GraphWalker: An I/O-Efficient and Resource-Friendly Graph Analytic System for Fast and Scalable Random Walks

Paper ID: 792

Abstract

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 paper, we propose GraphWalker, an I/O-efficient graph system for random walks by deploying a novel *state-aware I/O model* with *asynchronous walk updating*. GraphWalker is efficient to handle very large disk-resident graphs consisting of hundreds of billions of edges with only a single commodity machine, and it is also scalable to run tens of billions of random walks with thousands of steps long. Experiments on our prototype system show that GraphWalker can achieve more than an order of magnitude speedup when running a large amount of long random walks when compared with DrunkardMob, which is tailored for random walk based on the classical system GraphChi, as well as two state-of-the-art single-machine graph systems, Graphene and GraFSoft. Furthermore, comparing with the most recent distributed system KnightKing, which optimizes for random walks and runs on cluster machines, GraphWalker achieves comparable performance with only a single machine, thereby making it a more cost-effective alternative.

1 Introduction

Graph analysis has received a lot of attentions in recent years due to its wide use in many applications, e.g., page ranking [25, 42], social networks [11, 27], recommendation systems [16, 37], navigation systems [7, 13], etc. 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 kind of disk-resident large graphs is very time-consuming because of the massive random I/Os.

To improve the performance of analyzing large graphs on a single-machine, many out-of-core graph processing systems are proposed, such as GraphChi [31], X-Stream [46], GridGraph [58], FlashGraph [12], ODS [52], CLIP [6], Mosaic [39], Graphene [36], GraFBoost [26], V-Part [14] and LUMOS [51]. 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 [31]. 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.

We observe that these graph systems with the iteration-based model are only good for some applications which require to iteratively process the entire graph data, but they cannot efficiently support random walks, which have been proven to be efficient to analyze large graphs [8, 15, 20, 24, 30, 33, 34, 45, 47]. The major limitations are three folds. 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 walk state, e.g., the current vertex at which the walk stays, and just sequentially loads subgraphs into memory for analysis, so it results in very low I/O utilization when processing such subgraphs containing only few walks. Second, as the iteration-based model ensures a synchronized analysis, all walks move exactly one step in each iteration, in which all needed subgraphs are successively loaded. Some recently loaded subgraphs will be evicted out even if they still be useful to update more walks. 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. Lastly, 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.

To efficiently support parallel random walks, various design efforts are also made in recent years. One typical system is DrunkardMob [30], 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, e.g., it costs 2.3 hours to run one billion random walks with ten-step long on a medium-scale graph YahooWeb [5], and it is even unable to run random walks on very large graphs like CrawlWeb [3] due to its large memory consumption. KnightKing [55] 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 develop GraphWalker, which is an I/O-efficient and resource-friendly single-machine graph system. GraphWalker mainly focuses on improving the I/O efficiency by developing a state-aware I/O model with asynchronous walk updating. It also utilizes a lightweight block-centric walk management scheme to improve memory efficiency. More importantly, GraphWalker is also a resource-friendly solution which can process very large graphs on a single machine. In summary, our main contributions are as follows.

- We develop a novel *state-aware I/O model*, which leverages the state of each random walk to selectively load the graph block with the most walks from disk into memory, so as to improve the I/O utilization. We also propose an efficient blocking strategy to carefully manage graph data and adopt a walk-conscious caching scheme to improve cache efficiency.

- We develop an *asynchronous walk updating scheme*, which allows each walk to move as many steps as possible until it reaches the boundary of the currently loaded subgraph. Thus, we can fully utilize the loaded subgraph and greatly accelerate the progress of random walks. To address the straggler issues caused by asynchronous update, we also employ a probabilistic approach to balance the progress of each walk.

- We propose a *lightweight block-centric indexing scheme* to manage walk states and adopt a fixed-length walk buffering strategy to reduce the memory cost for recording walk states. We also develop a disk-based walk management scheme and use asynchronous batched I/Os to write walk states back to disk so as to support running massive random walks in parallel on huge graphs.

- We implement a prototype 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 [30], as well as two state-of-the-art single-machine graph systems, Graphene [36] and GraFSoft [26]. GraphWalker is also scalable to run massive random walks on huge graphs. Furthermore, its performance is even comparable with the state-of-the-art distributed random walk system KnightKing [55] running on a cluster of machines. This shows that GraphWalker is more resource friendly as it achieves similar performance with less resource demand.

The rest of this paper is organized as follows. In §2, we first introduce iteration-based graph systems, and analyze their limitations of supporting random walks. In §3 and §4, we present the design details of GraphWalker and its evaluation results. Finally, §5 reviews related work and §6 concludes.

2 Background and Motivation

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

2.1 Iteration-based Graph Computation

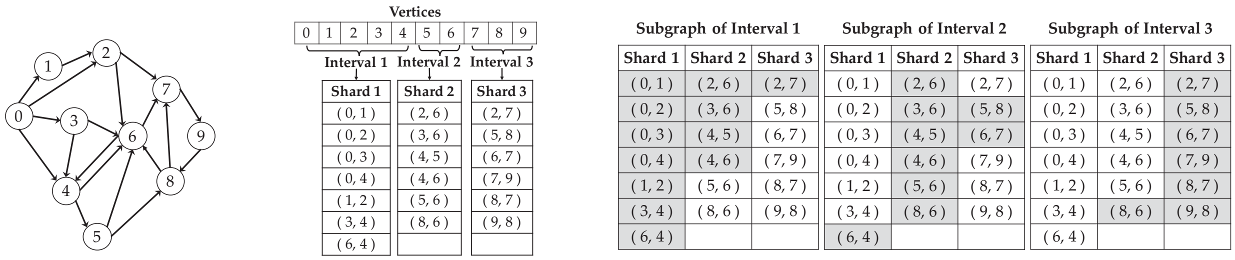
For simplicity, we take GraphChi [31], the pioneering single-machine iteration-based graph system, as an example to illustrate its key idea. GraphChi splits all vertices into disjoint intervals and associates each interval with a *shard*, which stores all the edges whose destination vertices lie in this 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).

To perform analysis, GraphChi loads all subgraphs iteratively by using the parallel sliding window (PSW), 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 conduct computation. This way, GraphChi transfers random accesses to a series of sequential accesses and greatly improves the performance of disk-resident graph processing. We like to point out that this iteration-based model is very representative and it is widely used in many graph systems [12, 14, 26, 36, 46, 52, 58].

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 a large number of random walks [15, 34, 53, 56], and it is also very common to have multiple tasks concurrently running on the same graph and thus induce to massive parallel walks. When supporting massive parallel random walks, graph systems with the iteration-based model suffer from several limitations, e.g., low I/O utilization and low walk updating rate, as well as high memory cost for managing walks. In the following, we analyze these limitations in details.

Limitation 1: Low I/O utilization. First of all, the iteration-based model leads to low I/O utilization 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. 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. 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 utilization. Some



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

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 2

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 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)		

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

Figure 1: Storage and I/O model in GraphChi

recent works like Graphene [36] adopt an *on-demand* I/O model to manage graph data with fine-grained blocks and skip loading blocks not containing any walks, but it does not fully address the low I/O utilization problem. 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.

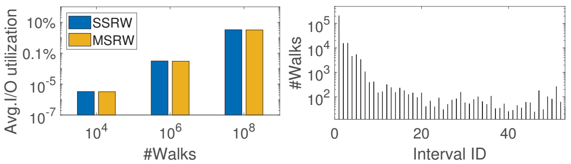
We also run experiments to demonstrate the skewed walk distribution and low I/O utilization. We use GraphChi to run 10^4, 10^6 and 10^8 walks of length ten on the Friendster graph consisting of 68.3 million vertices, and consider starting random walks from a single source (SSRW) or multiple random sources (MSRW). Please refer to §4.1 for 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 utilization is very low, especially for the case of small number of walks. Figure 2(b) further shows the distribution of walks over blocks after four iterations when running 10^6 walks started from a single source. We find that walks are distributed over all blocks after only four steps, and the distribution is heavily skewed.

In our GraphWalker, we propose a *state-aware I/O model*, which loads graph blocks by considering the states of walks. Precisely, it always chooses to load the block with the maximum number of walks so as to make more walks get updated by using an I/O. Our experiment results show that GraphWalker brings 2× to 4× I/O utilization (see §4.2.3).

Limitation 2: Low walk updating rate. The iteration-based model also leads to 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 above. 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 total 10^9 steps to walk, so the updating rate is as low as 10^{-6}. We also 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% walks still remain in the first block, and they could move more steps in the current iteration, so it results in the low walk updating rate. Recently, CLIP [6] proposes a *re-entry* way to reuse the loaded data to improve the I/O and computing efficiency. However, when running random walks, re-entry brings extra accesses to the whole subgraph, and it is also difficult to decide the number of times to re-entry because walks in the subgraph have different progresses.

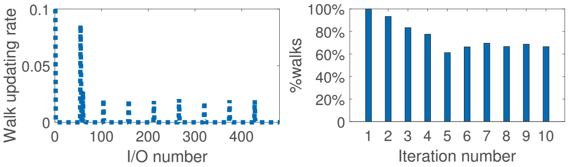
In GraphWalker, we propose an *asynchronous walk updating scheme* to allow walks to move as many steps as possible within the currently loaded subgraph. This scheme relaxes the strong synchronization of all walks, which is not necessary for random walk based algorithms. With our asynchronous walk updating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

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, e.g., GraphChi associates each edge with a dynamic array to store the walks currently traveling through the edge. This design incurs large memory cost, e.g., it needs at least 26.4 GB space to store only the walk indexes 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, 36], but it also incurs high memory cost, e.g., 5.6 GB to store walk indexes for YahooWeb.



(a) I/O utilization (b) Distribution of walks

Figure 2: I/O utilization and analysis



(a) Walk updating rate (b) Walks in the 1^{st} interval

Figure 3: Walk updating rate and analysis

DrunkardMob encodes a walk into a 32-bit or 64-bit representation and puts walks of adjacent 128 vertices into the same walk buffer to reduce the total size of walk indexes. It reduces the size of walk 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 many dynamic arrays for large graphs, e.g., 11.2 million for YahooWeb, it causes frequent memory re-allocation, which not only introduces memory fragments, but also brings extra time cost and limits the scale of graphs that could be analyzed. Second, DrunkardMob keeps all walks in memory, and it incurs very high cost to flush them to disk as there are too many files related to the walk indexes. Thus, the scale of walks could be analyzed by DrunkardMob is also limited.

In our GraphWalker, we adopt a block-centric method to manage walk data, so it greatly reduces the size of walk indexes. We also use fixed-length buffers to cache walks so as to avoid frequent memory re-allocation. With our lightweight scheme, both the scale of graphs and the number of walks that can be handled are no longer limited by memory capacity.

3 Design of GraphWalker

In this section, we first introduce the main idea of GraphWalker, which is an I/O-efficient and resource-friendly design targeted for random walks on single machine. We then present the details of its key design techniques, including state-aware graph loading, asynchronous walk updating, and lightweight walk management.

3.1 Main Idea

We target for 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* which leverages the state of each walk, e.g., the current vertex at which the walk stays. Briefly speaking, unlike the iteration-based model which 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 utilization and low walk updating rate problems can be efficiently addressed.

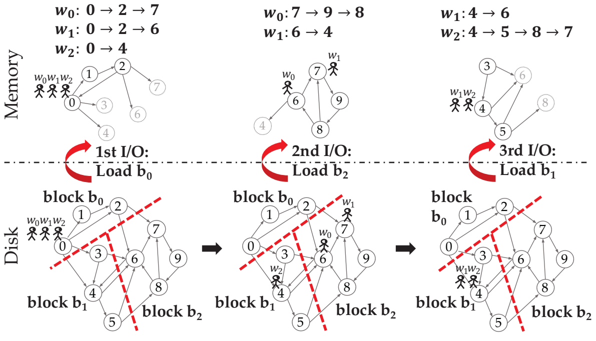


Figure 4: Main idea of the state-aware model

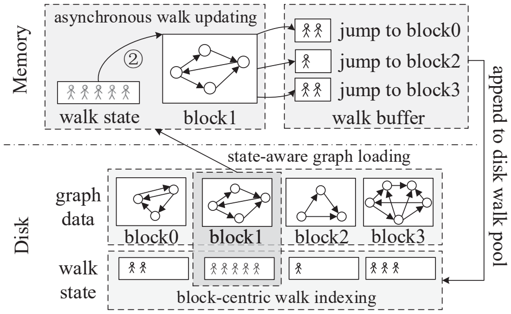


Figure 5: Overall design of GraphWalker

To further illustrate the above idea and analyze its benefits, we still consider the example graph in Figure 1(a). Suppose that we have to run three random walks which start at node 0 and have to move four steps. Figure 4 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 the 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 which 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 the remaining two walks are in block b_1, so we load b_1 into memory, and all walks can be finished. Note that three I/Os are required in this example. However, for the iteration-based model, it needs 12 I/Os, because it uses four iterations, and generates three I/Os in each iteration.

Remark. We like to emphasize that the state-aware model is different from the on-demand I/O model proposed in Graphene [36], which manages graph data with fine-grained blocks and loads only the blocks containing walks (i.e., demands) but skips the blocks without containing any walks. Thus, this method still follows the iteration-based model. Besides, even if there is only one walk in a block, it still has to load the block into memory for analysis.

Based on the above idea, we develop an I/O-efficient graph system, GraphWalker, which supports fast and scalable random walks. GraphWalker mainly consists of three parts: (1) *State-aware graph loading,* (2) *Asynchronous walk updating,* and (3) *Block-centric walk management*. The overall design of GraphWalker is also illustrated in Figure 5. In the following subsections, we present its design in details.

3.2 State-Aware Graph Loading

Graph data organization and partition. GraphWalker manages 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*

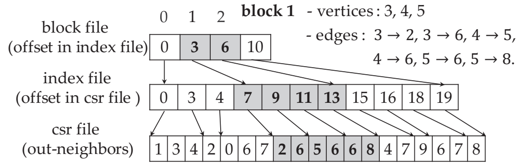


Figure 6: Graph data organization

file. GraphWalker partitions a graph into blocks according to vertex IDs. Specifically, we sequentially add vertices and their out-edges into a block according to the ascending order of their IDs until the data volume in the block exceeds a predefined *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 for random walks. Besides, this lightweight graph data organization decreases the storage cost of each subgraph, and thus reduces the time cost of graph loading. As GraphWalker partitions a graph by simply reading through the *index file* once to record the beginning vertex of each block, it is also flexible to adjust block size for different applications.

For setting the block size, we find that a trade-off exists. That is, using smaller blocks can avoid loading more data which are not needed for updating random walks, while using larger blocks can have more walks getting updated in each I/O. Besides, 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 utilization can get improved under this setting. In contrast, heavyweight tasks with a large number of walks prefer a large block size as large block size can increase the walk updating rate. Based on this understanding, we use an empirical analysis (see §4.4), 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 one 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.

Graph loading and block caching. GraphWalker converts the graph format and partitions graph blocks in preprocessing phase. During the phase of running random walks, GraphWalker chooses a graph block to load into memory according to the the states of walks, and in particular, it loads the block containing the largest number of walks. After finishing analysis over the loaded graph block, it then chooses another block to load in the same way.

To ease the impact of block size and improve cache efficiency, GraphWalker also enables 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 near future. Thus, the graph loading process with block caching works as follows. As illustrated in Figure 7, we first select a candidate block based on the state-aware model, to load this block, we check whether it is cached in memory or not. If it is already in memory, then we directly access memory to perform analysis. Otherwise, we load it from disk, and also evict out the block in memory containing the fewest walks if the cache is full. The maximum number of blocks cached in memory depends on the usable memory size.

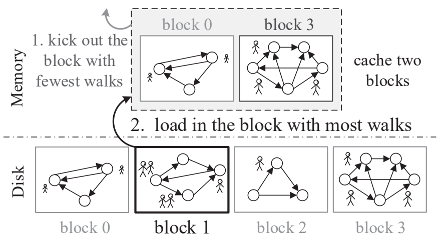

Memory
1. kick out the block with fewest walks
block 0
block 3
cache two blocks
2. load in the block with most walks
Disk
block 0
block 1
block 2
block 3

Figure 7: State-aware graph loading with block caching

We emphasize that this walk-conscious block caching scheme differs from 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, page cache manages data in memory at a *page* granularity, while 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 conventional page cache scheme.

3.3 Asynchronous Walk Updating

Note that in iteration-based systems, after loading a graph block, each walk in the loaded subgraph walks only one step, which induces to 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. 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.

However, the re-entering scheme may cause local straggler problem. That is, many walks are able to move one step at the first time when the graph block was just loaded, and as the number of re-entries increases, most walks may reach the boundary of the subgraph, and only few walks remain in the subgraph, and they cost multiple re-entries to finish. Our experiments show that the last 20% of walks in a block may cost 60% of re-entries. These re-entries have very low utilization and cost a lot of time. We also find that simply stopping walking over the currently loaded subgraph after certain re-entries cannot address the local straggler problem either, and it does not reduce the completion time as the last few walks still remain in the subgraph and we still need to re-load it with extra I/Os to finish the walks.

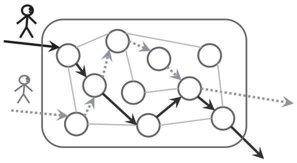


Figure 8: Asynchronous walk updating in parallel

To further improve the I/O utilization 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 above. Figure 8 shows an example of processing two walks within the same graph block. To accelerate the computation, we also use multi-threading 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 utilization and reduce the number of I/Os.

However, the state-aware model may lead to the *global straggler problem*. That is, some walks may move very fast and make a large progress as the graph data they needed can always be satisfied, while some other walks may move very slow as they may be trapped in some coldblocks which are not loaded into memory for a long time. As a result, GraphWalker can quickly complete most of the walks, but takes a long time to finish the remaining few walks. Our experiments also show that there are often a few walks incurring nearly half of the total I/Os.

To address the global straggler problem, we introduce a probabilistic approach into the state-aware graph loading process in GraphWalker. The idea is to give stragglers a chance to move some steps such that they can catch up the progress of most walks. Specifically, every time when 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. 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 for some cases.

3.4 Block-Centric Walk Management

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. We record each walk with 64 bits. The number of bits allocated for each variable is shown in Figure 9. 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.

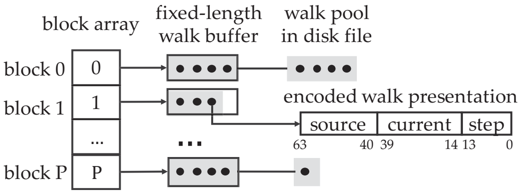


Figure 9: Block-centric walk management

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 which are currently in the block, referring to Figure 9. We implement each walk pool as a fixed-length buffer, which stores at most 1024 walks by default, so as to avoid dynamic memory allocation cost. When there are more than 1024 walks in a block, we flush them to disk and store them as a file called *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 small. For example, for running one billion walks in YahooWeb with 1.4 billion nodes, GraphWalker costs only 800 KB if it uses 100 graph blocks. However, DrunkardMob costs more than 4 GB to record one billion walks as each walk uses at least four bytes, and it also needs 45 MB to store the dynamic arrays.

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 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 finish computing with the loaded graph block, we clear the current walk pool and sum up the walks in both walk buffer and walk pool file of each block so as to get the walk states.

With this lightweight walk management, we save a lot of memory cost for storing walk states, thus it is able to support massive concurrent walks. Besides, the fixed-length walk buffering strategy turns many small I/Os for updating walk states into several large I/Os, which largely reduces the I/O cost for providing persistent storage of walk states.

4 Evaluation

GraphWalker aims for providing fast and scalable random walks, so we take DrunkardMob [30], the state-of-the-art single-machine random walk specific graph system, as a baseline for performance comparison. Note that both GraphWalker and DrunkardMob are implemented based on GraphChi, the pioneering and classical graph system. Besides, there are also a number of single-machine graph systems, which further optimize the system performance from different aspects. 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 [36] and GraFSoft [26]. To further validate its scalability, we compare it with the most recent distributed graph system optimized for random walks, i.e., KnightKing [55].

4.1 Experiment Settings

Testbed. All experiments are performed on a Dell Power Edge R730 machine with 64GB memory and 24 Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz processors. The entire graph data are stored on a 3.2TB RAID-0 consisting of seven 500GB SamSung 860 SSDs if we do not state specifically. We also study the performance of GraphWalker on HDDs. For the distributed system KnightKing [55], it is run on an 8-node cluster with 10Gbps Ethernet inter connection, each node is equipped with two 8-core Intel Xeon E5-2620 v4 processor with 20MB L3 cache and 64GB 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 can not be entirely put into the memory in our testbed, and CW is the largest web corpus available in public.

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

Table 1: Statistics of Datasets

Graph algorithms. Besides directly evaluating the performance of running random walks, we also consider the following four common random walk based algorithms.

- *Random Walk Domination (RWD)* [34]. 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)* [43, 44]. We use a special graphlet, triangle, as a study case. We randomly start 100 thousand random walks of length four to estimate the ratio of triangles in the graph.
- *Personalized PageRank (PPR)* [19, 25]. We focus on the random walk based version [15]. We simulate 2000 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 [15].
- *SimRank (SR)* [24]. It 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 2000 random walks of length 11, similar to the setting in [24].

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

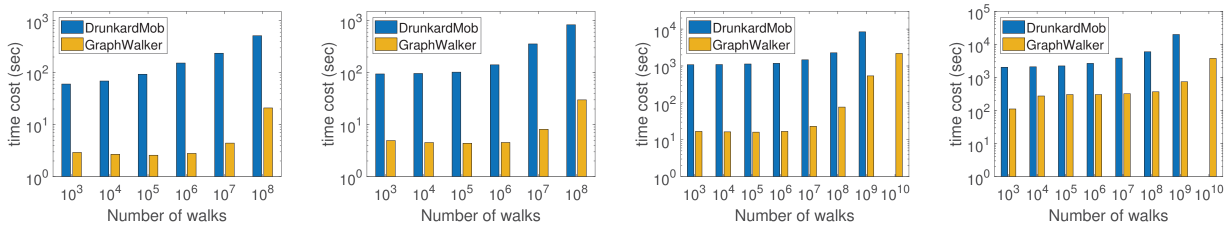
4.2 Comparison with RW-Specific Systems

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 usually require to start certain number of random walks with 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. Finally, we justify the improvement achieved by GraphWalker by using micro-benchmark results.

4.2.1 Performance Study in Entire Design Space

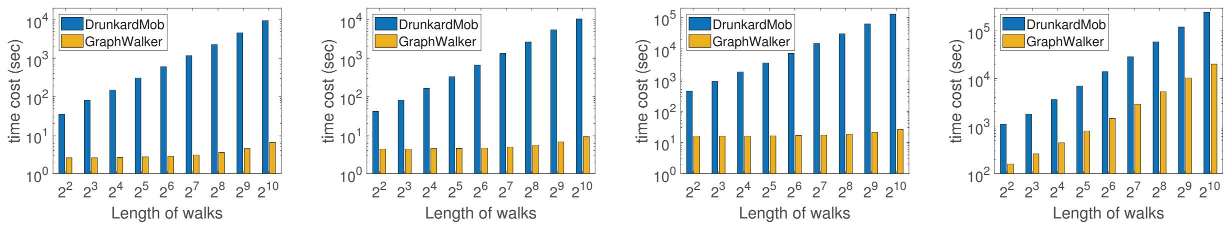
We first show the results by fixing the walk length to ten, but varying the number of walks from 10^3 to 10^{10}, as depicted in Figure 10. In each figure, the x-axis indicates the number of walks configured in each experiment, and the y-axis shows the time needed to finish running all these walks. First, 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, while GraphWalker takes only 17.8 seconds. That is, GraphWalker achieves 70× speedup. In general, GraphWalker achieves 16× to 70× speedup under all settings. In addition, for the case when the number of walks is not too large, then I/O cost is a dominate factor, so the total time of running different number 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. However, DrunkardMob even fails to run 10^{10} walks on large graphs, due to the out-of-memory error, so we do not show the results of DrunkardMob



(a) Twitter (b) Friendster (c) YahooWeb (d) Kron30

Figure 10: Performance of random walks with different number of walks



(a) Twitter (b) Friendster (c) YahooWeb (d) Kron30

Figure 11: Performance of random walks with different walk lengths

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 CrawlWeb, 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: (1) DrunkardMob keeps all walk states in memory, so it's hard to support massive walks. (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 its 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 CrawlWeb, even for running tens of billions of random walks, e.g., GraphWalker finishes running 10^{10} walks on CrawlWeb within around one 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 11. First, we can see that GraphWalker always run 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, while 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 small I/O cost. For very large graphs which can not 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 7× to 10× speedup even for Kron30. This experiment also demonstrates the scalability of GraphWalker in supporting long random walks which have thousands of steps.

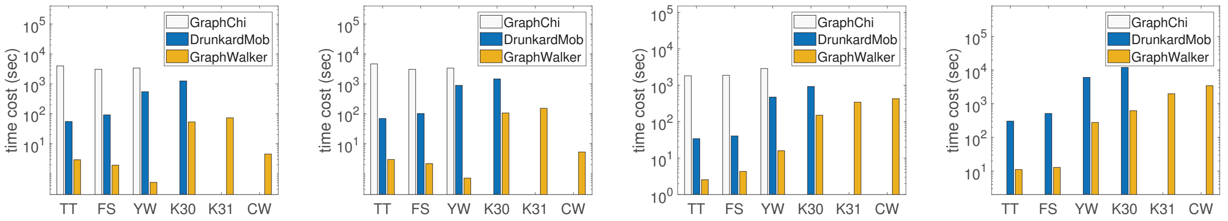
4.2.2 Performance of Random Walk based Algorithms

We now evaluate the performance of the four common random walk based algorithms described in §4.1. Since both GraphWalker and DrunkardMob are implemented based on GraphChi, to also show the improvement achieved by the RW-specific optimizations in DrunkardMob, we include the results of GraphChi in this experiment. From Figure 12, we can see that DrunkardMob indeed outperforms GraphChi, and GraphWalker still improves over DrunkardMob. In general, GraphWalker achieves 9× to 48× speedup upon 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 random walks. However, DrunkardMob needs to iteratively scan the entire graph and also updates walks in a synchronized manner, so it has a very low I/O utilization and takes very long time.

We like to point out that DrunkardMob again fails to handle the two largest graphs. Besides, GraphChi even fails to run all algorithms over Kron30 and also fails to run the RWD algorithm over all graphs due to the same reason explained in §4.2.1, 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.

4.2.3 Micro-benchmarks

Recall that the inefficiency of existing systems in running random walks mainly come from the iteration-based I/O



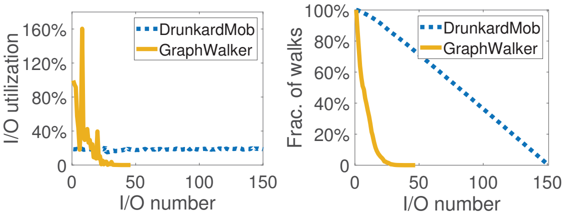
(a) Personalized PageRank (b) SimRank (c) Graphlet (d) Random Walk Domination

Figure 12: Performance of random walk based algorithms

model, and thus they suffer from low I/O utilization and low walk updating rate (refer to §2). To better understand why GraphWalker could significantly improve the overall performance as presented in the last subsection, we further consider these two micro-benchmarks to show how the state-aware I/O model in GraphWalker address the limitations. We also show the time cost breakdown to see how GraphWalker improves the performance of each part along the random walk process. *We only show the results of running RWD algorithm on YahooWeb, and results are similar for other settings.*

I/O utilization. I/O utilization is defined as the edge usage amount for updating walks divided by the total number of edges loaded by one I/O. Note that an edge may be reused by different walks, so we summing the total used times of all edges. Thus, the I/O utilization 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 six, so the total number of I/Os required by DrunkardMob is 150. We can see that the I/O utilization is only around 20% as shown in Figure 13(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%-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 utilization of GraphWalker is between 40% to 80%, which is 2× to 4× compared to that of DrunkardMob.

Walk updating rate. Now we study the walk updating rate, shown in Figure 13(b). Recall that DrunkardMob updates 50 million steps per I/O on average and costs 150 I/Os to finish the computation. While 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, 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 §3.3), 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.



(a) I/O utilization (b) Walk updating rate

Figure 13: I/O utilization and walk updating rate (DrunkardMob needs 150 I/Os and GraphWalker only needs 46 I/Os)

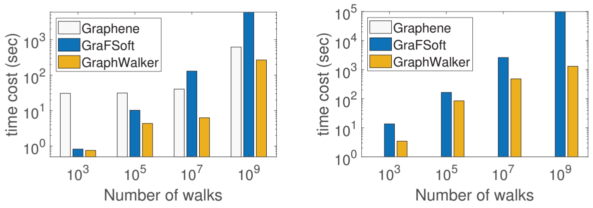
Time cost breakdown. To better understand the effect of the design optimizations in GraphWalker, we also show the time cost breakdown in Table 2. Note that in the whole execution procedure, there are three key operations: (1) graph loading, which loads graph blocks into memory with disk I/Os, (2) walk 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. Besides, the three operations are proceeded in an interleaved way, so we aggregate the total time of executing each operation. From the results, we can see that GraphWalker outperforms DrunkardMob in all aspects. We point out that the improvement is achieved by the integration of multiple design optimizations, which all contribute to the high efficiency of GraphWalker. For example, the improvement of graph loading mainly comes from the state-aware graph loading scheme with the lightweight data organization and block caching policy, but it also benefit from the asynchronous walk updating strategy which finally results in the reduction of I/Os.

4.3 Comparison with State-of-the-art Systems

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, huge page support to reduce TLB miss, etc. These optimizations are not

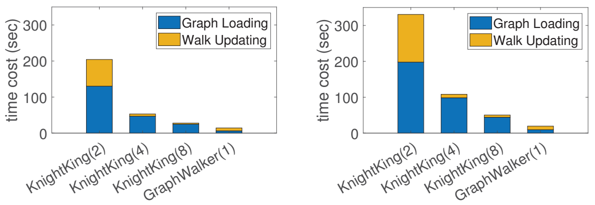
Time cost (s)	DrunkardMob	GraphWalker	Speedup
Graph Loading	1005	47	21×
Walk Updating	3029	214	14×
Walk Persisting	1056	16	66×
Total Runtime	5110	278	18×

Table 2: Time cost breakdown



(a) Friendster (b) CrawlWeb

Figure 14: Comparison with Graphene and GraFSoft



(a) Twitter (b) Friendster

Figure 15: Comparison with KnightKing

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 [36] and GraFBoost [26]. 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 above mentioned design optimizations.

In this experiment, we focus on the case of running random walks starting from a single source due to page limit. We fix the walk length as ten and vary the number of walks. Note that Graphene is a semi-external system which stores graph data on disk while keeps 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 CrawlWeb. We observe similar results for other graphs.

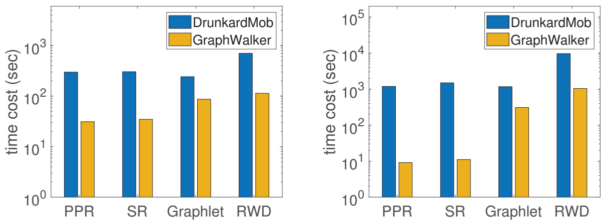
The results are shown in Figure 14, and we can have several conclusions. First, GraphWalker consistently outperforms Graphene even though Graphene is a semi-external system which does not require I/Os to write back walk states, and it achieves up to 19× speedup. More importantly, GraphWalker is also scalable to run huge amount of walks, as well as process extremely large graphs, while Graphene fails to run in these cases due to its high memory cost caused by the semi-external design. Second, compared to GraFSoft, GraphWalker achieves significant improvement, especially when the number of random walks becomes large. For example, when running one billion random walks on CrawlWeb, GraphWalker spends only 21.8 minutes while GraFSoft can not even complete the task within 24 hours. That is, GraphWalker achieves at least 40× speedup. More importantly, as we increase the number of walks, the increase of time for GraphWalker is *sub-linear* and much slower than that of GraFSoft, this further demonstrates the scalability of GraphWalker in supporting huge amount of random walks.

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 [55], which is the most recent distributed graph system optimized for random walks. In the interest of space, we focus on a random walk based algorithm which is also used in KnightKing, i.e., PPR. Specifically, it starts one walk at each vertex, and each walk terminates with probability t in each step. We set $t = 0.15$, which is a very common setting in various applications [15, 32]. 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. As KnightKing uses a cluster of eight machines for evaluation in its paper, to enable cross-validation, we also use eight machines at most, and 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 be undirected as in KnightKing.

Figure 15 shows the results, 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 processing I/Os, i.e., loading graph blocks. This is because KnightKing mainly focuses on optimizing the computing efficiency, but not disk I/Os. Note that the results of the computing time in this experiment are consistent with those in the KnightKing paper. In contrast, GraphWalker mainly targets for 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 incudes the time for walk index persistency. We also see that GraphWalker achieves comparable performance even compared with KnightKing running on eight machines. Even more, for the largest graph CrawlWeb, 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 more resource-friendly alternative.

4.4 Impact of System Configurations

Performance on HDDs. We also study the impact of storage devices by running experiments on hard disk drives (HDDs). Figure 16 shows the time cost of running the four algorithms we considered in this paper, and we only show the results for Friendster and YahooWeb due to page limit. Since HDDs have much lower random I/O performance than SSDs, the time cost of both DrunkardMob and GraphWalker are increased. When comparing GraphWalker with DrunkardMob, we observe similar results as in the case of SSDs stud-



(a) Friendster

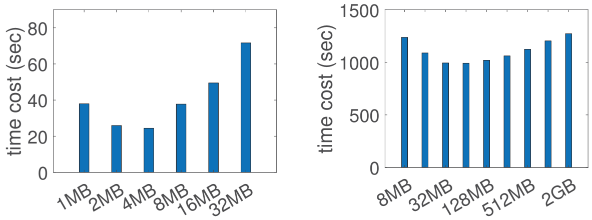
(b) YahooWeb

Figure 16: Performance on HDDs

ied before. Precisely, GraphWalker achieves 3× to 135× speedup under different settings.

Impact of block size. In GraphWalker, block size has an impact on both I/O utilization and walk updating rate. Specifically, smaller block size improves the I/O utilization, while 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 CrawlWeb as a study case. Here, we consider two PPR algorithms which start random walks from 1000 and 100000 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 17 shows the results. We find that it necessitates an appropriate block size setting to achieve the best performance due to the above analyzed tradeoff. The insight is that small blocks may be beneficial to lightweight tasks which require only a small number of random walks, as the I/O utilization can get improved under this setting. In contrast, large blocks may be beneficial to heavyweight tasks which require a large number of random walks, as 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 §3.2).



(a) PPR with 10^3 sources

(b) PPR with 10^5 sources

Figure 17: Impact of block size

5 Related Work

A number of 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 which can not reside on a single machine [9, 10, 17, 18, 28, 38, 40, 49, 57]. However, distributed graph systems usually require efficient graph partition and low-cost communication between machines. Besides, research efforts are also made to leverage large memory in analyzing large graphs [21, 41, 48] or utilizing GPUs to accelerate the computation [29, 35, 54].

On the other hand, graph processing on single machine for disk-resident graphs also receives a lot of attentions. GraphChi [31] was the pioneering work in this line of research to reduce random I/Os. Then various design optimizations are also proposed, such as optimizing I/O performance [36, 46, 52, 58], improving computation efficiency [6, 51], and leveraging high-performance emerging devices for storage or computation [12, 14, 26, 39], etc. We point out that the above graph systems are not designed specifically for random walks, and most of them adopt the iteration-based model. Differently, GraphWalker targets for supporting fast and scalable random walks, and it utilizes the state-aware model to optimize the I/O and computing efficiency.

In terms of random walk, besides developing new algorithms, such as random walk with restart [50], FolkRank [22] and TrustWalker [23], etc., there are also some works focusing on system design. To support massive random walks on large graphs, DrunkardMob [30] 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 single-machine random walk systems, there is also a distributed system KnightKing [55], which is recently proposed and also optimized for random walks. It provides a unified framework to support various random walks and focuses on optimizing the walking process without addressing disk I/Os. Different from KnightKing, GraphWalker mainly targets for addressing the I/O problem, and it is also more resource friendly as it can process massive random walks on large graphs on just a single machine.

6 Conclusion

In this paper, 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 distributed graph system running on a cluster machine. In the future work, we will consider to extend the state-aware design idea in GraphWalker to distributed clusters so as to process massive analytic tasks in parallel.

References

[1] Friendster. http://konect.uni-koblenz.de/networks/friendster.

[2] Graph500. https://graph500.org/.

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

[4] Twitter. http://an.kaist.ac.kr/traces/WWW2010.html.

[5] Yahoo Webscope Program. http://webscope.sandbox.yahoo.com.

[6] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. Squeezing Out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O. In *USENIX ATC*, 2017.

[7] Reid Andersen, Christian Borgs, Jennifer Chayes, Uriel Feige, Abraham Flaxman, Adam Kalai, Vahab Mirrokni, and Moshe Tennenholtz. Trust-based Recommendation Systems: an Axiomatic Approach. In *ACM WWW*, 2008.

[8] Ziv Bar-Yossef, Alexander Berg, Steve Chien, Jittat Fakcharoenphol, and Dror Weitz. Approximating Aggregate Queries about Web Pages via Random Walks. In *VLDB*, 2000.

[9] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-Miner: an Efficient Task-Oriented Graph Mining System. In *ACN EuroSys*, 2018.

[10] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *EuroSys*. ACM, 2015.

[11] Wei Chen, Yajun Wang, and Siyu Yang. Efficient Influence Maximization in Social Networks. In *ACM SIGKDD*, 2009.

[12] Disa Mhembere Da Zheng, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. FlashGraph: Processing Billion-node Graphs on an Array of Commodity SSDs. In *USENIX FAST*, 2015.

[13] Souvik Debnath, Niloy Ganguly, and Pabitra Mitra. Feature Weighting in Content Based Recommendation System Using Social Network Analysis. In *ACM WWW*, 2008.

[14] Nima Elyasi, Changho Choi, and Anand Sivasubramaniam. Large-Scale Graph Processing on Emerging Storage Devices. In *FAST*. USENIX, 2019.

[15] Dániel Fogaras, Balázs Rácz, Károly Csalogány, and Tamás Sarlós. Towards Scaling Fully Personalized Pagerank: Algorithms, Lower Bounds, and Experiments. *Internet Mathematics*, 2(3):333–358, 2005.

[16] Hector Gonzalez, Jiawei Han, Xiaolei Li, Margaret Myslinska, and John Paul Sondag. Adaptive Fastest Path Computation on a Road Network: a Traffic Mining Approach. In *VLDB*, 2007.

[17] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*. USENIX, 2012.

[18] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI*. USENIX, 2014.

[19] Taher H Haveliwala. Topic-Sensitive Pagerank. In *WWW*. ACM, 2002.

[20] Monika R Henzinger, Allan Heydon, Michael Mitzenmacher, and Marc Najork. Measuring Index Quality using Random Walks on the Web. *Computer Networks*, 31(11):1291–1303, 1999.

[21] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: a DSL for Easy and Efficient Graph Analysis. In *Proceedings of the ACM SIGARCH Computer Architecture News*, 2012.

[22] Andreas Hotho, Robert Jäschke, Christoph Schmitz, Gerd Stumme, and Klaus-Dieter Althoff. Folkrank: A Ranking Algorithm for Folksonomies. In *Lwa*, 2006.

[23] Mohsen Jamali and Martin Ester. Trustwalker: a Random Walk Model for Combining Trust-Based and Ttem-Based Recommendation. In *ACM SIGKDD*, 2009.

[24] Glen Jeh and Jennifer Widom. SimRank: a Measure of Structural-context Similarity. In *ACM SIGKDD*, 2002.

[25] Glen Jeh and Jennifer Widom. Scaling Personalized Web Search. In *WWW*. ACM, 2003.

[26] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, et al. GraFBoost: Using Accelerated Flash Storage for External Graph Analytics. In *ISCA*. IEEE, 2018.

[27] David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the Spread of Influence Through a Social Network. In *ACM SIGKDD*, 2003.

[28] Arijit Khan, Gustavo Segovia, and Donald Kossmann. On Smart Query Routing: for Distributed Graph Querying with Decoupled Storage. In *ATC*. USENIX, 2018.

[29] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 2014.

[30] Aapo Kyrola. Drunkardmob: Billions of Random Walks on Just a PC. In *ACM RecSys*, 2013.

[31] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. Graphchi: Large-scale Graph Computation on Just a PC. In *OSDI*, 2012.

[32] Meyer C D. Langville A N. Deeper inside Pagerank. In *Internet Mathematics*, 2004.

[33] Chul-Ho Lee, Xin Xu, and Do Young Eun. Beyond Random Walk and Metropolis-hastings Samplers: Why You Should Not Backtrack for Unbiased Graph Sampling. In *SIGMETRICS*, 2012.

[34] Rong-Hua Li, Jeffrey Xu Yu, Xin Huang, and Hong Cheng. Random-walk Domination in Large Graphs. In *ICDE*. IEEE, 2014.

[35] Hang Liu and H Howie Huang. Enterprise: Breadth-first graph traversal on GPUs. In *Proceedings of the SC-International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015.

[36] Hang Liu and H Howie Huang. Graphene: Fine-Grained IO Management for Graph Computing. In *FAST*. USENIX, 2017.

[37] Christian Lochert, Hannes Hartenstein, Jing Tian, Holger Fussler, Dagmar Hermann, and Martin Mauve. A Routing Strategy for Vehicular Ad Hoc Networks in City Environments. In *Proceedings of the Intelligent vehicles symposium*. IEEE, 2003.

[38] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: a Framework for Machine Learning and Data Mining in the Cloud. *VLDB*, 2012.

[39] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a Trillion-edge Graph on a Single Machine. In *EuroSys*. ACM, 2017.

[40] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *SIGMOD*. ACM, 2010.

[41] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *SOSP*. ACM, 2013.

[42] Larry Page. The Pagerank Citation Ranking: Bring Order to the Web. *Technical report*, 1998.

[43] Nataša Pržulj. Biological Network Comparison Using Graphlet Degree Distribution. *Bioinformatics*, 23(2):e177–e183, 2007.

[44] Nataša Pržulj, Derek G Corneil, and Igor Jurisica. Modeling Interactome: Scale-Free or Geometric? *Bioinformatics*, 20(18):3508–3515, 2004.

[45] Bruno Ribeiro and Don Towsley. Estimating and Sampling Graphs with Multidimensional Random Walks. In *SIGCOMM*, 2010.

[46] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric Graph Processing Using Streaming Partitions. In *SOSP*. ACM, 2013.

[47] Paat Rusmevichientong, David M Pennock, Steve Lawrence, and C Lee Giles. Methods for Sampling Pages Uniformly from the World Wide Web. In *Proceedings of the AAAI Fall Symposium on Using Uncertainty Within Computation*, 2001.

[48] Julian Shun and Guy E Blelloch. Ligra: a Lightweight Graph Processing Framework for Shared Memory. In *ACM SIGPLAN*, 2013.

[49] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. Arabesque: A System for Distributed Graph Mining. In *SOSP*. ACM, 2015.

[50] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. Fast Random Walk with Restart and Its Applications. In *ICDM*. IEEE, 2006.

[51] Keval Vora. LUMOS: Dependency-Driven Disk-based Graph Processing. In *ATC*. USENIX, 2019.

[52] Keval Vora, Guoqing (Harry) Xu, and Rajiv Gupta. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *USENIX ATC*, 2016.

[53] Rui Wang, Min Lv, Zhiyong Wu, Yongkun Li, and Yinlong Xu. Fast Graph Centrality Computation via Sampling: a Case Study of Influence Maximisation over OSNs. *International Journal of High Performance Computing and Networking*, 14(1):92–101, 2019.

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

[55] Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. KnightKing: A Fast Distributed Graph Random Walk Engine. In *SOSP*. ACM, 2019.

[56] Pengpeng Zhao, Yongkun Li, Hong Xie, Zhiyong Wu, Yinlong Xu, and John CS Lui. Measuring and Maximizing Influence via Random Walk in Social Activity Networks. In *International Conference on Database Systems for Advanced Applications*, pages 323–338. Springer, 2017.

[57] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *OSDI*. USENIX, 2016.

[58] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *USENIX ATC*, 2015.