Toward Fast and Scalable Random Walks over Disk-Resident Graphs via Efficient I/O Management

RUI WANG and YONGKUN LI, University of Science and Technology of China
YINLONG XU, University of Science and Technology of China
HONG XIE, Chongqing University
JOHN C. S. LUI, The Chinese University of Hong Kong
SHUIBING HE, Zhejiang University

Traditional graph systems mainly use the iteration-based model, which iteratively loads graph blocks into memory for analysis so as to reduce random I/Os. However, this iteration-based model limits the efficiency and scalability of running random walk, which is a fundamental technique to analyze large graphs. In this article, we first propose a state-aware I/O model to improve the I/O efficiency of running random walk, then we develop a block-centric indexing and buffering scheme for managing walk data, and leverage an asynchronous walk updating strategy to improve random walk efficiency. We implement an I/O-efficient graph system, GraphWalker, which is efficient to handle very large disk-resident graphs and also scalable to run tens of billions of random walks with only a single commodity machine. Experiments show that GraphWalker can achieve more than an order of magnitude speedup when compared with DrunkardMob, which is tailored for random walks based on the classical graph system GraphChi, as well as two state-of-the-art single-machine graph systems, Graphene and GraFSoft. Furthermore, when compared with the most recent distributed system KnightKing, GraphWalker still achieves comparable performance with only a single machine, thereby making it a more cost-effective alternative.

CCS Concepts: • **Information systems** → **Online analytical processing engines**; **Storage management**; *Information systems applications;*

Additional Key Words and Phrases: Graph processing system, random walk, graph storage

The work of Y. Li was supported in part by the National Key R&D Program of China (2018YFB1800203), Youth Innovation Promotion Association CAS. The work of J. C. S. Lui was supported in part by the GRF (14200420). The work of S. He was supported in part by the National Science Foundation of China (62172361).
Authors' addresses: R. Wang and Y. Li (corresponding author), University of Science and Technology of China, 96 Jinzhai Road, Baohe District, Hefei, The People's Republic of China; emails: rwang067@mail.ustc.edu.cn, ykli@ustc.edu.cn; H. Xie, Chongqing University, No. 174 Shazhengjie, Shapingba, Chongqing, The People's Republic of China; email: xiehong2018@cqu.edu.cn; Y. Xu, Anhui Province Key Laboratory of High Performance Computing, University of Science and Technology of China, 96 Jinzhai Road, Baohe District, Hefei, The People's Republic of China; email: ylxu@ustc.edu.cn; J. C. S. Lui, The Chinese University of Hong Kong, Shatin, NT, Hong Kong SAR, The People's Republic of China; email: cslui@cse.cuhk.edu.hk; S. He, Zhejiang University, 38 Zheda Road, West Lake District, Hangzhou, The People's Republic of China; email: heshuibing@zju.edu.cn.

1553-3077/2022/11-ART36 $15.00
https://doi.org/10.1145/3533579

ACM Reference format:
Rui Wang, Yongkun Li, Yinlong Xu, Hong Xie, John C. S. Lui, and Shuibing He. 2022. Toward Fast and Scalable Random Walks over Disk-Resident Graphs via Efficient I/O Management. *ACM Trans. Storage* 18, 4, Article 36 (November 2022), 33 pages.
https://doi.org/10.1145/3533579

1 INTRODUCTION

Graph analysis has received a lot of attention in recent years due to its wide use in many applications, e.g., page ranking [26, 43], social networks [11, 28], recommendation systems [16, 38], and navigation systems [7, 13]. As the size of graphs increases continuously, e.g., many web graphs already have billions of vertices and hundreds of billions of edges [3], it is difficult to make these large graphs fit in a single machine's memory. Instead, the whole graph has to be stored on disk, so analyzing this type of disk-resident large graph is quite time consuming because of the massive random I/Os. For some applications, it even costs several hours or days to run to have any result.

To improve the performance of analyzing large graphs on a single machine, many out-of-core graph processing systems are proposed, such as GraphChi [32], X-Stream [48], GridGraph [62], FlashGraph [12], ODS [54], CLIP [6], Mosaic [40], Graphene [37], GraFBoost [27], V-Part [14], and LUMOS [53]. One major effort of these systems is to reduce random disk I/Os. Generally, when a graph is too large to fit into the memory, these systems partition the entire graph into many subgraphs, and store each subgraph as a block on disk, e.g., *shard* in GraphChi [32]. To carry graph analysis, they adopt an *iteration-based model*. In each iteration, blocks are sequentially loaded into memory, then analysis related to the loaded subgraph is performed. This way, it turns massive random I/Os into a series of sequential I/Os and guarantees synchronized analysis over all blocks in each iteration.

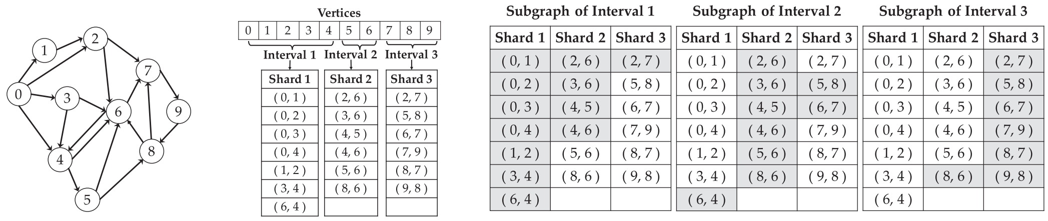
Random walk has been proven to be efficient to analyze large graphs [8, 15, 21, 25, 31, 34, 35, 47, 49], so it is widely used in various graph algorithms and applications. For example, **Personalized PageRank (PPR)** [15, 31] starts thousands of walks from the source vertex to compute visit frequencies in order to approximate PageRank values. **SimRank (SR)** [25] computes the similarity for a vertex pair by first starting many random walks from each of the vertex pairs and then computing the expected meeting time. **Random walk domination (RWD)** [35] starts walks from all vertices to measure the influence diffusion over the whole graph. To compute PPR for all vertices, and all-pair similarity, it is also required to start random walks from every vertex, which results in massive concurrent walks.

We observe that current graph systems with the iteration-based model cannot efficiently support random walks. The major limitations are threefold. First, due to the high randomness nature, many walks are unevenly scattered at different parts of the graph after several steps, so some subgraphs may contain only few walks. However, the iteration-based model is unaware of the status of walks, e.g., regardless of which vertices the walks currently stay, it just sequentially loads all needed subgraphs into memory for analysis, and this may result in very low I/O efficiency, especially when the processed subgraphs contain only few walks. Second, as the iteration-based model ensures a synchronized analysis, all walks move exactly one step in each iteration, and in each iteration, all needed subgraphs are successively loaded. Some recently loaded subgraphs will be evicted out even if they may be useful to update more walks, and this also incurs more I/Os. As a result, the walk updating efficiency is also limited and thus further exacerbates the I/O efficiency. This is true especially for applications demanding long walks. Last, due to the randomness of walks, the number of walks at each vertex varies dynamically, so existing graph systems usually use massive dynamic arrays to record the walks currently traveling through each edge or each vertex in the graph. However, this indexing design requires large memory space and thus limits the scalability of handling very large graphs.

Various design efforts have been made in recent years to improve the I/O efficiency of the iteration-based model. For example, DynamicShards [54] and Graphene [37] dynamically adjust the layout of graph blocks to reduce the loading of useless data in each iteration. CLIP [6] proposes the *re-entry scheme* and Lumos [53] proposes the *cross-iteration value propagation technique*, and both of them aim to make full use of the loaded blocks to avoid loading the corresponding graph portions in future iterations. These systems greatly improve performance, but they do not take into account the random walk features, and the preceding analyzed limitations are still not fully addressed. To efficiently support massive random walks, various design efforts have also been made in recent years. One typical system is DrunkardMob [31], which proposes several optimizations to reduce the memory usage of walk indexes so as to support a large amount of random walks. However, its scalability is still limited. For example, it costs 2.3 hours to run 1 billion random walks of 10 steps long on a medium-scale graph YahooWeb [5], and it is even unable to run random walks on very large graphs like **CrawlWeb (CW)** [3] due to its high memory consumption. KnightKing [58] is the most recent distributed graph system, which is also optimized for random walks. It provides a unified framework to support various random walks and mainly focuses on optimizing the walking process without addressing disk I/Os.

To address the I/O efficiency problem so as to efficiently support fast and scalable random walks, we first develop a graph storage engine tailored for random walks by utilizing a state-aware I/O model, then we develop an efficient computing framework by utilizing an asynchronous walk updating scheme to accelerate walk updates and a lightweight block-centric walk management scheme to improve memory efficiency. Finally, we realize an I/O-efficient graph system, GraphWalker. Note that GraphWalker is also a resource-friendly solution in the sense that it can process very large graphs (billions of nodes and edges) on a single commodity machine. In summary, our main contributions are as follows:

- We develop a novel *state-aware I/O model*, which leverages the states of each random walk to preferentially load the graph block with the most walks from disk into memory, so as to improve the I/O efficiency. To realize it, we first propose a *self-adaptive graph partitioning and loading strategy*, and also employ a probabilistic approach to balance the progress of each walk, so as to address the straggler problem and improve the I/O efficiency. In addition, we propose a walk-conscious caching scheme to improve the cache efficiency.
- We leverage an *asynchronous walk updating scheme* to allow each walk to move as many steps as possible until it reaches the boundary of the currently loaded subgraph in memory, so as to fully utilize the loaded subgraph and accelerate the progress of random walks. In addition, we develop a disk-based walk management scheme by designing a *lightweight block-centric indexing scheme* and a *fixed-length walk buffering strategy* to manage walk states, so as to support running massive random walks in parallel on huge graphs.
- We also leverage the state-aware I/O model and asynchronous walk updating scheme to accelerate the widely used random walks variants like **random walk with restart (RWR)** and **random walk with jump (RWJ)**. The idea is to utilize a joint-based optimization [15] to convert long walks to short walks so as to improve the computing efficiency.
- We implement a prototype GraphWalker and conduct extensive experiments to demonstrate its efficiency. Results show that GraphWalker can achieve more than *an order of magnitude speedup* compared with the random walk specific system DrunkardMob [31], as well as two state-of-the-art single-machine graph systems, Graphene [37] and GraFSoft [27]. GraphWalker is also scalable to run massive random walks on huge graphs, and its



Vertices

0	1	2	3	4	5	6	7	8	9

Interval 1 Interval 2 Interval 3

Shard 1	Shard 2	Shard 3
(0, 1)	(2, 6)	(2, 7)
(0, 2)	(3, 6)	(5, 8)
(0, 3)	(4, 5)	(6, 7)
(0, 4)	(4, 6)	(7, 9)
(1, 2)	(5, 6)	(8, 7)
(3, 4)	(8, 6)	(9, 8)
(6, 4)		

Subgraph of Interval 1			Subgraph of Interval 2			Subgraph of Interval 3		
Shard 1	Shard 2	Shard 3	Shard 1	Shard 2	Shard 3	Shard 1	Shard 2	Shard 3
(0, 1)	(2, 6)	(2, 7)	(0, 1)	(2, 6)	(2, 7)	(0, 1)	(2, 6)	(2, 7)
(0, 2)	(3, 6)	(5, 8)	(0, 2)	(3, 6)	(5, 8)	(0, 2)	(3, 6)	(5, 8)
(0, 3)	(4, 5)	(6, 7)	(0, 3)	(4, 5)	(6, 7)	(0, 3)	(4, 5)	(6, 7)
(0, 4)	(4, 6)	(7, 9)	(0, 4)	(4, 6)	(7, 9)	(0, 4)	(4, 6)	(7, 9)
(1, 2)	(5, 6)	(8, 7)	(1, 2)	(5, 6)	(8, 7)	(1, 2)	(5, 6)	(8, 7)
(3, 4)	(8, 6)	(9, 8)	(3, 4)	(8, 6)	(9, 8)	(3, 4)	(8, 6)	(9, 8)
(6, 4)			(6, 4)			(6, 4)		

(a) Example graph (b) Data organization in shards (c) Graph loading with parallel sliding window

Fig. 1. Storage and I/O model in GraphChi.

performance is comparable to that of KnightKing [58], which is the state-of-the-art distributed random walk system, running on a cluster of eight machines.

The rest of this article is organized as follows. In Section 2, we first introduce the iteration-based graph computation model and analyze its limitations for supporting random walks. In Sections 3 and 4, we present the design details of the state-aware graph storage and random walk computing framework, and in Section 5, we show the evaluation results. Finally, Section 6 reviews related work and Section 7 concludes.

2 BACKGROUND AND MOTIVATION

We first introduce the storage and computation process of the iteration-based model, then analyze its limitations for supporting random walks.

2.1 Iteration-Based Graph Computation

Graph sharding and subgraph loading. For simplicity, we take GraphChi [32], the pioneering single-machine iteration-based graph system, as an example to illustrate its key idea. We like to point out that this iteration-based model is quite representative and is widely used in many graph systems (e.g., [12, 14, 27, 37, 48, 53, 54, 62]). GraphChi splits all vertices into disjoint intervals and associates each interval with a *shard*, which stores all edges whose destination vertices lie in this interval. Edges in each shard are sorted according to their source vertices. For example, for the graph in Figure 1(a), its data organization in shards is illustrated in Figure 1(b).

Iteration-based computing. To perform analysis, GraphChi loads all subgraphs iteratively by using the parallel sliding window, which is illustrated in Figure 1(c). In each iteration, it loads the subgraphs in a round-robin order and guarantees synchronization between all computation tasks over the whole graph. Specifically, at each time slot, GraphChi loads one subgraph corresponding to one interval into memory for analysis. It first loads the in-edges from its corresponding shard, then loads the out-edges from other shards. As edges are sorted by source vertices in each shard, at most P sequential disk reads are needed to load the subgraph corresponding to one interval if there are P shards. Then GraphChi traverses the vertices of the loaded subgraph and conducts computation. This way, GraphChi transfers random accesses to a series of sequential accesses and greatly improves the performance of disk-resident graph processing.

2.2 Limitations in Supporting Random Walks

A random walk proceeds by starting at a source vertex, then repeats the process of randomly selecting a neighbor to visit. Many applications often need to simultaneously run massive random walks [15, 35, 55, 59], and it is also quite common to have multiple application tasks concurrently running on the same graph and thus induces massive parallel walks. When supporting massive parallel random walks, graph systems with the iteration-based model suffer from several limitations,

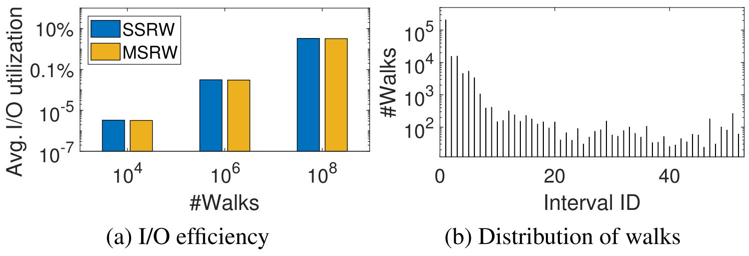


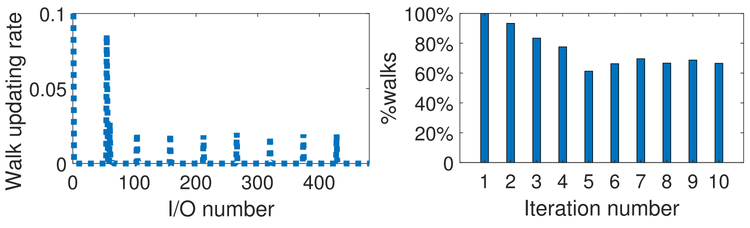
Fig. 2. I/O efficiency under different walk settings and distribution of the number of walks over graph blocks (intervals).

e.g., low I/O efficiency and low walk updating rate, as well as high memory cost for managing walks. In the following, we analyze these three limitations in detail.

Limitation 1: Low I/O efficiency. First of all, the iteration-based model leads to low I/O efficiency for random walks, which is defined as the number of edges used for updating walks divided by the number of edges loaded in one I/O, i.e., a subgraph loading. The main reason is that walks may be unevenly scattered across the entire graph after a few steps even if they started from the same source vertex, so the distribution of the number of walks in different graph blocks is highly skewed. As a result, even if there are only few walks in some blocks, they are still required to be loaded into memory, so it brings extremely low I/O efficiency. Some recent works, like DynamicShards [54] and Graphene [37], adopt an *on-demand* I/O strategy to dynamically adjust graph block layout and skip loading blocks that do not contain any walks so as to reduce the loading of useless edges, but the low I/O efficiency problem is still not fully addressed. As long as there is one walk in a block in one iteration, then this block still has to be loaded into memory for computation.

We also run experiments to demonstrate the skewed walk distribution and low I/O efficiency. We use DrunkardMob to run 10^4, 10^6, and 10^8 walks of length 10 on the Friendster graph consisting of 68.3 million vertices, and consider starting random walks from a single source (SSRW) or multiple random sources (MSRW). Please refer to Section 5.1 for a detailed experiment setting. Figure 2(a) shows the average I/O utilization, which is 3.1×10^{-6}, 3.2×10^{-4}, and 0.032 for SSRW with 10^4, 10^6, and 10^8 walks, respectively, and the results are similar for MSRW. We point out that the I/O efficiency is very low, especially for the case of small number of walks. Figure 2(b) further shows the distribution of number of walks in different blocks after four iterations when running 10^6 walks started from a single source. We find that walks are scattered in different blocks after only four steps, and the distribution is highly skewed. For example, the maximum number of walks in a block is 17,285× the minimum value. We also run the same experiments with Graphene, and the average I/O utilization is 6.1×10^{-3}, 3.1×10^{-3}, and 0.032 for MSRW when running 10^4, 10^6, and 10^8 walks, respectively. The results indicate that Graphene can greatly improve I/O efficiency when the number of walks is small, but the I/O efficiency is still limited when running massive random walks.

To address the limitation of low I/O efficiency, we propose a *walk-state-aware I/O model*, which loads graph blocks by considering the current states of random walks. Precisely, it always chooses to load the block with the maximum number of walks so as to make more walks get updated for each single I/O of loading a subgraph. To realize the preceding model, GraphWalker develops multiple optimizations to further improve I/O efficiency, including lightweight graph data and walk data management, fine-grained graph loading, selective block caching, and asynchronous walk updating. With these techniques, our experiment results show that GraphWalker achieves 2× to 4× I/O efficiency improvement (see Section 5.4.1).



(a) Walk updating rate (b) Walks in the first interval

Fig. 3. Walk updating rate and the fraction of walks that still remain in the first block in each iteration.

Limitation 2: Low walk updating rate. The iteration-based model also leads to a low walk updating rate, which is defined as the sum of walked steps of all walks in the loaded subgraph divided by the total steps needed to walk. This is because with the iteration-based model, each walk can only move one step in each iteration in a synchronized pace, which severely wastes the data in memory, as many walks can still make more moves over the loaded subgraph. To demonstrate, we run 10^6 random walks started from a single vertex by using the same setting as earlier. Figure 3(a) shows the walk updating rate. We find that all walks together move only 1K steps on average in one I/O, except for the first one, but we have a total of 10^9 steps to walk, so the updating rate is as low as 10^{-6}. We also observe similar results for Graphene. We further count the fraction of walks that still remain in the first block in each iteration as shown in Figure 3(b). We find that on average, 75.3% of walks still remain in the first block after one step, and they could move more steps in the current iteration, so it results in the low walk updating rate. Recently, CLIP [6] proposed a *re-entry* method and Lumos [53] proposed the *cross-iteration value propagation* technique to reuse the loaded data to improve the I/O and computing efficiency. However, when supporting random walks, the re-entry method would bring extra cost, as it needs to access the whole subgraph multiple times, and it is also unable to determine the same number of times of re-entries for all walks, because the progress of walks in a subgraph block may be quite different.

To address the limitation of low walk updating rate, we propose an *asynchronous walk updating scheme* based on the *re-entry* technique to allow random walks to move as many steps as possible within the currently loaded subgraph. This scheme relaxes the strong synchronization of all walks, which is not necessary for random walk based algorithms. With our asynchronous walk updating scheme, GraphWalker greatly increases the walk updating rate and reduces the completion time of all random walks. We also develop a probabilistic approach to balance walk progress so as to address the global straggler issue.

Limitation 3: High memory cost for managing walk data. Since the number of walks at each vertex is dynamic and unpredictable, walks are usually stored with massive dynamic arrays. For example, GraphChi associates each edge with a dynamic array to store the walks currently traveling through the edge. This design incurs high memory cost. For example, it needs at least 26.4 GB of space to store only the walk array indexes, not including the walk states information, for a medium-scale graph like YahooWeb [5], which has 1.4 billion vertices and 6.6 billion edges. Some systems use a vertex-centric way to manage walks [6, 12, 37], but it also incurs high memory cost, e.g., 5.6 GB to store the walk array indexes for YahooWeb.

DrunkardMob encodes the states of a walk into a 32-bit or 64-bit representation and puts walks of the adjacent 128 vertices into the same walk buffer to reduce the total size of walk indexes, as shown in Figure 4. It reduces the size of walk array indexes to 1/128 of that of the vertex-centric management, e.g., only 44.8 MB for YahooWeb. However, each walk buffer in DrunkardMob is also managed with a dynamic array, so it still suffers from the scalability problem. First, as it creates too

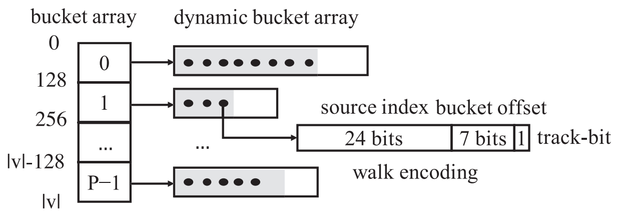


Fig. 4. Walk index management in DrunkardMob.

many dynamic arrays for large graphs, e.g., 11.2 million for YahooWeb, it causes frequent memory re-allocation, which not only introduces memory fragmentation but also brings extra time cost and limits the graph scale that could be analyzed. Second, DrunkardMob keeps all walks in memory, so the number of walks is limited by memory space, e.g., 10 billion walks cost at least 40 GB of memory. In addition, it incurs high cost to flush walk indexes to disk, as they are related to too many files. Thus, the scale of random walks that could be analyzed by DrunkardMob is also limited.

To address the limitation of high memory cost for managing walk data, we first adopt a block-centric method to manage walk data, so as to greatly reduce the size of walk array indexes. We also use fixed-length buffers to cache walks in each subgraph block, so as to avoid frequent memory re-allocation. With our lightweight walk data management scheme, both the scale of graphs and the number of walks that can be processed are no longer limited by memory capacity.

3 STATE-AWARE GRAPH STORAGE

In this section, we present the design details of the state-aware graph storage to improve the I/O efficiency of random walk over disk-resident graphs. We first introduce the main idea of state-aware storage, which adaptively schedules the graph loading I/Os by selecting a graph block to load according to the states of walks, such as the vertex IDs at which the walkers currently stay and the number of steps the walks have finished. Then we present the details of its key design techniques, including self-adaptive graph partitioning, state-aware graph loading, and walk-conscious graph caching.

3.1 Main Idea

We target supporting not only a very large number of walks, say tens of billions of walks, but also very long walks, say thousands of steps for each walk. To achieve this goal, the main idea is to adopt a *state-aware model* that leverages the states of each walk, e.g., the current vertex at which the walk stays. Briefly speaking, unlike the iteration-based model that blindly loads graph blocks sequentially, the state-aware model chooses to load the graph block containing the largest number of walks, and makes each walk move as many steps as possible until it reaches the boundary of the loaded subgraph. By doing this, walks can get updated as much as possible within each I/O. As a result, both the low I/O efficiency and low walk updating rate problems can be efficiently addressed.

To further illustrate the preceding idea and analyze its benefits, we still consider the example graph in Figure 1(a). Suppose we have to run three random walks that start at vertex 0 and have to move four steps. Figure 5 shows the process of graph loading and walk updating with the state-aware model. Specifically, in the first I/O, graph block b_0 is loaded into memory as it contains all three walks. With the loaded graph block b_0, walk w_0 and w_1 move two steps, and w_2 moves only one step as it requires other graph blocks that are not in memory for walking more steps. As two walks fall into block b_2, in the second I/O, block b_2 is loaded into memory, and walk w_0 finishes and w_1 can move one step. Finally, both of the remaining two walks are in block b_1, so we load b_1

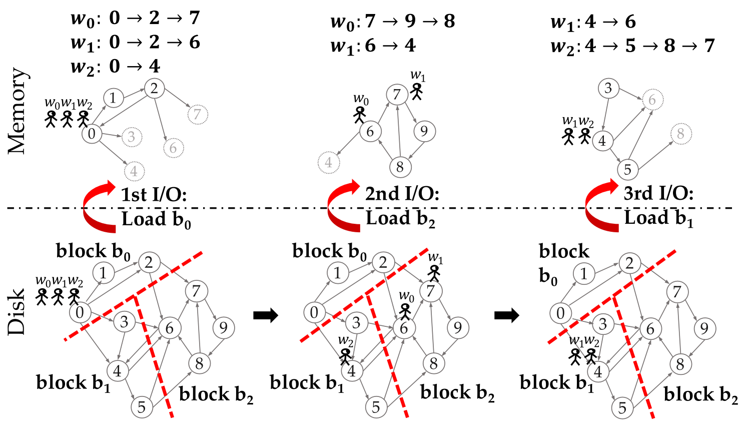


Fig. 5. Main idea of the state-aware model.

into memory, and all walks can be finished. Note that only three I/Os are required in this example. However, for the iteration-based model, it may need 12 I/Os, because it uses four iterations, and generates three I/Os in each iteration.

Remark. We would like to emphasize that the state-aware model is different from the on-demand I/O model proposed in DynamicShards [54] and Graphene [37]. Note that the on-demand I/O model dynamically adjusts the graph blocks layout in each iteration and skips the blocks without containing any walks, but it still follows the iteration-based manner. In addition, even if there is only one walk in a block, it has to load the block into memory for analysis.

There are three key issues that need to be addressed to efficiently realize the preceding idea of state-aware graph storage: (1) self-adaptive graph partitioning, (2) state-aware graph loading, and (3) walk-conscious graph caching. In the following sections, we present the design in details.

3.2 Self-Adaptive Graph Partitioning

In this section, we first present the lightweight subgraph partitioning strategy, which realizes fast and flexible subgraph repartitioning. Then, we introduce a heuristic approach for configuring the subgraph size according to the scale of random walks. In addition, we enable adaptive and timely adjustment of the subgraph size, so as to realize the optimal configuration of subgraph partitioning.

Lightweight graph partitioning. We manage graph data with the widely used **compressed sparse row (CSR)** format, which sequentially stores the out-neighbors of vertices as a *csr file* on disk and uses an *index file* to record the beginning position of each vertex in the csr file. We use the *chunk-based partition method*, which partitions a graph into blocks according to vertex IDs and puts the vertices with adjacent IDs into the same block. This partitioning method is widely used in many graph processing systems for its ease of implementation [6, 32, 37, 54, 58, 60, 62]. In addition, some research shows that in real-world graphs, vertices with adjacent IDs may trend to have more edge connections [60], thus contributing to less communication between different graph partitions. Specifically, we sequentially add vertices and their out-edges into a block according to the ascending order of their IDs until the data size of the block exceeds a predefined threshold denoted as *block size*, and then we create a new block. Figure 6 shows the data layout of the example graph in Figure 1(a). Note that the one-dimensional partitioning with CSR format is sufficient here, as the in-edges of vertices are not in need of random walks. In addition, this lightweight graph data organization decreases the storage cost of each subgraph and thus reduces the time cost of graph loading. As we partition a graph by simply reading through the *index file* once

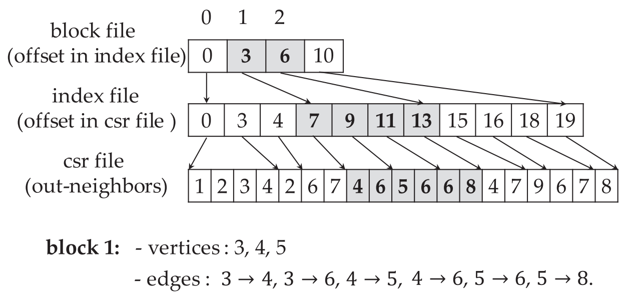


Fig. 6. Graph data organization of the example graph in Figure 1(a). The graph is stored in CSR format and block partition ranges are recorded in the block file.

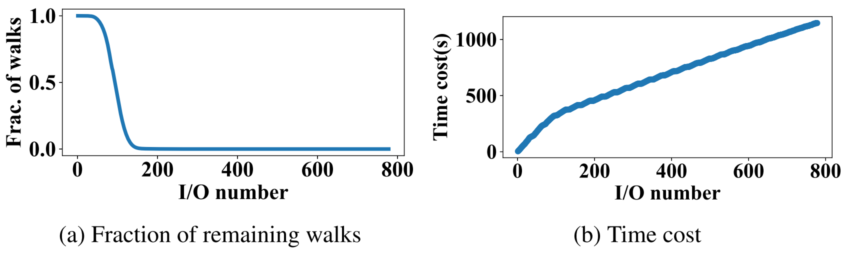


Fig. 7. Global straggler problem.

to record the beginning vertex of each block, it is also flexible to adjust the block size for different application scenarios.

Heuristic block size setting. For setting the graph block size, we find that a trade-off exists. In other words, using smaller blocks can avoid loading more data which are not needed for updating random walks, whereas using larger blocks can have more walks getting updated in each subgraph loading. In addition, different analysis tasks require different walk scales and thus prefer different block sizes. Lightweight tasks with a small number of walks prefer a small block size, as the I/O efficiency can get improved under this setting. In contrast, heavyweight tasks with a large number of walks prefer a large block size, as a large block size can increase the walk updating rate. Based on this understanding, we use an empirical analysis (see Section 5.5 and set the default block size as $2^{(\log_{10} R+2)}$ MB, where R is the total number of random walks. For example, in the case of running 1 billion walks, the default block size is 2 GB, which is usually smaller than the memory capacity of a commodity machine, so it is easy to keep a graph block in memory.

Self-adaptive block size adjusting. In addition, our experiments show that using a fixed block size during the whole executing process is not the best choice for random walk based tasks, because the number of unfinished random walks decreases as the task proceeds. In addition, it is common to have the *global straggler problem*. In other words, some walkers may move very fast and make large progress as the graph data they accessed can always be loaded into memory, whereas other walkers may move very slow as they may be trapped in some blocks which have a very low chance of being loaded into memory. As a result, GraphWalker can quickly complete the majority of walks, but it may take a long time to finish the remaining small number of walks. To better understand the global straggler problem, we run the RWD algorithm, which starts one walk at each vertex and instructs each walk to run six steps, over the Kron30 graph. Please refer to Section 5.1 for the detailed experiment settings. We show the percentage of remaining walks that are not finished and the total time cost after processing each block in Figure 7. The results show that there are few

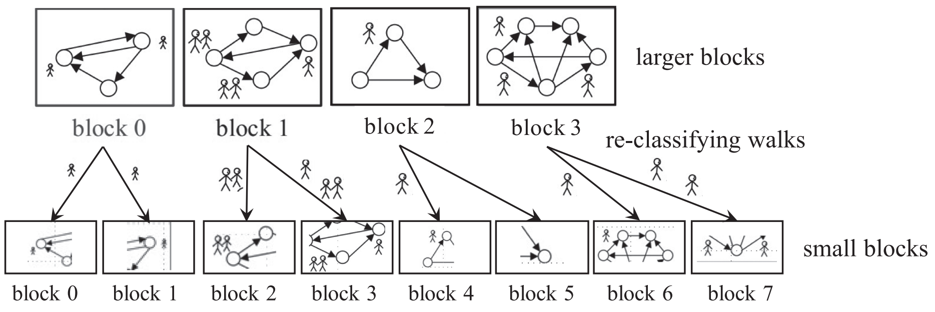


Fig. 8. Splitting blocks needs to reclassify the walks.

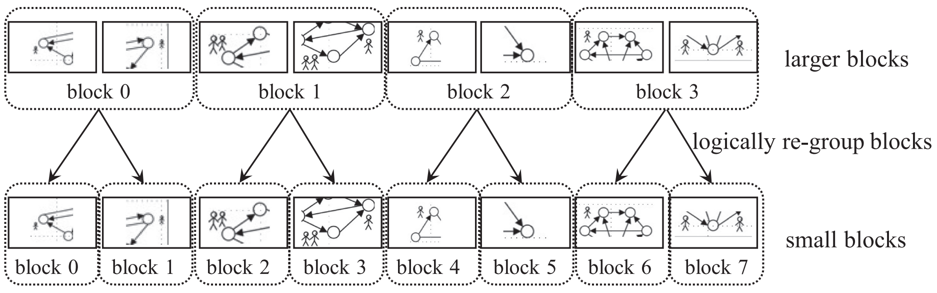


Fig. 9. Illustration on the self-adaptive block size configuration. The graph is first partitioned into fine-grained subgraphs, and the size of the loaded graph block in each I/O is configured by adjusting the number of small subgraphs.

walkers spending more than half of the total I/Os to finish their updates, and these I/Os cost nearly half of the total time cost, which seriously influences the efficiency.

Through the optimizations in the following sections, we can alleviate the global straggler problem, so as to reduce the number of I/Os cost by these stragglers. Meanwhile, considering the reduced number of unfinished random walks, we can also reduce the amount of data that need to be loaded into memory for each I/O. Thus, we propose a *self-adaptive block size adjusting strategy*, which adaptively adjusts the block size according to the remaining number of random walks during the task running process. Specifically, we reset the block size as $2^{\lfloor \log_2 R \rfloor + c}$ KB before each subgraph loading, where R is the total number of unfinished random walks, and c is a constant with a default value $c = 1$, which is used for adjusting the block sizes for different applications. For example, in the case of the remaining 1 million walks, the default block size changes to 1 GB. We also set an upper limit, e.g., 4 GB, to avoid too large block sizes.

As the number of remaining walks keeps decreasing, the most appropriate block size also changes. Thus, the intuitive way is to dynamically split large blocks to smaller ones. However, when we split a large block into many small blocks, we also need to reclassify the walks belonging to the original large block, as illustrated in Figure 8. This would bring a large time cost, because all walks are stored together in the walk pool, and it needs to traverse all of these walks so as to locate the walks belonging to each divided small block.

To avoid the overhead of reclassifying walks and facilitate the management of subgraphs, we first split the whole graph into many fine-grained subgraphs, e.g., the size of each subgraph is set as 2 MB by default, then a number of consecutive subgraphs is grouped together to form a large one. In other words, the loading of graph data can choose a reformed large subgraph in each I/O, as illustrated in Figure 9. Each time, a large subgraph is selected for loading and computing, and

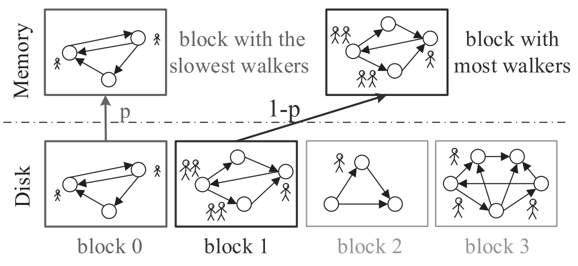


Fig. 10. Walk progress synchronizing.

the appropriate subgraph size (or the number of small subgraphs to be loaded in a single I/O) is calculated according to the total amount of the remaining random walks. For example, in the previous example, when a billion random walks are started at the same time, the appropriate subgraph size is 4 GB, so at the beginning phase, we can load 4 GB/2 MB = 2,048 consecutive subgraphs into memory for calculation in each I/O. As the total number of unfinished random walks keeps decreasing, e.g., when there are only 64,000 random walks left, the subgraph size should be changed to $2^6 = 64$ MB. To realize this adjustment, we can load 64 MB/2 MB = 32 consecutive subgraphs into memory instead of 1,024 subgraphs.

3.3 State-Aware Graph Loading

Based on the subgraph partitioning scheme, we further design a state-aware graph loading strategy according to the distribution of the number of random walks among the subgraphs. The idea is to make the maximum number of random walks get updated during each I/O, so as to maximize the I/O efficiency. However, this state-aware graph loading scheme may incur the global straggler problem, so we also propose a probabilistic approach and a fine-grained vertex loading scheme to address it.

Preferential graph loading. Note that we convert the graph format and partition graph blocks in the preprocessing phase. During the phase of running random walks, we choose a graph block and load it into memory according to the states of walks, and in particular, we preferentially load the block containing the largest number of walks. After finishing analysis over the loaded graph block, we load another block into memory for analysis in the same way.

Probabilistic I/O scheduling. However, simply applying the preceding preferential graph loading scheme may lead to the *global straggler problem*. In other words, some walks may move very fast and make large progress as the graph data they needed can always be satisfied, whereas some other walks may move very slow as they may be trapped in some blocks which are not loaded into memory for a long time. As a result, GraphWalker can quickly complete most walks but takes a very long time to finish the remaining few walks.

To address the global straggler problem, we introduce a probabilistic approach to the state-aware graph loading process. The idea is to give stragglers a chance to move some steps such that they can catch up with the progress of most walks. Specifically, every time we choose a graph block to load, we assign a probability p to choose the block containing walks with the slowest progress, i.e., with the smallest number of walked steps, and with probability $1 - p$, we still load the block with the most walks to follow the procedure of the preferential policy. The probabilistic strategy of loading graph blocks is illustrated in Figure 10. Note that the global straggler problem will be mitigated more efficiently as p increases, but the efficiency of the majority of walks will decrease. So there is a trade-off for setting p. Based on our empirical analysis, we find that $p = 0.2$ is an appropriate setting, and we can get 20% improvement in some cases for some cases.

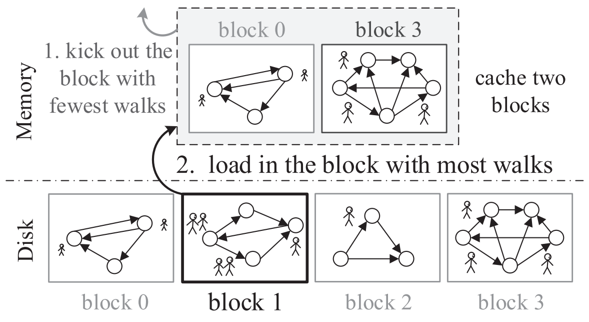

Memory
Disk
1. kick out the block with fewest walks
2. load in the block with most walks
cache two blocks
block 0
block 3
block 0
block 1
block 2
block 3

Fig. 11. State-aware graph loading with block caching.

Fine-grained vertex loading. The probabilistic I/O scheduling strategy can relieve the global straggler problem to some extent, but the last remaining few walkers always need some I/Os to load the corresponding subgraphs into memory to complete the walk updates. Thus, the last few subgraph loading I/Os always bring low efficiency, because only a very small amount of data of the loaded subgraph is in need. Based on this observation, we further adopt a fine-grained vertex loading strategy for the last remaining walk stragglers. Specifically, we load the corresponding graph data in the unit of a vertex instead of a subgraph block, thus avoiding the loading of useless graph data when processing the stragglers.

3.4 Walk-Conscious Graph Caching

Walk-conscious caching scheme. To ease the impact of block size and improve cache efficiency, we also enable block caching by developing a walk-conscious caching scheme to keep multiple blocks in memory. The rationale is that blocks with more walks are more likely to be needed again in the near future. Thus, the graph loading process with block caching works as follows. As illustrated in Figure 11, we first select a candidate block based on the state-aware model, and to load this block, we check whether it is cached in memory or not. If it is already in memory, then we directly access the memory to perform the analysis. Otherwise, we load it from the disk and also evict out the block in memory containing the fewest walks if the cache is full. Note that the maximum number of blocks cached in memory, denoted as *nmblock*, depends on the usable memory size.

We emphasize that this walk-conscious block caching scheme differs from a conventional page cache in the following aspects. First, we do not adopt prefetching, as the state-aware model does not prefer to access graph blocks sequentially. Second, the page cache manages data in memory at a *page* granularity, whereas we manage at a *block* granularity so as to fit the block-based graph loading and computing. Last but not least, the eviction policy also leverages walk states, which is different from LRU. Our experiments show that the walk-conscious block caching scheme always outperforms the conventional page cache scheme. We also experimentally show the caching performance with different cache sizes in Section 5.5.

Variable-size block caching. As the block sizes are changing during the computation process, we need to carefully manage blocks with different sizes in the cache space. One choice is to always cache the blocks with the finest granularity, e.g., in a unit of 2-MB graph blocks, and fetch those small blocks to compose the larger subgraph for random walk forwarding. However, many blocks may be a little bit smaller than the predefined block size according to the partition rules introduced in Section 3.2, because if the remaining space in a block cannot store all out-edges of the current vertex, then all of its out-edges will be stored in a newly created block. Since many small 2-MB blocks may not be fully used, we cannot directly fetch those small blocks from a contiguous

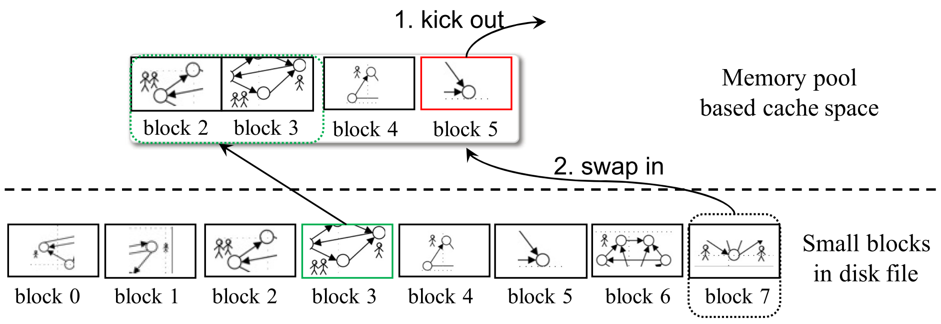


Fig. 12. Variable-size block caching.

memory space to compose the larger subgraph for random walk forwarding. And if we cache each small block by its actual size, which may be slightly smaller than 2 MB, then we cannot always implement the correct block swapping to ensure a contiguous memory space for consecutive small blocks, because the evicted block may be smaller than the block to be loaded.

To address the preceding problems, we adopt a memory pool based cache space management strategy to manage the cached blocks in contiguous memory space and meanwhile implement the correct block swapping, by taking into consideration the changes of block sizes. First, the block sizes are computed according to the total number of unfinished random walks, which keeps decreasing, so the block sizes are also not incremental. Second, the block sizes are computed based on the formula $2^{\lfloor \log_2 R \rfloor + c}$ KB (refer to Section 3.2), so the block sizes always decrease to one-half, and as a result, either all consecutive small blocks in the chosen subgraph are cached or none of them is cached in memory. Based on the preceding two observations, we design the variable-size block caching strategy and manage the cache space as a large memory pool as shown in Figure 12. We first allocate a large contiguous memory pool, and the size can be computed based on the memory budget by default, or set by users. When a subgraph with k consecutive small blocks (each with a maximum size 2 MB) is chosen, we first fetch $k \times 2$ MB space from the memory pool and then load the blocks to the contiguous space, and a little memory space may be wasted as some of the loaded blocks may be smaller than 2 MB. The next time we want to visit a subset of the loaded space, we can just return all contiguous blocks as a larger subgraph for walk forwarding since they are already in memory. When the chosen subgraph is not cached and the remaining cache space has run out, we choose a cached subgraph to evict out, and if the evicted subgraph is a subset of a larger contiguous space with more consecutive 2 MB blocks, then we kick out all of these consecutive small blocks to ensure enough space. After that, we swap in the loaded subgraph and reset the remaining space for the next subgraph loading.

4 WALK INDEX MANAGEMENT AND WALK UPDATES

4.1 Overview

In this section, we present the design details of how to manage walk indexes and update walks. To support large-scale concurrent random walks on large-scale graphs, on the one hand, we need to reduce the memory overhead of maintaining the states of random walks (i.e., walk indexes), and on the other hand, we need to accelerate the updating rate of random walks. To achieve these two goals, we develop a block-centric walk management and asynchronous walk updating strategy, respectively. The basic idea of the block-centric walk management is to reduce the size of walk indexes, and use a fixed-length buffering policy to limit the memory occupation of the walk data of each subgraph, so as to significantly reduce the overhead of managing the states of walks. In

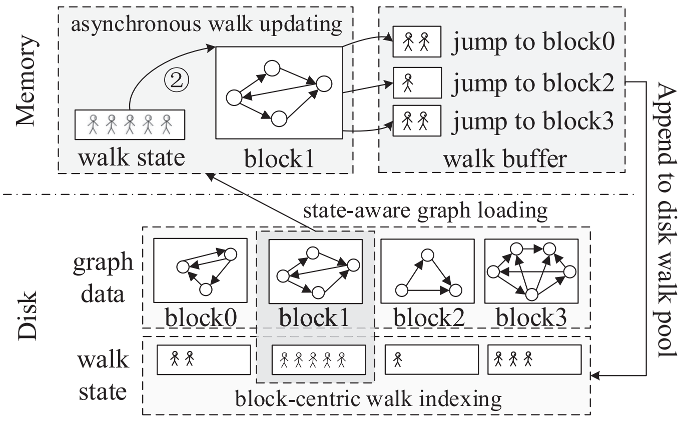


Fig. 13. Overall design of GraphWalker.

addition, the idea of the walk-centric asynchronous updating method is to allow each walk to be continuously updated until it goes out of the boundary of the loaded subgraph in memory, so as to improve the utilization of the loaded data and improve the updating rate of random walks. In addition, we propose an optimization to improve the efficiency of RWR.

Based on the preceding design, we finally implement an I/O-efficient graph system, GraphWalker. Figure 13 shows the processing flow of running a large number of concurrent random walks in GraphWalker, and it can be summarized as four steps: (1) load a sub-graph into memory through the state-aware graph loading strategy (see Section 3), and the corresponding walk data of the sub-graph is also loaded at the same time; (2) update walks over the loaded subgraph based on the asynchronous updating scheme; (3) persist walk data to disk files when the walks jump to other subgraphs and the corresponding memory walk buffer is full; and (4) repeat the preceding three steps until all random walks have completed.

4.2 Block-Centric Walk Management

Walk data structure. We record each walk with three variables, source, current, and step, which indicate the start vertex, the offset of the current vertex in the block, and the number of moved steps, respectively, by default. We record each walk with 64 bits. The number of bits allocated for each variable is shown in Figure 14. This data structure can support starting random walks at 2^{24} source vertices simultaneously, and it also allows each walk to move up to 2^{14} steps. Note that there is no limit on the total number of walks, as we can start many walks at each vertex. In addition, we provide APIs for users to record more walk states in the *WalkDataType* structure for their own random walk algorithms (refer to Section 4.5).

Block-centric walk indexing. To reduce the memory overhead of managing all walk states, we propose a block-centric scheme. For each graph block, we use a walk pool to record the walks that are currently in the block, referring to Figure 14. We implement each walk pool as a fixed-length buffer, which stores at most 1,024 walks by default, so as to avoid dynamic memory allocation cost. When there are more than 1,024 walks in a block, we flush them to disk and store them as a file called the *walk pool file*. Note that we encode each walk with a 64-bit *long* data type, so each walk pool only costs 8 KB. This way, the memory cost for managing walk state is very low. For example, for running 1 billion walks in YahooWeb, with 1.4 billion vertices, GraphWalker costs only 800 KB if it uses 100 graph blocks. However, DrunkardMob costs more than 4 GB to record 1 billion walks, as each walk uses at least four bytes. In addition, these walks jump among the 11.2M dynamic

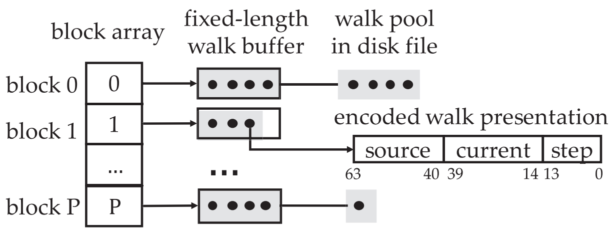


Fig. 14. Block-centric walk management.

arrays (refer to Section 2.2) and thus cause frequent memory re-allocation and bring extra time cost.

Walk pool management. When we load a graph block into memory, we also load its walk pool file into memory and merge the walks with those stored in the in-memory walk pool. Then we perform random walks and update walks in the current walk pool. During the update process, when a walk pool is full, we flush all walks in the walk pool to disk by appending them to the corresponding walk pool file and clear the buffer. When finished computing with the loaded graph block, we clear the current walk pool and sum up the walks in both the walk buffer and walk pool file of each block so as to update the walk states. With this lightweight walk management, we save a lot of memory cost for storing walk states, and thus it is able to support massive concurrent walks. In addition, the fixed-length walk buffering strategy turns many small I/Os for updating walk states after each subgraph processing into several large I/Os, which largely reduces the I/O cost for providing persistent storage of walk states.

4.3 Asynchronous Walk Updating

Loaded data re-entry. Note that in iteration-based systems, with a vertex-centric computing model, after loading a graph block, each walk in the loaded subgraph walks only one step, which induces to a very low walk updating rate. In fact, after walking one step, many walks are still staying at the vertices in the current subgraph, so they can be further updated with more steps by traversing the vertices in the subgraph again. To improve the I/O efficiency, some works use the *loaded data re-entry* [6], which allows the walks to reuse the loaded data. The idea is to re-enter the subgraph again to walk one more step by traversing the vertices in the subgraph again. Moreover, one can also keep re-entering the subgraph until all of the walks reach the boundary of the subgraph.

Local straggler problem. However, the re-entering scheme in iteration-based systems may cause the local straggler problem. In other words, many walks are able to move one step the first time the graph block was just loaded, and as the number of re-entries increases, most walks may reach the boundary of the subgraph, but only few walks remain in the subgraph, and they cost multiple re-entries to finish. To understand the effect of the local straggler problem, we start one walk at each vertex and set the walk length as 6 on the Twitter graph. Please refer to Section 5.1 for detailed settings. We measure the ratio of walks that can get updated in each re-entry of a graph block, and the results are shown in Figure 15. The results show that it totally costs 11 re-entries to finish all walks. However, after 4 re-entries, more than 80% of walks have moved out of the block. In other words, the remaining 7 re-entries only update less than 20% of walks. In fact, all graph blocks show similar trends in our experiments. So the I/O efficiency in the iteration-based system with the vertex-centric model is still very low even with re-entering. We also find that simply stopping walks over the currently loaded subgraph after certain re-entries cannot address the local straggler problem, and it does not reduce the completion time either. The main reason is that even if we temporarily stop walking after some re-entries, say, 4 re-entries in the preceding

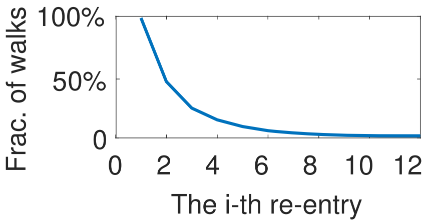


Fig. 15. Local straggler problem.

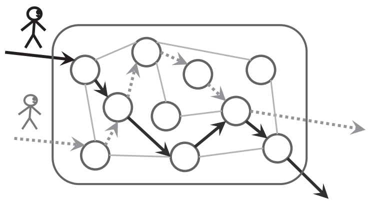


Fig. 16. Asynchronous walk updating in parallel.

experiment example, we still have 20% of walks remaining at some vertices in this subgraph. The graph block is still reloaded with extra I/Os to finish the walks.

Asynchronous walk updating. To further improve the I/O efficiency and walk updating rate, GraphWalker adopts an asynchronous walk updating strategy, which allows each walk to keep updating until it reaches the boundary of the loaded graph block. After finishing a walk, we choose another walk to process until all walks in the current graph block are processed. Then we load another graph block based on the state-aware model described earlier. Figure 16 shows an example of processing two walks within the same graph block. To accelerate the computation, we also use multithreading to update walks in parallel. We emphasize that with our asynchronous walk updating model, we completely avoid useless visits of vertices and eliminate the local straggler problem. By making full use of the loaded graph block, we significantly improve the I/O efficiency and reduce the number of I/Os.

Parallel random selection. For fast processing of the many random walks, we use multithreading to process the walks of a block in parallel. We need to generate a random number to do the random neighbors selecting. However, if the multithreads simultaneously call the library function *rand()*, they actually used the same seed for computing, which would invoke the data race and cause deadlock. Therefore, we use the function *rand_r(&seed)* instead, who specifically assigns the seed value and avoid the data race, and we use the walk value as the seed value to ensure the randomness.

4.4 Optimization on RWR/RWJ

RWR and RWJ. RWR [52] is a variant of the simple random walk. The only difference is that at each step, the walker will jump back to the source vertex with a small probability c, i.e., restart with probability c, and with probability $1-c$, it randomly walks to a neighbor vertex just like the simple random walk. A similar random walk variant is RWJ, in which the walker will jump randomly to an arbitrary vertex with a small probability c at each step, and with probability $1 - c$ randomly walks to a neighbor vertex. Note that RWR and RWJ are widely used in many applications, such as PPR [20] and automatic caption of images [44]. More importantly, for random walk over directed graphs, in which some sink vertices may not have out-edges, RWR and RWJ are usually used, instead of the simple random walk, so as to guarantee the convergence.

Optimization of joining short walks.Optimization of joining short walks. We find that running long walks may take longer time than running more shorter walks, as long walks are hard to compute in parallel. This is also validated by our experiments (see Section 5.2.1). Thus, to further improve the performance of RWR, we adopt the idea of *joining short walks*, which was first proposed and proved to be correct in the work of Fogaras et al. [15]. Specifically, we start more random walks from the same source vertex, and at each step, if the original RWR will restart with probability c, then we simply stop the walking with probability c. With probability $1 - c$, we follow the same process of the simple random walk. We call this process **random walk with stop (RWS)**. This way, we can generate a lot of walking sequences with different lengths, but all of them start from the same source vertex, then we can join these walks to form long walks.

We further use an example to illustrate the preceding joining method. Suppose that we start random walks from vertex 0 by following the RWS process and generate the following three short walks.

$$\begin{aligned} w_0 &= 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \\ w_1 &= 0 \rightarrow 4 \rightarrow 7 \rightarrow 3 \\ w_2 &= 0 \rightarrow 7 \rightarrow 4 \rightarrow 8 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 7 \rightarrow 2 \rightarrow 1 \rightarrow 6 \end{aligned}$$

Then we can join the three short walks w_0, w_1, w_2 to form a long walk as follows.

$$\begin{aligned} W = &\underbrace{0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4}_{w_0} \rightarrow \underbrace{0 \rightarrow 4 \rightarrow 7 \rightarrow 3}_{w_1} \rightarrow \\ &\underbrace{0 \rightarrow 7 \rightarrow 4 \rightarrow 8 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 7 \rightarrow 2 \rightarrow 1 \rightarrow 6}_{w_2} \end{aligned}$$

The rationale is that as long as the stop probability in RWS is equal to the restart probability in RWR, then W can be taken as one walk sequence generated by RWR.

Note that this optimization can also be applied to optimize RWJ, in which the walker will jump to a random vertex with probability c at each step. The only difference is that when generating short walks for joining, we should start more random walks from the randomly chosen vertices, then follow the same process of RWS, and finally join these walks to form long walks to simulate RWJ.

Summary. We point out that this joining method can also be used in the iteration-based computation model in existing systems, although the benefit is limited. Because the iteration-based computation model adopts a synchronized computation, all walks have strictly the same progress, whereas the walks conducted with RWS may have very different lengths. Thus, the number of iterations needed in the iteration-based model depends on the longest walk of RWS, and the last few iterations may contain only one or two walks, with a lot of loaded data wasted. Another approach to solve this problem and accelerate the progress is to truncate the paths at a given length L, and it results in only a very slight loss of precision [15]. However, with the state-aware graph loading strategy and the asynchronous walk updating in GraphWalker, it is easy and also efficient to realize the optimization of joining short walks for RWR/RWJ without losing any precision caused by walk truncating. We also conduct experiments to validate the efficiency of GraphWalker compared with the iteration-based computation graph system, by using this same joining method in Section 5.4.4.

4.5 GraphWalker APIs

4.5.1 Key Interfaces. We also provide programming APIs for users to develop their own random walk algorithms on top of GraphWalker. We describe the key interfaces as follows.

ALGORITHM 1: *ForwardWalk()* function for Node2Vec.

```
Require: $w, G(b), p, q$ ▷ $w$ is walk data, $G(b)$ is subgraph data, $p$ and $q$ are return parameter and in-out parameter defined by Node2Vec.
Ensure: $w$ is currently stay in $G(b)$
  $last\_adjlist = w.last\_adjlist$
  $t = w.last\_vert$
  $v = w.cur\_vert$
  $h = w.hop$
  while $v \in G(b)$ and $h \leq max\_hop$ do    ▷ Forward this walk until it reaches the subgraph boundary
      updateInfo()
      $cur\_adjlist = adjlist[v]$
      while 1 do
          $x \leftarrow$ random neighbor in $adjlist[v]$
          $acc\_prob \leftarrow$ random probability
          if $(d_{tx} = 0$ and $acc\_prob \leq 1/p)$ or $(d_{tx} = 1$ and $acc\_prob \leq 1)$ or $(d_{tx} = 2$ and $acc\_prob \leq 1/q)$ then
              $last\_adjlist = cur\_adjlist$
              $t = v$
              $v = x$
              $h++$
              break
          end if ▷ $d_{tx}$ is the distance between $t$ and $x$. $d_{tx} = 0$ means $t$ and $x$ are the same vertex, $d_{tx} = 1$ means $x$ is adjacent with $t$ and $d_{tx} = 2$ otherwise.
      end while
  end while
```

WalkDataType. We use an encoded structure to record the source vertex, current vertex, and the number of walked steps to reduce the memory consumption as described in Section 4.2 by default. For usability, we also use a template-based class API named *WalkDataType* for users to record all walk state data in need for their applications. For example, for some applications such as graph embedding algorithms, which may need walk path information, i.e., the vertex sequence visited by each walker, then an extra unique walk ID can also be used to record in this data structure.

StartWalks. This interface is used for starting and initializing walks according to the needs of different random walk applications. For instance, single-source PPR starts all walks from the given source vertex, and RWD starts a given number of walks from each vertex in the graph. All started walks are initialized with $current = source$, and $step = 0$, and distributed to the corresponding walk pools according to the graph blocks to which they belong.

ForwardWalk. This is the core interface for executing random walk algorithms, which are specified with the walk updating model, walk transition probability, and walk termination condition of random walk. The default walk updating way is walk-centric updating, i.e., a walk keeps forwarding for multiple steps in a subgraph until it reaches the boundary of the subgraph or meets the termination condition. The walk transition probability is the key part for random walks. For example, a simple random walk just randomly selects the neighbor of the current vertex with unbiased probability, and an RWR will jump back to the source vertex with a small probability c at each step. The walk termination condition is generally specified as each walk terminating with a fixed walk step or terminating with a fixed probability at each step.

UpdateInfo. We also develop an interface to update the information for the specific applications during the walk forwarding process, such as recording the visit frequencies of each vertex in PPR, or the walk path for each walker, and so on.

4.5.2 Supports for High-Order Random Walks. Users can also develop high-order random walk algorithms on top of GraphWalker, through storing the extra needed information in *WalkDataType* and implementing the interfaces provided earlier. For example, to realize the high-order random walk algorithm *Node2Vec* [19], more information about the previously visited vertex is needed, so we can add those walk state data (the last visited vertex and its adjacency list) to *WalkDataType* besides the default setting. Then, a Node2Vec walk determines the walk transition probability according to the previously visited vertex, so we implement it in the *ForwardWalk()* function and state the pseudocode in Algorithm 1. We also evaluate the performance of running Node2Vec in GraphWalker compared with KnightKing [58] in Section 5.3.2.

5 EVALUATION

GraphWalker aims for providing fast and scalable random walks, so we take DrunkardMob [31], the state-of-the-art single-machine random walk specific graph system, as a baseline for performance comparison. In addition, there are several single-machine graph systems, which further optimize the system performance from different aspects, such as fine-grained block partition, asynchronous I/O to support pipeline between I/O and computation, and huge page support to reduce TLB miss. These optimizations are not specific for random walk, and many of them are also orthogonal to the optimizations in GraphWalker. For completeness, we also compare GraphWalker with two state-of-the-art open-source single-machine graph systems, Graphene [37] and GraFSoft [27]. To further validate its scalability, we compare GraphWalker with the most recent distributed random walk graph system, i.e., KnightKing [58].

5.1 Experiment Settings

Testbed. All experiments are performed on a Dell Power Edge R730 machine with 64 GB of memory and 24 Intel Xeon CPU E5-2650 v4 @ 2.20-GHz processors. The entire graph data are stored on a 3.2-TB RAID-0 consisting of seven 500-GB SamSung 860 SSDs if we do not state specifically. We also study the performance of GraphWalker on HDDs. For the distributed system KnightKing [58], it is run on an eight-node cluster with 56-Gbps Ethernet interconnection, and each node is equipped with two eight-core Intel Xeon E5-2620 v4 processors with 20-MB L3 cache and 64 GB of DRAM.

Dataset. Table 1 lists the statistics of the six graph datasets we used. TT [4], FS [1], YW [5], and CW [3] are real-world graphs. K30 and K31 are two synthetic graphs generated with Graph500 Kronecker [2]. These graphs are all widely used in graph system evaluations. CSR Size indicates the minimum storage cost by storing graphs in CSR format, and Text Size is the size of the dataset stored in text format as an edge list. We point out that Kron30, Kron31, and CW are large graphs that cannot be entirely put into the memory in our testbed, and CW is the largest publicly available web corpus.

We also show the default partition results of these datasets in GraphWalker in Table 2, which includes the default block size, the number of blocks in total, and the number of blocks cached in memory for each graph. Note that the default block size is 2 MB as stated in Section 3.2. However, there may be some super vertices with too many neighbors in the graph, which cannot be fully stored in a single block of 2 MB. In these cases, we set the default block size as the finest granularity that can fully store the vertex with the maximum number of neighbors in the graph, e.g., 16 MB for Twitter. The number of cached blocks is computed based on the memory budget, e.g., we set the default memory budget as 44 GB for a machine with 64 GB of memory.

Table 1. Statistics of Datasets

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	CSR Size	Text Size
Twitter (TT)	61.6M	1.5B	6.2 GB	26.2 GB
Friendster (FS)	68.3M	2.6B	10.7 GB	47.3 GB
YahooWeb (YW)	1.4B	6.6B	37.6 GB	108.5 GB
Kron30 (K30)	1B	32B	136 GB	638 GB
Kron31 (K31)	2B	64B	272 GB	1.4 TB
CrawlWeb (CW)	3.5B	128B	540 GB	2.6 TB

Table 2. Default Partition Results of Datasets in GraphWalker

Dataset	Block Size	No. of Blocks	No. of Cached Blocks
Twitter (TT)	16 MB	356	356
Friendster (FS)	2 MB	4,935	4,935
YahooWeb (YW)	2 MB	12,660	12,660
Kron30 (K30)	128 MB	1,027	352
Kron31 (K31)	128 MB	2,054	352
CrawlWeb (CW)	2 MB	243,047	22,528

Graph algorithms. In addition to directly evaluating the performance of running random walks, we consider the following four common random walk based algorithms:

- *Random walk domination* [35]: We start one walk of length 6 from each vertex in the graph to find a vertex set that has the maximum influence diffusion. It aims to find a target set of vertices to maximize the expected number of vertices that can be reached by L-length random walks started from the vertices in the target set. Here, we set $L = 6$ and start one walk at each vertex in the graph.
- *Graphlet concentration (graphlet)* [45, 46]: We use a special graphlet, triangle, as a study case. We randomly start 100,000 random walks of length 4 to estimate the ratio of triangles in the graph.
- *Personalized PageRank* [20, 26]: We focus on the random walk based version [15], and we simulate 2,000 random walks of length 10 starting at each query source vertex to approximate the PPR, which was shown to be sufficient to ensure the accuracy.
- *SimRank* [25]: SimRank measures the similarity between a pair of vertices by starting a certain number of walks from each query vertex to compute the expected meeting time. In the experiments, we start 2,000 random walks of length 11 respectively from the query pair vertices to compute the expected meeting time, similar to the setting in the work of Jeh and Widom [25].

Remark. The first two are graph computation algorithms that utilize the entire graph, whereas the other two are graph query algorithms that need only a portion of the graph. We point out that all of them are classical and representative graph algorithms. We run each experiment 10 times and compute the average completion time. Before each execution, we also clear the page cache to avoid its impact.

5.2 Comparison with RW-Specific Systems

We validate the efficiency of GraphWalker by comparing it with DrunkardMob, the state-of-the-art single-machine system that is specially optimized for random walk. Both GraphWalker and DrunkardMob are implemented based on GraphChi. Note that random walk based algorithms

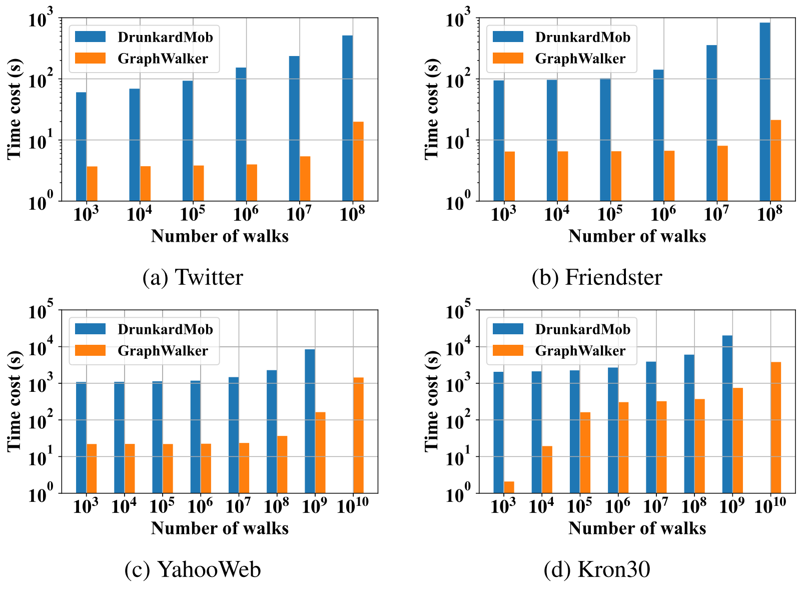
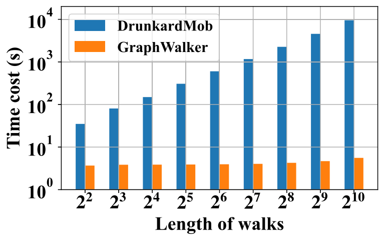


Fig. 17. Performance of random walks with different numbers of walks by fixing the walk length as 10.

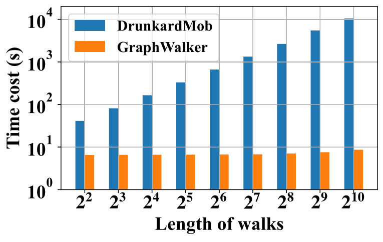
usually require to start a certain number of random walks with a certain walk length, so we evaluate the performance by considering different random walk configurations, so as to study the performance of the entire design space and demonstrate the scalability of GraphWalker in supporting large amount of random walks with very long walk length. We also show the performance of the four random walk based algorithms.

5.2.1 Performance Study in Entire Design Space. We first show the results by fixing the walk length to 10 but varying the number of walks from 10^3 to 10^{10}, as depicted in Figure 17. In each figure, the x-axis indicates the number of walks configured in each experiment, and the y-axis shows the time needed to finish running all of these walks. First, we can see that GraphWalker is consistently faster than DrunkardMob under all settings for different numbers of walks and different graph datasets. In particular, in the case of running 10^6 walks on YahooWeb, DrunkardMob costs near 20 minutes, whereas GraphWalker takes only 17.8 seconds. In other words, GraphWalker achieves 70× speedup. In general, GraphWalker achieves 16× to 167× speedup under all settings. In addition, for the case when the number of walks is not too large, the I/O cost is a dominant factor, so the total time of running different numbers of walks is almost a constant. However, as the number of walks continues to increase, computation cost becomes larger, so the total time cost also increases linearly when we run more random walks.

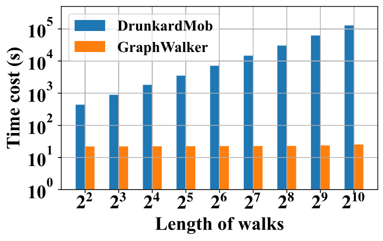
One attractive feature of GraphWalker we like to highlight is its scalability. We point out that even for running tens of billions of random walks on large graphs, GraphWalker can still finish within a reasonable time, e.g., from tens of seconds to around 1 hour, depending on the graph scale. DrunkardMob cannot finish running 10^{10} walks even within tens of hours. However, DrunkardMob even fails to run 10^{10} walks on large graphs, e.g., YahooWeb and Kron30, due to the out-of-memory error, so we do not show the results of DrunkardMob in the setting of more than 10^{10} walks. More importantly, when the graph becomes really large, DrunkardMob may fail to run. For example, for Kron31 and CW, DrunkardMob also encounters the out-of-memory error. Thus, we do not show the results on them for the interest of space. The main reasons are as follows:



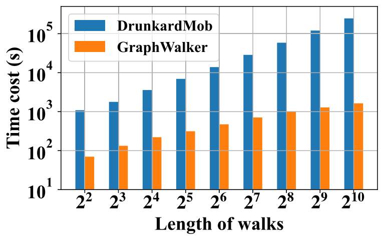
(a) Twitter



(b) Friendster



(c) YahooWeb



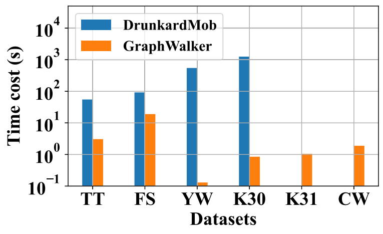
(d) Kron30

Fig. 18. Performance of random walks with different walk lengths by fixing the number of walks as 10^5.

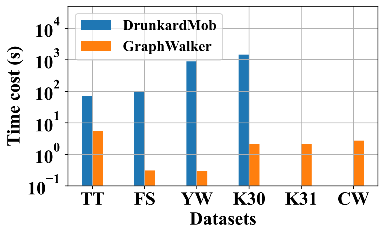
(1) DrunkardMob keeps all walk states in memory, so it is hard to support massive walks, and (2) DrunkardMob employs a dynamic array index for every 128 vertices, so it incurs a large memory overhead when the graph becomes really large, and it is also hard to write the walks to disk for too many open files needed. However, because of the block-centric walk indexing design and keeping walk states on disk, GraphWalker is capable to support huge graphs, e.g., Kron31 and CW, even for running tens of billions of random walks. For example, GraphWalker finishes running 10^{10} walks on CW within around 1 hour.

We also evaluate the performance by varying the walk length. Here we fix the number of walks as 10^5 and vary the length of each walk from 2^2 to 2^{10}. The results are shown in Figure 18. First, we can see that GraphWalker is always much faster than DrunkardMob, and it achieves even more than three orders of magnitude in the best case. In particular, when the graph is not extremely large, e.g., for Twitter, Friendster, and YahooWeb, the time cost of DrunkardMob continues to increase when running longer walks, whereas that of GraphWalker is almost a constant. This is because GraphWalker can cache almost the whole graph in memory for medium-sized graphs, due to the lightweight block storage and optimized block catching strategy, and thus incurs very low I/O cost. For very large graphs that cannot be fully put in memory, e.g., Kron30, the time cost of both DrunkardMob and GraphWalker increases as walks get longer, as GraphWalker needs to swap in and kick out blocks between memory and disk in this case. However, we point out that GraphWalker is much faster, e.g., it achieves 13× to 151× speedup even for Kron30. This experiment also demonstrates the scalability of GraphWalker in supporting long random walks that have thousands of steps.

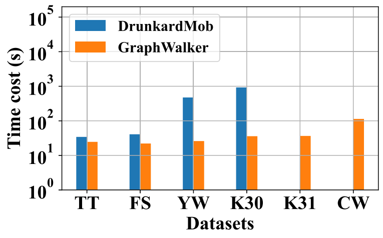
5.2.2 Performance of Random Walk Based Algorithms. We now evaluate the performance of the four common random walk based algorithms described in Section 5.1. From Figure 19, we can see that for general cases, GraphWalker achieves 9× to 691× speedup on DrunkardMob. In particular, in some special cases, e.g., running PPR and SR on YahooWeb, GraphWalker even achieves more than three orders of magnitude speedup. This is because YahooWeb has a very good locality at the query vertices, so GraphWalker only needs to load several corresponding subgraphs to run



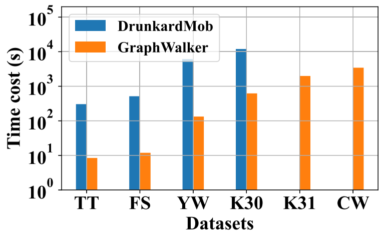
(a) Personalized PageRank



(b) SimRank



(c) Graphlet



(d) Random walk domination

Fig. 19. Performance of random walk based algorithms.

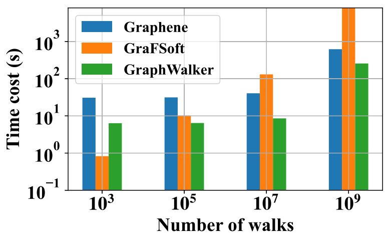
random walks. However, DrunkardMob needs to iteratively scan the entire graph and updates walks in a synchronized manner, so it has a very low I/O efficiency and takes a long time.

We like to point out that DrunkardMob again fails to handle the two largest graphs for the same reason explained in Section 5.2.1, so we skip the results in these cases. Note that this experiment also demonstrates the scalability of GraphWalker in supporting massive walks and huge graphs.

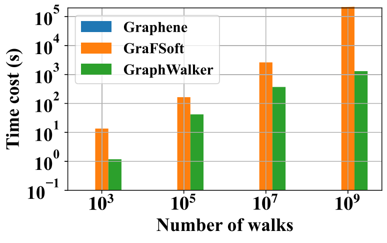
5.3 Comparison with State-of-the-Art Systems

5.3.1 Single-Machine Graph Systems. There are a number of optimizations being proposed in recent single-machine graph systems, e.g., fine-grained block partition, asynchronous I/O to support pipeline between I/O and computation, and huge page support to reduce TLB miss. These optimizations are not specific for random walks, so many of them are also orthogonal to the optimizations in GraphWalker. Thus, to further demonstrate the efficiency of GraphWalker, we also compare it with two state-of-the-art open-source single-machine systems, Graphene [37] and GraFBoost [27]. Note that GraFBoost also uses hardware to accelerate computation, and for fair comparison, we only focus on the pure software implementation of GraFBoost called *GraFSoft*. Note that GraphWalker is implemented based on the baseline system GraphChi, so it does not include the preceding design optimizations. In this experiment, we focus on the case of running random walks starting from a single source. We fix the walk length as 10 and vary the number of walks. Note that Graphene is a semi-external system that stores graph data on disk while keeping all walk states in memory, so it is unable to handle the case of massive walks, e.g., greater than 10^9 walks, or large graphs, e.g., larger than Friendster, due to its high memory cost. In the interest of space, we only show the results for Friendster, the largest graph that Graphene can process, as well as the largest graph CW. We observe similar results for other graphs.

The results are shown in Figure 20, and we can have several conclusions. First, GraphWalker consistently outperforms Graphene even though Graphene is a semi-external system that does not require I/Os to write back walk states, and it achieves up to 40× speedup. More importantly, GraphWalker is also scalable to run huge amount of walks, say tens of billions, as well as process extremely large graphs, e.g., hundreds of billions of edges as in CW, whereas Graphene fails to run



(a) Friendster



(b) CrawlWeb

Fig. 20. Comparison with Graphene and GraFSoft.

in these cases due to its high memory cost caused by the semi-external design. Second, compared with GraFSoft, when the number of walks is small, the improvement of GraphWalker is limited, because each block can only have a few walks given the small total number and the state-aware I/O model cannot bring too much benefit. However, the improvement of GraphWalker increases as the number of random walks gets larger. For example, when running 1 billion random walks on CW, GraphWalker spends only 21.8 minutes, whereas GraFSoft cannot even complete the task within 24 hours. In other words, GraphWalker achieves at least 40× speedup. More importantly, as we increase the number of walks, the increase of time for GraphWalker is *sublinear* and much slower than that of GraFSoft, and this further demonstrates the scalability of GraphWalker in supporting a huge amount of random walks.

5.3.2 Distributed Random Walk System. To further demonstrate the scalability of GraphWalker, and its resource-friendly feature, we also compare it with a distributed graph system, KnightKing [58], which is the most recently proposed distributed graph system optimized for random walks. Note that KnightKing mainly focuses on the walk updating efficiency, whereas GraphWalker mainly considers the I/O efficiency, thus the optimization techniques in the two systems are also orthogonal to each other. We focus on two random walk based algorithms that are used in KnightKing, i.e., PPR and Node2Vec. Specifically, we let PPR starts one walk at each vertex, and each walk terminates with probability t in each step as KnightKing does. We set $t = 0.15$, which is a very common setting in various applications [15, 33]. Note that smaller t means larger average number of walk steps and requires more computations, so KnightKing uses a small t to demonstrate its computing efficiency. For Node2Vec [19], we also start one walk at each vertex and let each walk move 80 steps, the same as KnightKing. Please refer to the work of Grover and Leskovec [19] and Yang et al. [58] for more algorithm introduction about Node2Vec. As the work on KnightKing [58] uses a cluster of eight machines for evaluation, to enable cross validation, we also use eight machines at most, and they are connected with 1 or 56 Gbps Ethernet. We focus on the largest two graphs that can be handled by KnightKing with eight machines, i.e., Twitter and Friendster. We also convert the two graphs to undirected graphs as in KnightKing.

Table 3 shows the results, where *KK* refers to KnightKing, *GW* refers to GraphWalker, and the number after each system name denotes the number of machines being used. We can see that for KnightKing, as the cluster size increases, the computing time, i.e., the time for updating walks, gets reduced greatly, but it still costs a lot of time for graph preparation, which includes reading files, partitioning, and organizing graphs. This is because KnightKing mainly focuses on optimizing the walk computing efficiency, but not disk I/Os for graph management. However, GraphWalker only costs a little time to realize the lightweight logical graph partitioning (refer to Section 3.2). In addition, KnightKing causes extra cost for walk communication between cluster machines, especially for slow networks. Note that the results of the walk updating time in this experiment are

Table 3. Comparison with KnightKing

(a) PPR on Twitter

Time Cost (s)	1 Gpbs		56 Gpbs		—
	KK(4)	KK(8)	KK(4)	KK(8)	GW(1)
Graph preparing	44.70	31.87	35.90	23.09	0.41
Walk updating	2.22	1.62	1.68	1.21	3.19
Extra communication or I/O	13.82	16.28	0.40	0.95	4.95

(b) PPR on Friendster

Time Cost (s)	1 Gpbs		56 Gpbs		—
	KK(4)	KK(8)	KK(4)	KK(8)	GW(1)
Graph preparing	108.84	72.31	89.65	57.64	0.66
Walk updating	7.23	3.49	5.70	2.82	5.55
Extra communication or I/O	12.63	14.68	0.37	1.02	8.93

(c) Node2Vec on Twitter

Time Cost (s)	1 Gpbs		56 Gpbs		—
	KK(4)	KK(8)	KK(4)	KK(8)	GW(1)
Graph preparing	43.21	30.80	34.47	21.66	0.41
Walk updating	51.34	26.12	50.18	26.00	230.50
Extra communication or I/O	339.99	324.94	10.69	13.31	6.77093

(d) Node2Vec on Friendster

Time Cost (s)	1 Gpbs		56 Gpbs		—
	KK(4)	KK(8)	KK(4)	KK(8)	GW(1)
Graph preparing	107.89	73.48	90.16	57.366	0.66
Walk updating	102.19	51.86	99.48	52.50	333.02
Extra communication or I/O	398.35	351.11	17.45	23.85	9.23

consistent with those in the work on KnightKing. In contrast, GraphWalker mainly targets the I/O efficiency problem, and also adapts the walk updating process accordingly based on its I/O model, so it can realize very fast random walks over disk-resident graphs. Here the walk updating time also includes the time for walk index persistency. We also see that GraphWalker achieves comparable performance even compared with KnightKing running on eight machines for PPR. As for Node2Vec, KnightKing outperforms GraphWalker for its improvement for updating high-order random walks. In addition, for the largest graph CW, KnightKing may need a larger cluster to run according to the estimation of its used resources when processing other smaller graphs. Thus, we can conclude that GraphWalker is also a resource-friendly alternative.

5.4 Discussion of Design Choice

5.4.1 Efficiency of State-Aware Graph Loading and Walk Updating. Recall that the inefficiency of existing systems in running random walks mainly come from the iteration-based I/O model, and thus they suffer from low I/O efficiency and low walk updating rate (refer to Section 2). To better understand why GraphWalker could significantly improve the overall performance as presented in the previous sections, we further show the efficiency of our state-aware graph loading and walk updating strategies by considering these two micro-benchmarks to show how GraphWalker addresses the limitations. We also show the time cost breakdown to see how GraphWalker improves the performance of each part along the random walk process. Note that as DrunkardMob uses the fixed block size setting, we also turn off the self-adaptive block size adjusting and fine-grained vertex loading modules in GraphWalker in this group of experiments for a fair comparison. *We*

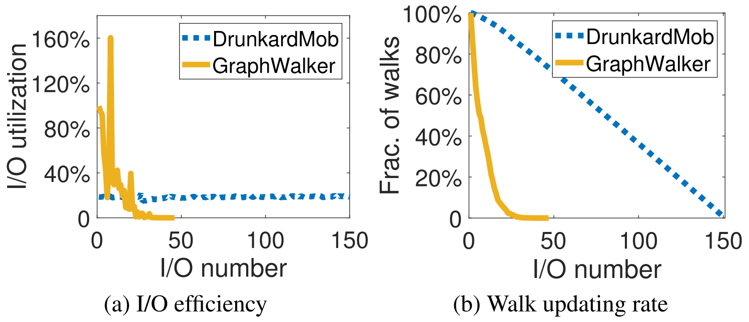


Fig. 21. I/O efficiency and walk updating rate.

only show the results of running the RWD algorithm on YahooWeb, and the results are similar for other settings.

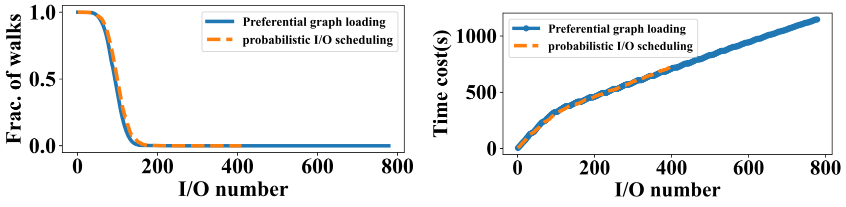
I/O efficiency. I/O efficiency is defined as the edge usage amount for updating walks divided by the total number of edges loaded into memory by one I/O, i.e., a subgraph loading. Here the subgraph may be cached in memory. For example, DrunkardMob may cache some data in the Linux page cache, and GraphWalker may cache some blocks in the walk-conscious cache space (refer to Section 3.4), thus we may not need to actually read from the disk for these cases. Note that an edge may be reused by different walks, so we sum up the total times of being used for all edges. Thus, the I/O efficiency defined here may exceed 100% when one edge is used by multiple walks. We point out that DrunkardMob partitions the YahooWeb graph into 25 shards, and the walk length is 6, so the total number of I/Os required by DrunkardMob is 150. We can see that the I/O efficiency is only around 20% as shown in Figure 21(a). In contrast, GraphWalker needs only 46 I/Os to complete all walks, so the number of I/Os is significantly reduced. The I/O utilization of GraphWalker is also much higher than that of DrunkardMob. Specifically, the utilization of the first few I/Os reaches up to 80% to 160%; this is because the subgraphs loaded by the first few I/Os have the most walks, and many of them may use more than one edge to update. Even for most I/Os, the I/O efficiency of GraphWalker is between 40% and 80%, which is 2× to 4× compared to that of DrunkardMob.

Walk updating rate. Now we study the walk updating rate, shown in Figure 21(b). Recall that DrunkardMob updates 50 million steps per I/O on average and costs 150 I/Os to finish the computation. Although GraphWalker significantly improves the walk updating rate, it only needs 46 I/Os to complete all walks and updates 185 million steps per I/O on average, which is 3.7× higher than that of DrunkardMob. The main reason is that DrunkardMob adopts the iteration-based I/O model, and it walks only one step for each walk when loading one block. In contrast, GraphWalker develops an asynchronous walk updating method to fully utilize the loaded graph data in memory (see Section 4.3), so each walk may move multiple steps over the subgraph loaded by each I/O. As a result, GraphWalker saves a lot of I/Os and completes all random walks more quickly than DrunkardMob.

Time cost breakdown. To better understand the effect of the design optimizations in GraphWalker, we also show the time cost breakdown in Table 4. Note that in the whole execution procedure, there are three key operations: (1) graph loading, which loads graph blocks into memory with disk I/Os; (2) walk updating, which updates the walk states maintained in memory; and (3) walk persisting, which includes to read walk states from disk into memory and write back updated states to disk for persistency. In addition, the three operations are proceeded in an interleaved way, so we aggregate the total time of executing each operation. From the results, we can

Table 4. Time Cost Breakdown

Time Cost (s)	DrunkardMob	GraphWalker	Speedup
Graph loading	1,005	47	21×
Walk updating	3,029	214	14×
Walk persisting	1,056	16	66×
Total runtime	5,110	278	18×



(a) Fraction of remaining walks (b) Time cost

Fig. 22. Impact of probabilistic I/O scheduling.

see that GraphWalker outperforms DrunkardMob in all aspects. The improvement is achieved by the integration of multiple design optimizations. Specifically, data loading efficiency mainly comes from the state-aware graph loading with lightweight graph data organization, which reduces the amount of I/Os, and the asynchronous walk updating strategy also contributes to the reduction of the number of I/Os. For the walk updating efficiency, it mainly comes from the state-aware graph loading and asynchronous walk updating, which increase the number of walk steps that can be updated over each loaded subgraph. Finally, the block-centric walk indexing and fixed-length walk buffer strategy convert many small I/Os of persisting walk states into large I/Os and thus decrease the I/O cost for managing walk states.

5.4.2 Efficiency of Probabilistic I/O Scheduling. GraphWalker introduces a probabilistic approach to relieve the global straggler problem, and we also study the improvement achieved by this approach. We also run RWD with and without the probabilistic I/O scheduling, respectively, on Kron30 graph, and show the percentage of remaining walks that are not finished and the total time cost after processing each subgraph in Figure 22. The x-axis shows the I/O number, i.e., subgraph loading number, which includes the situations of reading target blocks from disk and having already cached the target blocks in memory. We can see that after applying the probabilistic I/O scheduling strategy, we may reduce more than 30% I/Os to finish the forwarding for all walks, and also reduce the total time cost, thus improving the walk updating efficiency.

5.4.3 Efficiency of Self-Adaptive Graph Loading. We also study the improvement achieved by the self-adaptive graph loading strategy by running 10^6 random walks of 10 steps on the CW dataset. Figure 23 shows the time cost of using different block size configuration policies. The x-axis indicates the block size configuration policies. “Fixed” represents the policy of using fixed subgraph size (1 GB in this experiment), “Adaptive” represents the walk-state-aware self-adaptive policy, i.e., resetting the block size according to the remaining number of unfinished random walks, and “Adaptive-VL” means the policy of using both the self-adaptive block size adjustment and the vertex-centric fine-grained graph loading. The y-axis of the figure shows the time needed to finish running all walks. From the results, we can see that by using the self-adaptive block size adjustment policy, we can reduce the time cost by around 21.7%, and by also using the fine-grained vertex

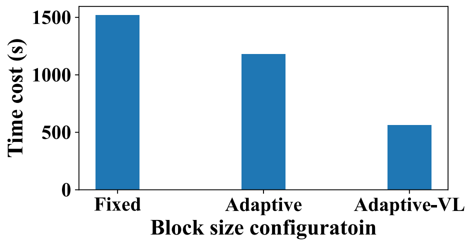
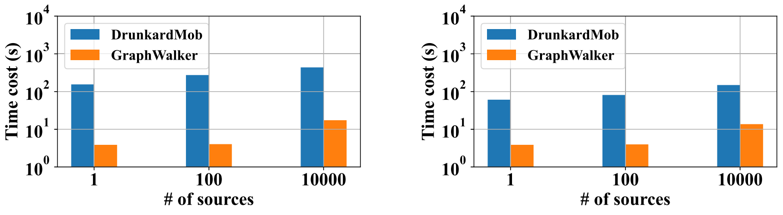


Fig. 23. Performance improvement of self-adaptive graph loading.



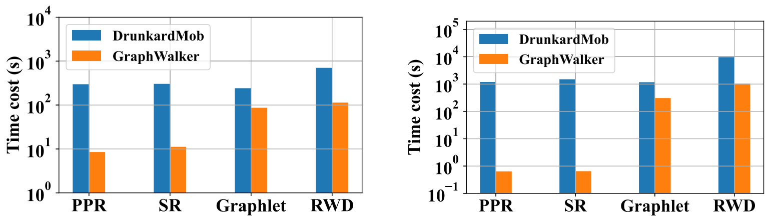
(a) PPR without truncating (b) PPR with truncating walks of length $L > 10$

Fig. 24. Performance improvement of joining short walks for RWR.

loading strategy, we can further reduce half of the time cost. These results show the effectiveness of the walk-state-aware and self-adaptive graph loading ways proposed in GraphWalker.

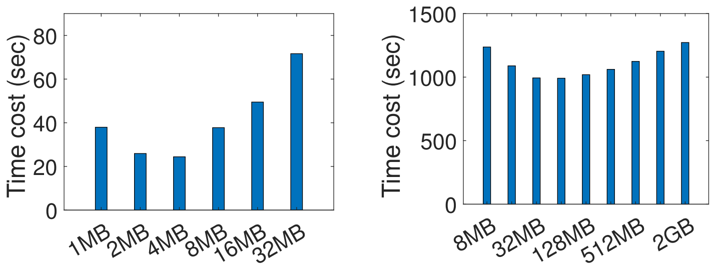
5.4.4 Efficiency of Optimization on RWR. Now we validate the efficiency of the optimization of joining short walks on RWR, by comparing GraphWalker with the iteration-based computation graph system DrunkardMob, and let both systems use the same joining method introduced in Section 4.4. We take RWR with parameter $c = 0.15$ as the study case and use many joined short RWSs to simulate a very long RWR for computing PPR [15]. We restate that the number of iterations needed in the iteration-based model depends on the longest walk of RWS, which restricts the computation efficiency. One approach to improve the efficiency is to truncate the paths at a given length L, with the cost of only a very slight loss of precision [15]. The formal out-of-core graph processing systems used this truncate-based approach by default [31, 32], so we also use the same settings for previous experiments for a fair comparison. Here, we run both of the two versions of PPR, i.e., PPR with and without truncating, for a comprehensive comparison. For each version, we run PPR with 1 single source, 100 sources, and 10,000 sources on the Twitter graph.

Figures 24(a) and 24(b) show the results of PPR with and without truncating walks, respectively. We can see that DrunkardMob saves more than half the time by truncating walks longer than 10 steps, with the cost of some precision loss. This means that in the version of PPR without truncating, the iteration-based computation model costs more than half the time to finish the last few long walks. GraphWalker costs similar time for the two versions, as it can quickly finish the long walks without any precision loss. This demonstrates that this joining method without truncating is quite suitable to cooperate with our proposed state-aware graph loading model and the asynchronous walk updating strategy, thus further demonstrating the effectiveness of GraphWalker. In addition, GraphWalker always outperforms DrunkardMob for both versions, and achieves 25× to 68× speedup and 11× to 20× speedup for PPR with and without truncating, respectively.



(a) Friendster (b) YahooWeb

Fig. 25. Performance on HDDs.



(a) PPR with 10^3 sources (b) PPR with 10^5 sources

Fig. 26. Impact of block size.

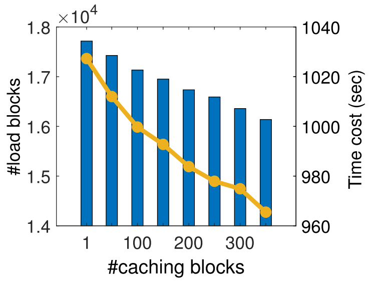
5.5 Impact of System Configurations

5.5.1 Performance on HDDs. We also study the impact of storage devices by running experiments on HDDs. Figure 25 shows the time cost of running the four algorithms we considered in this article, and we only show the results for Friendster and YahooWeb here. Since HDDs have much lower random I/O performance than SSDs, the time cost of both DrunkardMob and GraphWalker is increased. When comparing GraphWalker with DrunkardMob, we observe similar results as in the case of SSDs studied before. Precisely, GraphWalker achieves 3× to 2,305× speedup on HDDs.

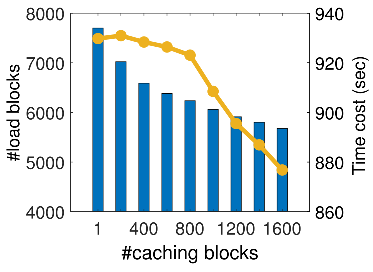
5.5.2 Impact of Block Size. In GraphWalker, block size has an impact on both I/O efficiency and walk updating rate. Specifically, smaller block size improves the I/O efficiency, whereas larger blocks can make walks move more steps for each single I/O and thus improves the walk updating rate. To study the impact of block size, we keep only one block in memory and run the PPR algorithm on CW as a study case. Here, we consider two PPR algorithms that start random walks from 1,000 and 100,000 sources, respectively. Note that the two cases are representative to denote two typical scenarios of accessing only a part of the graph or accessing most of the entire graph.

Figure 26 shows the results. We find that it necessitates an appropriate block size setting to achieve the best performance due to the preceding analyzed trade-off. The insight is that small blocks may be beneficial to lightweight tasks that require only a small number of random walks, as the I/O efficiency can get improved under this setting. In contrast, large blocks may be beneficial to heavyweight tasks that require a large number of random walks, as a large block setting increases the walk updating rate. Based on this observation, we also propose a method to set the block size, which is determined according to the number of walks (see Section 3.2).

5.5.3 Impact of Block Caching. GraphWalker proposes a walk-conscious block caching scheme, instead of using the Linux page cache. We study the efficiency of this caching scheme by varying



(a) 10^5 randomly started walks



(b) PPR with 1,000 sources

Fig. 27. Impact of block caching.

the number of cached blocks denoted by *nmblocks*. Note that larger *nmblocks* imply that less memory space is used for the page cache. Thus, varying *nmblocks* actually compares the efficiency of the walk-conscious block caching scheme with that of the Linux page cache. We run 10^5 walks of length 10, which are randomly started at different vertices, and also run a PPR algorithm with 1,000 sources on CW as study cases. Figure 27 shows the results. The rectangular bars record the number of blocks loaded from disk, and the lines refer to the time cost. We find that both the number of I/Os to load blocks from disk and the time cost always decrease as we cache more blocks in memory with the walk-conscious block caching scheme and reduce the Linux page cache size. This result implies that the walk-conscious block caching scheme is more effective than the Linux page cache.

6 RELATED WORK

Several graph systems have been proposed in recent years, and some of them develop distributed systems based on a cluster of machines so as to handle very large graphs that cannot reside on a single machine [9, 10, 17, 18, 29, 39, 41, 51, 60, 61]. However, distributed graph systems usually require an efficient graph partition and low-cost communication between machines. In addition, research efforts are made to leverage large memory in analyzing large graphs [22, 42, 50] or utilizing GPUs to accelerate the computation [30, 36, 56].

Graph processing on single machine for disk-resident graphs also has received a lot of attention. GraphChi [32] is the pioneering work, and X-Stream [48] further develops a different computation model based on edge streams. GridGraph [62] optimizes I/Os by selectively loading needed graph blocks to bypass useless graph data. DynamicShards [54] and Graphene [37] also aim to reduce the loading of useless edges by dynamically adjusting the graph partition layout. CLIP [6] and Lumos [53] improve the utilization of the loaded graph blocks so as to reduce the number of I/Os. There are also a body of works to leverage high-performance emerging devices to improve performance [12, 14, 27, 40]. The preceding systems are not designed specially for random walks, so most of them still follow the iteration-based model. Different from them, GraphWalker targets supporting massive concurrent random walks in a fast and scalable way, and its key idea is to utilize the states of walks to optimize the process of graph loading and computing so as to improve the I/O and computing efficiency.

In terms of random walk, besides developing new algorithms, such as RWR [52], FolkRank [23], and TrustWalker [24], there are some works focusing on system design. To support massive random walks on large graphs, DrunkardMob [31] proposes an encoded representation and a lightweight efficient index so as to be able to run billions of random walks on a single machine. As it follows the iteration-based model, it still suffers from the I/O deficiency problem. Different from

DrunkardMob, GraphWalker focuses on optimizing the I/O management, and it develops a new state-aware model with asynchronous walk updating to improve I/O performance. In addition, GraphWalker also allows walk states to be stored on disk, instead of putting only in memory as in DrunkardMob, so it is more scalable to run more walks on larger graphs.

Besides the single-machine out-of-core random walk systems, there are the distributed system KnightKing [58] and the in-memory system FlashMob [57], which were recently proposed and also optimized for random walks. KnightKing provides a unified framework to support various random walks and focuses on optimizing the walking process. FlashMob targets making memory accesses more sequential and regular to improve the cache and memory bandwidth utilization. Different from KnightKing and FlashMob, GraphWalker mainly targets addressing the I/O problem, and it is also more resource friendly, as it can process massive random walks on very large graphs on just a single machine.

7 CONCLUSION

In this article, we proposed GraphWalker, which is an I/O-efficient system for supporting fast and scalable random walks over large graphs on a single machine. GraphWalker carefully manages graph data and walk indexes, and optimizes I/O efficiency by using state-aware graph loading and asynchronous walk updating. Experiment results on our prototype show that GraphWalker outperforms state-of-the-art single-machine systems, and it also achieves comparable performance with a distributed graph system running on a cluster machine. In future work, we will consider extending the state-aware design idea in GraphWalker to distributed clusters so as to process massive analytic tasks in parallel.

REFERENCES

[1] Friendster. [n.d]. Home Page. http://konect.uni-koblenz.de/networks/friendster.
[2] Graph500. [n.d]. Home Page. Retrieved October 5, 2022 from https://graph500.org/.
[3] Web Data Commons. [n.d]. The 2012 Common Crawl Graph. Available at http://webdatacommons.org.
[4] ANLAB Traces. [n.d]. Twitter. Available at http://an.kaist.ac.kr/traces/WWW2010.html.
[5] Yahoo! [n.d]. Yahoo Webscope Program. Retrieved October 5, 2022 from http://webscope.sandbox.yahoo.com.
[6] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. 2017. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk I/O. In *Proceedings of USENIX ATC*.
[7] Reid Andersen, Christian Borgs, Jennifer Chayes, Uriel Feige, Abraham Flaxman, Adam Kalai, Vahab Mirrokni, and Moshe Tennenholtz. 2008. Trust-based recommendation systems: An axiomatic approach. In *Proceedings of WWW*. ACM, New York, NY.
[8] Ziv Bar-Yossef, Alexander Berg, Steve Chien, Jittat Fakcharoenphol, and Dror Weitz. 2000. Approximating aggregate queries about web pages via random walks. In *Proceedings of VLDB*.
[9] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: An efficient task-oriented graph mining system. In *Proceedings of EuroSys*. ACM, New York, NY.
[10] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of EuroSys*. ACM, New York, NY.
[11] Wei Chen, Yajun Wang, and Siyu Yang. 2009. Efficient influence maximization in social networks. In *Proceedings of KDD*. ACM, New York, NY.
[12] Disa Mhembere Da Zheng, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. Flash-Graph: Processing billion-node graphs on an array of commodity SSDs. In *Proceedings of FAST*.
[13] Souvik Debnath, Niloy Ganguly, and Pabitra Mitra. 2008. Feature weighting in content based recommendation system using social network analysis. In *Proceedings of WWW*. ACM, New York, NY.
[14] Nima Elyasi, Changho Choi, and Anand Sivasubramaniam. 2019. Large-scale graph processing on emerging storage devices. In *Proceedings of FAST*.
[15] 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.
[16] Hector Gonzalez, Jiawei Han, Xiaolei Li, Margaret Myslinska, and John Paul Sondag. 2007. Adaptive fastest path computation on a road network: A traffic mining approach. In *Proceedings of VLDB*.

[17] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of OSDI*.

[18] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of OSDI*.

[19] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of KDD*. ACM, New York, NY, 855–864.

[20] Taher H. Haveliwala. 2002. Topic-sensitive pagerank. In *Proceedings of WWW*. ACM, New York, NY.

[21] 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.

[22] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: A DSL for easy and efficient graph analysis. *ACM SIGPLAN Notices* 47, 4 (2012), 349–362.

[23] Andreas Hotho, Robert Jäschke, Christoph Schmitz, Gerd Stumme, and Klaus-Dieter Althoff. 2006. FolkRank: A ranking algorithm for folksonomies. In *Proceedings of LWA*.

[24] Mohsen Jamali and Martin Ester. 2009. TrustWalker: A random walk model for combining trust-based and ttem-based recommendation. In *Proceedings of KDD*. ACM, New York, NY, 397–406.

[25] Glen Jeh and Jennifer Widom. 2002. SimRank: A measure of structural-context similarity. In *Proceedings of KDD*. ACM, New York, NY, 538–543.

[26] Glen Jeh and Jennifer Widom. 2003. Scaling personalized web search. In *Proceedings of WWW*. ACM, New York, NY.

[27] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. 2018. GraFBoost: Using accelerated flash storage for external graph analytics. In *Proceedings of ISCA*. IEEE, Los Alamitos, CA.

[28] David Kempe, Jon Kleinberg, and Éva Tardos. 2003. Maximizing the spread of influence through a social network. In *Proceedings of KDD*. ACM, New York, NY, 137–146.

[29] Arijit Khan, Gustavo Segovia, and Donald Kossmann. 2018. On smart query routing: For distributed graph querying with decoupled storage. In *Proceedings of USENIX ATC*.

[30] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: Vertex-centric graph processing on GPUs. In *Proceedings of HPDC*. ACM, New York, NY, 239–252.

[31] Aapo Kyrola. 2013. DrunkardMob: Billions of random walks on just a PC. In *Proceedings of RecSys*. ACM, New York, NY.

[32] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of OSDI*.

[33] A. N. Langville and C. D. Meyer. 2004. Deeper inside PageRank. *Internet Mathematics* 1, 3 (2004), 335–380.

[34] Chul-Ho Lee, Xin Xu, and Do Young Eun. 2012. Beyond random walk and Metropolis-Hastings samplers: Why you should not backtrack for unbiased graph sampling. In *Proceedings of SIGMETRICS*.

[35] Rong-Hua Li, Jeffrey Xu Yu, Xin Huang, and Hong Cheng. 2014. Random-walk domination in large graphs. In *Proceedings of ICDE*. IEEE, Los Alamitos, CA.

[36] Hang Liu and H. Howie Huang. 2015. Enterprise: Breadth-first graph traversal on GPUs. In *Proceedings of SC*. IEEE, Los Alamitos, CA.

[37] Hang Liu and H. Howie Huang. 2017. Graphene: Fine-grained IO management for graph computing. In *Proceedings of FAST*.

[38] Christian Lochert, Hannes Hartenstein, Jing Tian, Holger Fussler, Dagmar Hermann, and Martin Mauve. 2003. A routing strategy for vehicular ad hoc networks in city environments. In *Proceedings of IV*. IEEE, Los Alamitos, CA.

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

[40] 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 EuroSys*. ACM, New York, NY.

[41] 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 SIGMOD*. ACM, New York, NY.

[42] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of SOSP*. ACM, New York, NY.

[43] Larry Page. 1998. *The PageRank Citation Ranking: Bring Order to the Web*. Technical Report. Stanford University.

[44] Jia-Yu Pan, Hyung-Jeong Yang, Christos Faloutsos, and Pinar Duygulu. 2004. Automatic multimedia cross-modal correlation discovery. In *Proceedings of KDD*. ACM, New York, NY, 653–658.

[45] Nataša Pržulj. 2007. Biological network comparison using graphlet degree distribution. *Bioinformatics* 23, 2 (2007), e177–e183.

[46] Nataša Pržulj, Derek G. Corneil, and Igor Jurisica. 2004. Modeling interactome: Scale-free or geometric? *Bioinformatics* 20, 18 (2004), 3508–3515.

[47] Bruno Ribeiro and Don Towsley. 2010. Estimating and sampling graphs with multidimensional random walks. In *Proceedings of SIGCOMM*.

[48] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of SOSP*. ACM, New York, NY.

[49] Paat Rusmevichientong, David M. Pennock, Steve Lawrence, and C. Lee Giles. 2001. Methods for sampling pages uniformly from the World Wide Web. In *Proceedings of AAAI*.

[50] Julian Shun and Guy E. Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. *ACM SIGPLAN Notices* 48, 8 (2013), 135–146.

[51] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: A system for distributed graph mining. In *Proceedings of SOSP*. ACM, New York, NY.

[52] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. 2006. Fast random walk with restart and its applications. In *Proceedings of ICDM*. IEEE, Los Alamitos, CA.

[53] Keval Vora. 2019. LUMOS: Dependency-driven disk-based graph processing. In *Proceedings of USENIX ATC*.

[54] Keval Vora, Guoqing (Harry) Xu, and Rajiv Gupta. 2016. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *Proceedings of USENIX ATC*.

[55] Rui Wang, Min Lv, Zhiyong Wu, Yongkun Li, and Yinlong Xu. 2019. Fast graph centrality computation via sampling: A case study of influence maximisation over OSNs. *International Journal of High Performance Computing and Networking* 14, 1 (2019), 92–101.

[56] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. *ACM SIGPLAN Notices* 51, 8 (2016), Article 11, 12 pages.

[57] Ke Yang, Xiaosong Ma, Saravanan Thirumuruganathan, Kang Chen, and Yongwei Wu. 2021. Random walks on huge graphs at cache efficiency. In *Proceedings of SOSP*. ACM, New York, NY, 311–326.

[58] 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 SOSP*. ACM, New York, NY.

[59] Pengpeng Zhao, Yongkun Li, Hong Xie, Zhiyong Wu, Yinlong Xu, and John C. S. Lui. 2017. Measuring and maximizing influence via random walk in social activity networks. In *Proceedings of DASFAA*. 323–338.

[60] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *Proceedings of OSDI*.

[61] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2020. LiveGraph: A transactional graph storage system with purely sequential adjacency list scans. *Proceedings of the VLDB Endowment* 13 (2020), 1020–1034.

[62] 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 USENIX ATC*.

Received 30 August 2021; revised 19 January 2022; accepted 25 April 2022