is required to load the CSR information for the ancillary blocks. However, 64 bytes disk I/Os would be incurred if the pure full-load method is used to load both block 1 and block 2. In this example, block I/Os are saved by 18.8% by mixing the full load and on-demand load methods. Furthermore, there is no need to allocate memory to store the slice of the index file of block 2. This example implies that it is worthy of making a trade-off between two block loading methods.

5.2 Linear Regression Model for Block Loading

The key to selecting a loading method for a block is to estimate the corresponding cost. However, it is difficult to develop heuristics for the cost model since the number of activated vertices is task-dependent and different data structures between the two block loading methods also influence the efficiency. In this paper, we develop a learning-based model to predicate the cost of each loading method. Next, we will describe the model and its training method.

5.2.1 *Linear Regression Models for Cost Estimation.*

A block processing can be divided into the loading stage and the executing stage. Under full-load mode, to process a block and the corresponding $\mathbb{W}$, the whole block is loaded into memory (loading stage), and the walk updates (executing stage) are totally in memory without disk I/Os. Under on-demand load mode, only a portion of the block is loaded in the format of CSR segmentation (loading stage). The walk updates incur new disk I/Os when newly activated vertices are extended (executing stage). Compared with the full-load mode, the loading stage of on-demand load might be shorter, and the executing stage may get longer because of new disk I/Os. Therefore, we treat the two phases together as a whole to estimate the cost.

For a certain block B, let N_v be the number of total vertices in the block, and $\mathbb{A}$ be the set of activated vertices. It is intuitive that when $|\mathbb{A}|$ is very close to N_v, it should be more efficient to process the block under full-load mode than on-demand-load mode. This is because under such circumstances loading an entire block is faster than $|\mathbb{A}|$ small I/Os, which accelerates the loading stage, and since there is no need to invoke I/Os when executing walk updates, the executing stage is also faster. In random walk tasks, it is very expensive to obtain the accurate $|\mathbb{A}|$ when $|\mathbb{W}|$ is large, so we use $|\mathbb{W}|$ to roughly estimate $|\mathbb{A}|$. Let $\eta = |\mathbb{W}|/N_v$, which roughly represents the ratio of vertices whose information is needed in the block B. Let t_f and t_o be the total time of the loading and executing stage under full-load mode and on-demand load mode, respectively. Empirical studies on datasets in Table 2 show that there exists an η_0 such that in general: $\begin{cases} t_f > t_o & \text{if} \quad \eta > \eta_0; \\ t_f < t_o & \text{if} \quad \eta < \eta_0. \end{cases}$

We further find out that t_f-η follows a linear regression model $t_f = \alpha_f \eta + b_f$ for each block, and t_o-η follows $t_o = \alpha_o \eta$ when $\eta < \eta_0$. Here b_f means the cost of loading stage in full-load mode, and no intercept exists in t_o-η model because no separated loading is needed when $\mathbb{W} = \varnothing$ under on-demand-load mode.

5.2.2 *Model Training and Learning of Thresholds.*

To train the parameters α_f, b_f, α_o, we run the task twice to get the running log. Full-load mode is used for ancillary blocks in the first run, while in the second run the on-demand load mode is used. After getting the t_f-η and t_o-η running logs, we use these data to train α_f, b_f and α_o. Then we calculate $\eta_0 = \frac{b_f}{\alpha_o - \alpha_f}$, and use η_0 as the loading mode switching threshold for the ancillary block. Specifically, if $\eta > \eta_0$, full-load mode is used; otherwise, on-demand-load mode is used.

6 EXPERIMENTS

In the following sections, we evaluate the advantages of GraSorw by comparing with considerable baselines, and study the effectiveness of our technical contributions. We also study the impact of different graph partition methods to GraSorw, the parameter sensitivity of

Table 2: Graph datasets and partition information.

Graph	|V|	|E|	Text Size	CSR Size	Block Size	Block Number	Edge-Cut
LiveJournal (LJ)	4.8M	85.7M	1.2GB	364MB	20000KB	17	76.51%
Twitter (TW)	41.7M	2.4B	37GB	9.3GB	512MB	18	89.36%
Friendster (FR)	65.6M	3.6B	58GB	14GB	512MB	27	91.43%
UK200705 (UK)	105M	6.6B	6.6B	26GB	1GB	25	32.49%
Kron29 (KR)	277M	33.7B	497GB	128GB	10GB	13	92.66%
CrawlWeb (CW)	3.6B	226B	4.6TB	864GB	100GB	9	-

GraSorw, like the variation of walk distribution and block size, and the applicability of GraSorw for first-order random walks.

6.1 Experimental Settings

We carefully implement GraSorw in C++. The graph is stored in the *Compressed Sparse Row (CSR)* format and is by default sequentially partitioned into blocks according to the IDs of the vertices. For more details about the implementation we refer the reader to our technical report [22]. All experiments are run on a server with 2 Intel Xeon(R) Gold 5220 CPU and 377 GB memory. The graph data is stored on an SSD. Without specific clarification, each experiment is run in parallel, and the number of threads is set to 72.

Datasets. We use 6 datasets in our experiments: LJ [4], TW [5], FR [2], UK [6], KR, which is a synthetic graph generated by Graph500 kronecker [3] and CW [1]. The statistics are listed in Table 2. Block Size is manually set by the user. All graphs are processed into undirected. Results on more graphs with different distributions are reported in our technical report [22].

Second-order random walk models. We use Node2vec models in our experiments. Since we mainly focus on I/O performance improvement, we set two hyper-parameters p and q to 1 by default.

Benchmarks. We choose two different benchmarks:

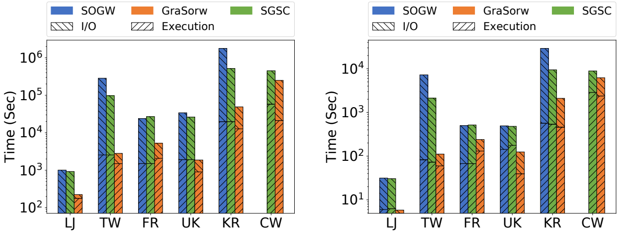
1) *Random walk generation using the Node2vec model (RWNV).* Following the random walk sampling approach in Node2vec, every node in the graph samples a set of random walks with a fixed length. Here we use the same parameters from the original work [15], and sample 10 walks per vertex with a walk length of 80.

2) *PageRank Query using the Node2vec Model (PRNV).* Given a query node v, we run the second-order random walk with restart to estimate the second-order PageRank value [47]. The decay factor is 0.85, the maximum length is 20, and the total sample size is $4|V|$. In addition, we randomly choose 10 to 100 query nodes for each dataset according to their data size.

Baselines: Since there are no existing systems designed for out-of-core second-order random walk processing, we take the following two implementations as baselines:

1) *Second-Order GraphWalker (SOGW).* The naive solution implemented on GraphWalker, which retrieves the previous vertex information directly from the disk as small vertex I/Os. We set the number of blocks in memory to 2 to make the memory cost equal to that of the other two methods. That is, if the block going to be loaded is already in memory, then no block loading is required. The block replacement strategy is the same as that of GraphWalker.

2) *Second-order GraphWalker with Static Cache (SGSC).* A static vertex cache whose size is the same as the block size is set in memory. Before the execution starts, we tally the out-degrees of all vertices in the graph and store the top-k vertices such that the degree sum is no less than the maximum edge number in one block according to the block size. There is no item replacement of the vertex cache during the execution.



(a) Efficiency of RWNV.

(b) Average query time of PRNV

Figure 6: End-to-end performance comparison.

6.2 End-to-End Performance

We first evaluate the overall performance of GraSorw compared to the two baseline systems, SOGW and SGSC. Due to the inefficiency of SOGW and SGSC, they cannot finish the tasks with standard parameters in reasonable time constraints when processing large graphs except LiveJournal. In this paper, we estimate their costs on graphs except for LiveJournal as below: According to the empirical studies in GraphWalker [46], the total time increases linearly with the walk length when it is not possible to put the whole graph in memory. Besides, we find that when running second-order random walk tasks on SOGW, since the previous vertex information should be retrieved from disk, which accounts for most of the time as shown in Figure 1(a), the total time also increases linearly with the number of walks. Therefore, we shorten the walk length for RWNV and start fewer walks from a vertex for both RWNV and PRNV. After obtaining the cost of the small-scale task, we estimate the cost by multiplying the corresponding coefficients. In addition, all results of GraSorw are obtained by running the complete task.

Figure 6 presents the results of RWNV and PRNV on various graphs. The results of SOGW on CrawlWeb are missing because the small-scale task used to estimate the total time cannot finish in two days. With this lower bound of the small-scale task execution time, we estimate that SOGW cannot finish the complete task in two weeks for both RWNV and PRNV. For SGSC, the time of the vertex cache initialization is included in I/O time. Among three systems, we can see that GraSorw achieves the best performance in both tasks on all these graphs. In particular, on Twitter, SOGW takes more than two days to finish the RWNV task, while GraSorw only takes 47 minutes, which achieves 95× speed up. On a larger graph that occupies hundreds of Gigabytes in CSR format such as Kron29, performing second-order random walk tasks is much more challenging, as the traditional disk-based methods cost about 20 days, evaluated by SOGW. Fortunately, with the help of GraSorw, such a task can be finished in half a day, which is much more reasonable. On CrawlWeb, which takes almost 900GB of memory in CSR format, GraSorw still achieves the best efficiency for both tasks, with a speedup of 1.81× for RWNV and 1.43× for PRNV, compared to SGSC. In most graphs, the SGSC is slightly faster than SOGW, as a result of the existence of the static vertex cache in memory, which makes it possible to retrieve the information of some important vertices from memory rather than by invoking vertex I/Os. However, SGSC takes more time to run such a task on Friendster. One possible reason is that for Friendster, the cache hit rate in SGSC is low so that the time of initiating the static vertex cache is longer than the time saved from its benefits. From the result comparison between SGSC and GraSorw, we can see that for

Table 3: I/O efficiency of different execution engines. Wall time is the total running time of the task, and it is decomposed into execution time, block I/O time and other overheads such as walk initiating and walk loading. Execution time is the cost of walk updating. The percentages in parentheses are the ratio of the cost of Bi-Block to the one of PB, respectively.

graph	Method	RWNV				PRNV			
		Wall Time (s)	Execution Time (s)	Block I/O Number	Block I/O Time(s)	Wall Time (s)	Execution Time (s)	Block I/O Number	Block I/O Time(s)
LJ	**PB**	332	189	13584	90	9.8	5.7	38	3
	Bi-Block	**175** (53%)	**100** (53%)	**6299** (46%)	**42** (46%)	**5.8** (6%)	**3.6** (63%)	**21** (56%)	**2** (64%)
TW	**PB**	6868	1905	15779	4463	249.8	83.5	419	138.2
	Bi-Block	**2960** (43%)	**1198** (63%)	**8090** (51%)	**1384** (31%)	**111.6** (45%)	**44.2** (53%)	**255** (61%)	**43.9** (32%)
FR	**PB**	14526	3982	34117	9743	439.9	103.6	964	283.5
	Bi-Block	**6716** (46%)	**3217** (81%)	**18550** (54%)	**2882** (3%)	**240** (55%)	**102.7** (99%)	**581** (6%)	**94.3** (33%)
UK	**PB**	20707	4143	29309	16043	554.1	102.1	659	379.6
	Bi-Block	**3789** (18%)	**744** (18%)	**10039** (34%)	**2596** (16%)	**146.5** (26%)	**32** (30%)	**312** (47%)	**81.0** (21%)
Kron29	**PB**	133491	24312	19592	104962	5793.3	827.0	878	4728
	Bi-Block	**49694** (37%)	**12738** (52%)	**11608** (59%)	**34024** (32%)	**2102.5** (36%)	**366.9** (44%)	**520** (59%)	**1582.3** (34%)
CrawlWeb	**PB**	911114	316320	6384	568576	39649	22296	100	12309.4
	Bi-Block	**249529** (27%)	**21206** (7%)	**2624** (41%)	**228256** (40%)	**6218.1** (16%)	**892.8** (4%)	**45** (45%)	**3772.6** (31%)

fixed memory size, rather than leverage the memory space to store as many large-degree vertices as possible, it is more efficient to use the memory to load blocks (i.e., ancillary blocks) with the triangular bi-block scheduling. Overall, GraSorw has achieved 1.81× to 95× performance improvement in RWNV task, and 1.43× to 19.1× improvement in PRNV task.

In the figure, we also present the time cost breakdown for each result, visualized as *Execution* time and *I/O* time. We see that the I/O time cost for each task on all graphs has decreased significantly in GraSorw. GraSorw reduces the I/O overhead most on Twitter, increasing efficiency by 213× in RWNV and 138× in PRNV, compared to SOGW. In SOGW, the expensive I/O cost comes from the massive light vertex I/Os, while in GraSorw, with the help of buckets and the ancillary block, these vertex I/Os invoked to retrieve the information of the previous vertex of walks are converted into block I/Os, which are more efficient.

6.3 The I/O-Efficiency of Bi-Block Execution Engine

Here we compare the I/O efficiency of two execution engines in GraSorw, the plain bucket engine (PB) and the bi-block execution engine (Bi-Block). The former organizes walks in buckets without the triangular bi-block scheduling strategy and the skewed walk storage. In the plain bucket engine, walks are associated with their current blocks, and the current walks are distributed to buckets according to their previous blocks. There are also two block slots in memory called the current block and the ancillary block, yet no triangular bi-block scheduling is used, where the schedule of the ancillary block starts from b_0 to b_{N_b-1}. We use the state-aware block scheduling strategy proposed by GraphWalker to schedule the current block.

The results of the two engines are shown in Table 3. The wall time with Bi-Block is 18% to 53% of the one with PB for RWNV, and 16% to 60% for PRNV. The performance improvement is more significant on larger graphs such as UK, Kron29 and CrawlWeb, which are more than 60%. Next, we deeply compare the block I/Os and execution time of the two engines.

Block-I/O comparison. We first focus on the block I/O overhead of two engines. The block I/O number in the bi-block execution engine is only 34% to 59% of the one in the plain bucket model for RWNV, and 45% to 61% for PRNV, respectively. This is consistent with the theoretical analysis that the triangular bi-block scheduling strategy approximately cuts half of the block I/Os, according to Equation 3. Concretely, the reason for block I/O reduction is twofold. First, during each time slot, half of the ancillary block loading whose ID is less than the current block is saved. Second, the current block loading stops on b_{N_b-2}, so one block loading for the current block is saved in each time slot. Another observation is that when processing large graphs except for LJ, the block I/O time of the bi-block engine is reduced to 16% to 40% on both tasks, which is less than the reduction factor (i.e., 50%) of block I/O number. One reason is that some expensive random I/Os of loading blocks in the plain bucket engine are converted to sequential block I/Os in the bi-block execution engine during the block scheduling. In the plain bucket engine, after loading the current block, the loading of ancillary blocks starts from b_0, which incurs a random block I/O, while in GraSorw, the loading starts from the next block to the current block, which is sequential.

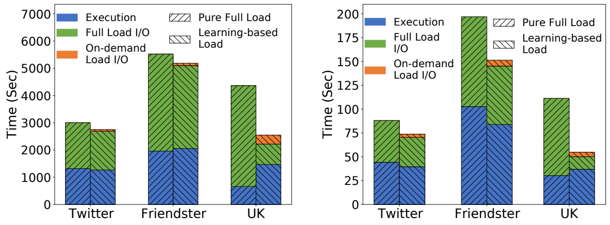
Execution time comparison. The execution time of the bi-block execution engine also decreases compared to the one of the plain bucket execution engine. For example, on UK, the execution time of the bi-block execution engine decreases to 18% for RWNV and 30% for PRNV, and on CrawlWeb, such decrease reaches 7% and 4% for RWNV and PRNV, respectively. One reason is that the bi-block execution engine reduces the thread management overhead by reducing the number of block I/Os in a time slot. Concretely, in GraSorw the current walks are executed in parallel, and the system needs to manage the threads, like the initiating, destroying. In both two engines, each loading of the ancillary block corresponds to a bucket execution. In the bi-block execution engine, the number of bucket execution is only half of that in the plain bucket engine, since the number of ancillary block I/Os has been reduced to around 50% as discussed above. Therefore, the initiating and destroying overhead in thread management is decreased, leading to a decrease of total execution time.

6.4 The Effectiveness of Learning-based Block Loading

In this experiment, we first describe the efficiency of GraSorw when using the learning-based block loading model, then analyze the improvement of I/O utilization.

Table 4: The performance of different loading methods with different partitions for the RWNV task.

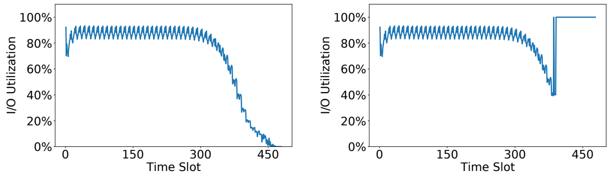
Graph	Partition	Pure Full Load				Learning-based Load					
		Wall Time(s)	Execution Time(s)	Block I/O Time(s)	Block I/O Num	Wall Time(s)	Execution Time(s)	Block I/O Time(s)	Block I/O Num	On-demand I/O Time(s)	On-demand I/O Num
TW	**Seq**	3434	1317	1689	9936	3229	1266	1419	8224	61	1714
TW	**METIS**	2829	1039	1541	7540	2465	1053	1056	5145	96	2168
UK	**Seq**	4798	662	3705	13587	2992	1467	749	2650	332	10628
UK	**METIS**	1856	98	1044	3751	1165	166	294	998	38	2558



(a) RWNV

(b) PRNV

Figure 7: Performance of the learning-based block loading model on various datasets.



(a) Full block loading.

(b) Learning-based block loading.

Figure 8: I/O-utilization with different block loading on Twitter.

Efficiency. The overall performance of using the learning-based block loading model is shown in Figure 7. The execution time is the time cost of updating walks. Under pure full load mode, the execution time does not include any I/O costs, while using the learning-based block loading method, the execution time includes some I/O costs which is incurred by the on-demand block loading method because the on-demand block loading method might bring in random vertex I/O during walk processing to get the vertex information that has not been loaded at the beginning. Therefore, in most of the results, the execution time of the learning-based method is longer than that of the full-load mode. For example, in Figure 7(a), the execution time increases by 805 seconds in graph UK compared to the pure full-load mode. However, the total time by using the learning-based block loading model is less because of the reduction of block I/Os (see results of sequential partition on UK in Table 4). To be concrete, the difference between block I/O time in pure full load mode and the sum of block I/O time and on-demand load I/O time in learning-based block loading model is 2624 seconds (see results of sequential partition on UK in Table 4), which is much greater than the increase of execution time. Such trade-off is leveraged by the learning-based model described in Section 5.

I/O-Utilization. We then discuss the I/O utilization with the two block loading methods. We take the I/O utilization of a specific block (e.g., block 10) in Twitter when it is loaded as an ancillary block as an example, and the results of other blocks in Twitter or other graphs are similar. Figure 8 shows the results under the pure full-load and learning-based block loading model. The I/O utilization is tallied after the execution of the corresponding bucket, and the x-axis represents the time slot of each block loading. The block I/O-utilization remains stable around 0.87 in the first 300 loads, we call this part the *plateau.* After the plateau, the block I/O-utilization under pure full-load mode decreases close to 0. In this period, many walks reach the termination condition, and less information is required to update the few remaining walks. Since the pure full-load method still loads the whole block into memory, it suffers from low I/O utilization. Our learning-based block loading model is aware of the decreasing of update walks and is able to switch to on-demand loading mode. The on-demand loading ensures 100% of I/O utilization, and only a small portion of the block is loaded into memory.

6.5 The Influence of Different Partitions

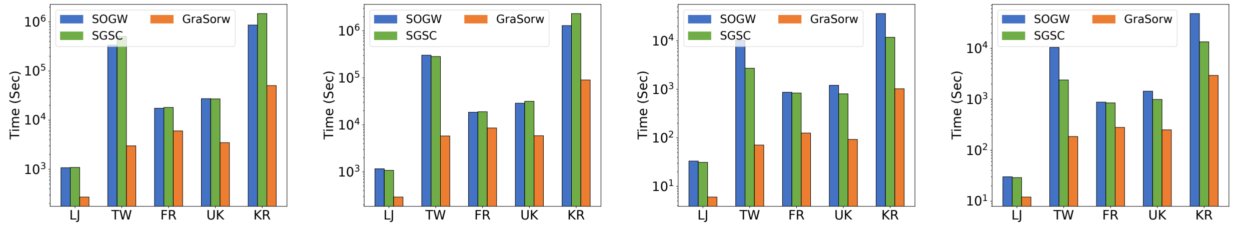
Different graph partition methods influence the overall performance. We compare the performance of GraSorw under METIS partition and the default sequential partition with the RWNV task. Since METIS fails to partition graph FR and Kron29 in memory on our server, we only evaluate the performance of METIS partition on Twitter and UK, and the results of running RWNV task are shown in Table 4. The partitioned block number is the same as that in sequential partitioning, and we ensure that the size of each block is roughly the same, where the biggest block is not larger than 1.03× the smallest one. We use the default k-way partition algorithm provided by METIS. For graph UK, the edge-cut of the METIS graph partition decreases significantly to 0.33%. For graph Twitter, the edge-cut is 55.14%.

The block I/O number of METIS partition is reduced compared to that in sequential partition for both loading methods. Specifically, for UK, the number of block I/O dropped by 72% under METIS partition in pure full load mode and 63% in the learning-based load mode. This is because the density of blocks is increased, and walks are more likely to update inside the block than moving out of the block. Consequently, walks are able to move more steps forward during a time slot, and the block I/O number is reduced. Furthermore, according to the discussion in Section 6.3, the decrease of block I/Os leads to the improvement of execution time as well.

Another observation is that METIS partition improves the efficiency of GraSorw when using the learning-based block loading model. Under sequential partition, the learning-based block loading model reduces 6% of wall time in Twitter compared to the pure full load method, while under METIS partition, the reduction reaches 13%. This is caused by the decrease of edge-cut. For a graph partition with a lower edge-cut, most of the walks are able to reach the termination condition with fewer time slots because they tend to move forward inside the block. However, there are still a few walks that tend to jump between different blocks, thus causing lots of block I/Os that have low I/O utilization under pure full-load mode. With the learning-based block loading model, these block I/Os can

Table 5: Results of First-order random walk execution.

Dataset	GraphWalker			GraSorw-No-LBL			GraSorw		
	Wall Time (s)	Execution Time (s)	Block I/O Time (s)	Wall Time (s)	Execution Time (s)	Block I/O Time (s)	Wall Time (s)	Execution Time (s)	Block I/O Time (s)
LJ	137	**84**	53	**133**	86	48	135	88	**47**
TW	1366	851	515	1399	871	**528**	**1302**	**793**	**509**
FR	**2122**	**1313**	809	2200	1362	838	2128	1346	782
UK	2242	1463	779	1867	1189	677	**1782**	**1123**	**660**



(a) RWNV ($p = 4, q = 0.25$)

(b) RWNV ($p = 0.25, q = 4$)

(c) PRNV ($p = 4, q = 0.25$)

(d) PRNV ($p = 0.25, q = 4$)

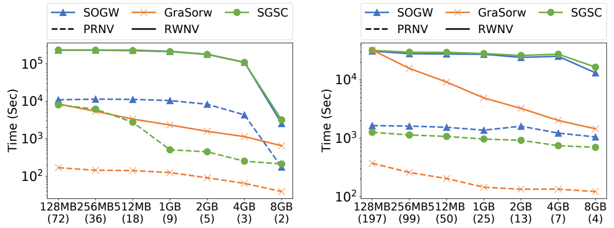
Figure 9: Wall time (log-scale) of RWNV and PRNV under different walk distributions.

be completed by the on-demand loading method, thus increasing the I/O utilization and improving efficiency.

6.6 Parameter Sensitivity

6.6.1 Random Walk Distribution. To study the performance of GraSorw under different walk distributions, we add sensitivity experiments with different p, q of Node2vec, and the results of wall time are shown in Figure 9. We can see that GraSorw achieves the least wall time (i.e., best efficiency) in all cases. For RWNV, since all of the vertices are activated, the static cache strategy in SGSC does not bring in significant improvement. In fact, on Kron29, the wall time of SGSC is much longer, that is 1.7× and 1.8× that of SOGW under the two random walk distributions, respectively. On the other hand, GraSorw is able to handle this situation, and the wall time is only 6% that of SOGW on Kron29 when $p = 4, q = 0.25$ and 7% when $p = 0.25, q = 4$. For tasks such as PRNV in which only a few vertices are initially activated, SGSC saves more time compared to that of RWNV. However, GraSorw still achieves the best result. For instance, when executing PRNV ($p = 4, q = 0.25$) on Twitter, the wall time of SGSC is 26% that of SOGW, while GraSorw is only 0.7%. Moreover, on Friendster, SGSC only saves sightly 3%~4% of wall time of SOGW, the time saved by GraSorw still achieves 86% for PRNV ($p = 4, q = 0.25$) and 68% for PRNV ($p = 0.25, q = 4$).

6.6.2 Block Size. We also study the performance of GraSorw with a variation of block size and the number of blocks, and the results on two representative graphs are shown in Figure 10. More results are reported in our technical report [22]. Similar to previous tasks, there is no significant difference between SOGW and SGSC for RWNV, and for PRNV, the latter is only slightly faster. On the other hand, GraSorw consistently outperforms the baselines. The only exception happens when conducting RWNV on UK with the block size set to 128MB. This is because when block size is small, the number of partitioned blocks of large graphs is big (197 in this case), and there are many ancillary block I/Os in GraSorw. Another observation is that as the block size increases, the advantage of GraSorw becomes more and more obvious. For example, on Twitter, the maximum performance improvement by GraSorw is achieved when the block size is set to 2GB, which is 114× and 91× speedups for RWNV and PRNV, respectively, compared to SOGW. The third observation is that when the block size reaches to 8GB, the difference between GraSorw and two baselines SOGW and SGSC becomes less obvious. For example, on Twitter, the wall time of SOGW is about 97× slower than the one of GraSorw when block size is 4GB (3 blocks in total), while it decreases to 4× slower when block size is enlarged to 8GB (2 blocks in total). This is because as the number of blocks decreases, walks can move more steps inside one block, and the number of walks crossing blocks becomes small, resulting in fewer individual vertex I/Os. In summary, GraSorw achieves significant time reduction under different block sizes and the number of blocks.



(a) Twitter

(b) UK200705

Figure 10: Wall time (log-scale) of RWNV and PRNV under different block sizes. In x-axis, the number in parentheses represents the number of partitioned blocks.

6.7 The Performance of First-order Random Walk in GraSorw

Finally, we evaluate the performance of first-order random walk in GraSorw to demonstrate the applicability of our system. We take DeepWalk as the benchmark of first-order random walk tasks and compare GraSorw with the state-of-the-art disk-based first-order random walk system GraphWalker. We also compare the results of GraSorw without learning-based block loading (GraSorw-No-LBL), which uses the Iteration-based method mentioned in Section 4.1 to schedule the current blocks. The experimental results are shown in Table 5. We see that GraSorw or GraSorw-No-LBL achieves the best efficiency on three datasets, i.e., LJ, TW and UK, and is comparable to GraphWalker on FR. Compared with GraphWalker, GraSorw saves 21%, 23% and 16% of the wall time, execution time and block I/O time on UK, respectively. With the help of learning-based block loading method, the time cost of block I/Os in GraSorw is the least among all these four datasets, and it is 85%~99% of the one in GraphWalker. This is because some heavy I/Os of loading the complete block is converted to light vertex I/Os when the number of remained walks is small. In addition, comparing GraSorw-No-LBL with GraphWalker, both the execution time and block I/O time are similar. This demonstrates that the iterative block scheduling is effective for first-order random walks, and it is feasible to replace the state-aware block scheduling mechanism in GraphWalker with the iterative block scheduling.

7 RELATED WORK

Many systems have been designed to analyze large graphs in recent years. Some studies focused on how to migrate the benefits of distributed computing to graph processing. Pregel [30] proposes a synchronization model that represents various typical graph processing tasks as a series of iterations to run them on a cluster of machines. GraphLab [26] proposes a model for asynchronous processing, and PowerGraph [14] takes into account the power-law property of natural graphs for faster access to vertex information. However, distributed systems have high requirements for the running environment, which is expensive, and have high communication costs between nodes.

As another solution that is lightweight, inexpensive, and scalable, many single machine disk-based graph processing systems have been proposed. GraphChi [20] first organizes the graph data on disk in shards, thus converting random I/Os to sequential I/Os in each shard. X-Stream [37] employs a new edge-centric graph computation model that enables the system to stream the list of edges read from disk directly. GridGraph [51] designs a more clever fine-grained subgraph partitioning to avoid loading useless information into memory and accelerate I/O processing. In addition, the features of SSD are also considered by some systems. Liu et al. [24] designed a disk-based full-granularity I/O management by reorganizing the SSD format to store graph data completely on disk, which makes the performance of dedicated SSD-based graph processing systems closer to that of the memory-based graph processing systems. Due to the generality, these aforementioned systems did not take into consideration the features of random walks and entail more time for random walk processing.

Meanwhile, due to the wide applicability of random walk, several dedicated systems or frameworks have been proposed to accelerate random walk processing. Most of them are designed in-core. Shao et al. [40] proposed a framework for rational use of available memory, which switches between different sampling algorithms for different nodes to balance the time and space overheads. ThunderRW [41] designs a step-centric programming model to address the high CPU pipeline slots stalled due to irregular memory access in random walk tasks. UniNet [50] brings in a new edge sampler based on Metropolis-Hastings to efficiently sample the next steps of random walks and proposes a framework that provides a uniform representation of different random walk models and allows users to implement new graph representation learning models flexibly. However, these frameworks use a memory-based model and cannot provide help in the scalability of the large graph. KnightKing [49] is a distributed system aiming to optimize random walk processing and employs an efficient algorithm in second-order random walk sampling. There are also systems focusing on out-of-core random walk processing. DrunkardMob [19] encodes each walk and stores them in memory to support parallelism for billions of random walks. In a single block, it clusters vertices into batches and manages walks belonging to the batch together in the corresponding bucket. GraphWalker [19] adopts block-centric walk management and also uses a disk to store walks. It proposes state-aware block scheduling and asynchronous walk updating to reduce block I/Os. Different from these systems, GraSorw focuses on optimizing a large amount of random vertex I/Os in second-order random walk tasks, and converts these I/Os into sequential by employing bi-block execution engine and increases the I/O utilization with the help of learning-based block loading model.

8 CONCLUSION

Second-order random walk is an important method for modeling higher-order dependencies in data. The existing disk-based graph system cannot efficiently support the second-order random walk. We proposed an I/O-efficient disk-based second-order random walk system. To reduce the massive light vertex I/Os, we developed a bi-block execution engine with a triangular bi-block scheduling strategy, which smartly converts small random I/Os into large sequential I/Os. To improve the I/O-utilization, we introduced a learning-based block loading model to select the proper block loading method automatically. Finally, we empirically evaluated our system on five large graphs, and the results demonstrated GraSorw significantly surpasses the existing disk-based random walk systems. Furthermore, considering that the processing of second-order random walks in most of these applications is an independent phase, GraSorw can be easily embedded or integrated into existing second-order random walk-based applications. On the other hand, the techniques proposed in GraSorw is also promising in optimizing graph sampling functions in the deep learning computing frameworks, such as MindSpore[1], and we treat this research as our future work.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (Nos. U1936104, 61832001, 62192784), CAAI-Huawei MindSpore Open Fund, and CCF-Tencent Open Fund. Lei Chen's work is partially supported by National Key Research & Development Program of China Grant No. 2018AAA0101100, the Hong Kong RGC GRF Project 16202218, CRF Project C6030-18G, C1031-18G, C5026-18G, AOE Project AoE/E-603/18, RIF Project R6020-19, Theme-based project TRS T41-603/20R, China NSFC No. 61729201, Guangdong Basic and Applied Basic Research Foundation 2019B151530001, Hong Kong ITC ITF grants ITS/044/18FX and ITS/470/18FX, Microsoft Research Asia Collaborative Research Grant, HKUST-NAVER/LINE AI Lab, Didi-HKUST joint research lab, HKUST-Webank joint research lab grants and HKUST Global Strategic Partnership Fund (2021 SJTU-HKUST).

[1] https://www.mindspore.cn/

REFERENCES

[1] April 17, 2022. *Crawlweb*. http://webdatacommons.org/hyperlinkgraph/index.html

[2] April 17, 2022. *Friendster*. https://snap.stanford.edu/data/com-Friendster.html

[3] April 17, 2022. *Graph500*. https://graph500.org/

[4] April 17, 2022. *LiveJournal*. https://snap.stanford.edu/data/soc-LiveJournal1.html

[5] April 17, 2022. *Twitter*. https://old.datahub.io/dataset/twitter-social-graph-www2010

[6] April 17, 2022. *UK200705*. http://law.di.unimi.it/webdata/uk-2007-05/

[7] Ziv Bar-Yossef, Alexander Berg, Steve Chien, Jittat Fakcharoenphop, and Dror Weitz. 2000. Approximating aggregate queries about web pages via random walks. In *Proceedings of the 26th International Conference on Very Large Data Bases*. 535–544.

[8] Renátó Besenczi, Norbert Bátfai, Péter Jeszenszky, Roland Major, Fanny Monori, and Márton Ispány. 2021. Large-scale simulation of traffic flow using Markov model. *Plos one* 16, 2 (2021), e0246062.

[9] Paolo Boldi and Marco Rosa. 2012. Arc-community detection via triangular random walks. In *Proceedings of 2012 8th Latin American Web Congress*. 48–56.

[10] Xiaoheng Deng, Genghao Li, Mianxiong Dong, and Kaoru Ota. 2017. Finding overlapping communities based on Markov chain and link clustering. *Peer-to-Peer Networking and Applications* 10, 2 (2017), 411–420.

[11] Yuxiao Dong, Nitesh V Chawla, and Ananthram Swami. 2017. metapath2vec: Scalable representation learning for heterogeneous networks. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 135–144.

[12] Tim S Evans and Renaud Lambiotte. 2009. Line graphs, link partitions, and overlapping communities. *Physical Review E* 80, 1 (2009), 016105.

[13] 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.

[14] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX symposium on operating systems design and implementation*. 17–30.

[15] 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*. 855–864.

[16] Monika R. Henzinger, Allan Heydon, Michael Mitzenmacher, and Marc Najork. 1999. Measuring index quality using random walks on the Web. *Computer Networks* 31, 11 (1999), 1291–1303.

[17] 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.

[18] Sanjeev Kumar. 1998. The PageRank Citation Ranking: Bringing Order to the Web. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization*. 111–112.

[19] Aapo Kyrola. 2013. DrunkardMob: Billions of random walks on just a PC. In *Proceedings of the 7th ACM Conference on Recommender Systems*. 257–264.

[20] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*. 31–46.

[21] Amy N Langville and Carl D Meyer. 2011. Google's PageRank and beyond. In *Google's PageRank and Beyond*. Princeton university press.

[22] 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. *arXiv preprint arXiv:2203.16123* (2022).

[23] Xueting Liao, Yubao Wu, and Xiaojun Cao. 2019. Second-Order CoSimRank for Similarity Measures in Social Networks. In *2019 IEEE International Conference on Communications*. 1–6.

[24] Hang Liu and H. Howie Huang. 2017. Graphene: Fine-grained IO management for graph computing. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies*. 285–299.

[25] Qin Liu, Zhenguo Li, John C.S. Lui, and Jiefeng Cheng. 2016. PowerWalk: Scalable personalized pagerank via random walks with vertex-centric decomposition. In *Proceedings of International Conference on Information and Knowledge Management*. 195–204.

[26] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A framework for machine learning and data mining in the cloud. In *Proceedings of the VLDB Endowment*. 716–727.

[27] Meng Ma, Weilan Lin, Disheng Pan, and Ping Wang. 2021. ServiceRank: Root Cause Identification of Anomaly in Large-Scale Microservice Architecture. *IEEE Transactions on Dependable and Secure Computing* 5971, c (2021), 1–15.

[28] 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 12th European Conference on Computer Systems*. 527–543.

[29] David Maier. 1978. The Complexity of Some Problems on Subsequences and Supersequences. *J. ACM* 25, 2 (1978), 322–336.

[30] Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 135–145.

[31] Kang Ning and Hon Wai Leong. 2006. Towards a Better Solution to the Shortest Common Supersequence Problem: A Post. In *Computer and Computational Sciences, International Multi-Symposiums on*. 84–90.

[32] Yun Peng, Byron Choi, and Jianliang Xu. 2021. Graph Learning for Combinatorial Optimization: A Survey of State-of-the-Art. *Data Science and Engineering* 6, 2 (2021), 119–141.

[33] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online learning of social representations. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 701–710.

[34] Sergio Porta, Paolo Crucitti, and Vito Latora. 2006. The network analysis of urban streets: A dual approach. *Physica A: Statistical Mechanics and its Applications* 369, 2 (2006), 853–866.

[35] Tahleen Rahman, Bartlomiej Surma, Michael Backes, and Yang Zhang. 2019. FairWalk: Towards fair graph embedding. In *Proceedings of the International Joint Conference on Artificial Intelligence*. 3289–3295.

[36] Sascha Rothe and Hinrich Schütze. 2014. CoSimRank: A flexible & efficient graph-theoretic similarity measure. *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, 1392–1402.

[37] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. 472–488.

[38] Kari-Jouko Räihä and Esko Ukkonen. 1981. The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science* 16, 2 (1981), 187–198.

[39] Yingxia Shao, Shiyue Huang, Yawen Li, Xupeng Miao, Bin Cui, and Lei Chen. 2021. Memory-aware framework for fast and scalable second-order random walk over billion-edge natural graphs. *VLDB Journal* 30, 5 (2021), 769–797.

[40] 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 ACM SIGMOD International Conference on Management of Data*. 1797–1812.

[41] Shixuan Sun, Yuhang Chen, Shengliang Lu, Bingsheng He, and Yuchen Li. 2021. ThunderRW: An In-Memory Graph Random Walk Engine. In *Proceedings of the VLDB Endowment*. 1992–2005.

[42] Anton Tsitsulin, Davide Mottin, Panagiotis Karras, and Emmanuel Müller. 2018. VERSE: Versatile graph embeddings from similarity measures. In *Proceedings of the World Wide Web Conference*. 539–548.

[43] Jonathan S. Turner. 1989. Approximation algorithms for the shortest common superstring problem. *Information and Computation* 83, 1 (1989), 1–20.

[44] Keval Vora, Guoqing Xu, and Rajiv Gupta. 2016. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *Proceedings of the 2016 USENIX Annual Technical Conference*. 507–522.

[45] Ping Wang, Jingmin Xu, Meng Ma, Weilan Lin, Disheng Pan, Yuan Wang, and Pengfei Chen. 2018. CloudRanger: Root cause identification for cloud native systems. In *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 492–502.

[46] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John C.S. Lui. 2020. GraphWalker: An I/O-efficient and resource-friendly graph analytic system for fast and scalable random walks. In *Proceedings of the 2020 USENIX Annual Technical Conference*. 559–571.

[47] Yubao Wu, Yuchen Bian, and Xiang Zhang. 2016. Remember where you came from: On the second-order random walk based proximity measures. In *Proceedings of the VLDB Endowment*. 13–24.

[48] Jianye Yang, Wu Yao, and Wenjie Zhang. 2021. Keyword Search on Large Graphs: A Survey. *Data Science and Engineering* 6, 2 (2021), 142–162.

[49] Ke Yang, Ming Xing 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.

[50] Xingyu Yao, Yingxia Shao, Bin Cui, and Lei Chen. 2021. UniNet: Scalable network representation learning with metropolis-hastings sampling. In *Proceedings of International Conference on Data Engineering*. 516–527.

[51] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the 2015 USENIX Annual Technical Conference*. 375–386.