



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

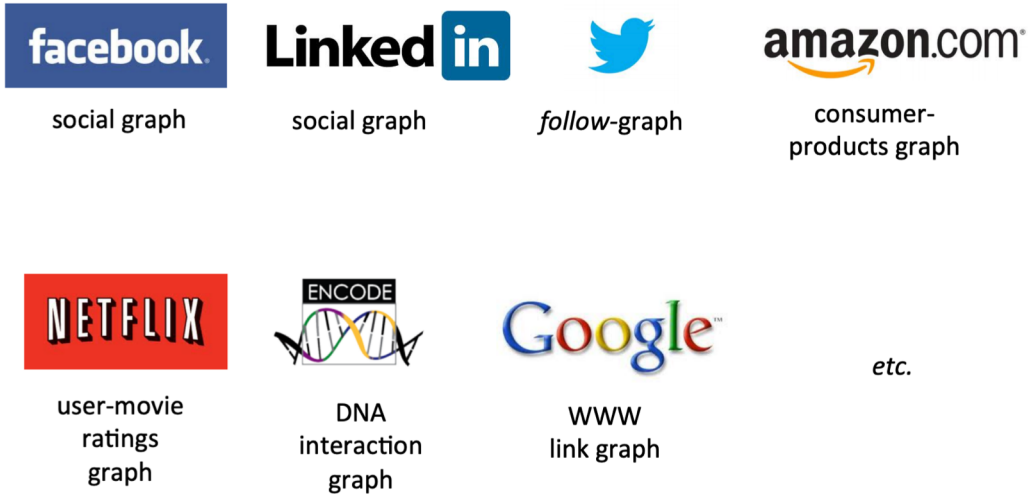
Rui Wang and Yongkun Li, *University of Science and Technology of China*; Hong Xie,
Chongqing University; Yinlong Xu, *University of Science and Technology of China*;
John C. S. Lui, *The Chinese University of Hong Kong*

<https://www.usenix.org/conference/atc20/presentation/wang-rui>

Presenter: Tao Sun


Background

BigData with *Structure*: BigGraph

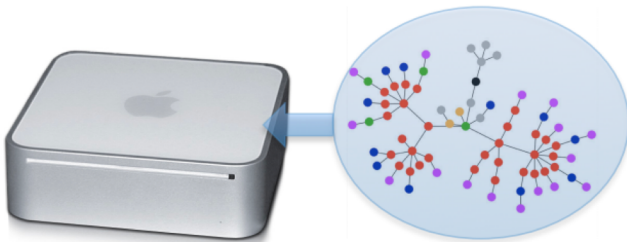


Big Graphs != Big Data

Data size:

 140 billion connections ≈ 1 TB

Not a problem!



Traditional graph systems mainly use the iteration-based model

- Partition the graph into subgraphs
- Store each subgraph as a block on a disk
- In each iteration, blocks are sequentially loaded into memory
- Turn random I/Os into serial I/Os
- Synchronize over all blocks in each iteration

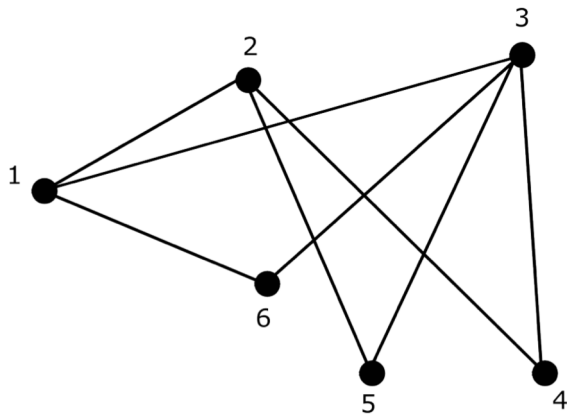
Random Walks

Random walks have been proven to be efficient to analyze large graphs

Let G be a graph or digraph with the additional assumption that if G is a digraph, then $\deg^+(v) > 0$ for every vertex v . Now consider an object placed at vertex v_j . At each stage the object must move to an adjacent vertex. The probability that it moves to the vertex v_i is $\frac{1}{\deg(v_j)}$ or $\frac{1}{\deg^+(v_j)}$ if (v_j, v_i) is an edge on G and G is a graph or digraph, respectively. Otherwise the probability is 0. Therefore if we define

$$m_{ij} = \begin{cases} \frac{1}{\deg(v_j)} & \text{if } (v_j, v_i) \text{ is an edge in the graph } G \\ \frac{1}{\deg^+(v_j)} & \text{if } (v_j, v_i) \text{ is an edge in the digraph } G \\ 0 & \text{otherwise} \end{cases}$$

M^5 which represents corresponds to random walks of length 5 along this graph:



$$M = \begin{pmatrix} 0 & 1/3 & 1/4 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 & 1/2 & 0 \\ 1/3 & 0 & 0 & 1/2 & 1/2 & 1/2 \\ 0 & 1/3 & 1/4 & 0 & 0 & 0 \\ 0 & 1/3 & 1/4 & 0 & 0 & 0 \\ 1/3 & 0 & 1/4 & 0 & 0 & 0 \end{pmatrix}$$

$$M^5 = \begin{pmatrix} 0.1192 & 0.2858 & 0.2582 & 0.0911 & 0.0911 & 0.1940 \\ 0.2858 & 0.0370 & 0.0723 & 0.3395 & 0.3395 & 0.1921 \\ 0.3442 & 0.0965 & 0.1273 & 0.4063 & 0.4063 & 0.2717 \\ 0.0608 & 0.2263 & 0.2032 & 0.0243 & 0.0243 & 0.1144 \\ 0.0608 & 0.2263 & 0.2032 & 0.0243 & 0.0243 & 0.1144 \\ 0.1293 & 0.1281 & 0.1359 & 0.1144 & 0.1144 & 0.1134 \end{pmatrix}$$

Iteration-based model

Current graph systems with the iteration-based model cannot efficiently support random walks. The major limitations are three folds.

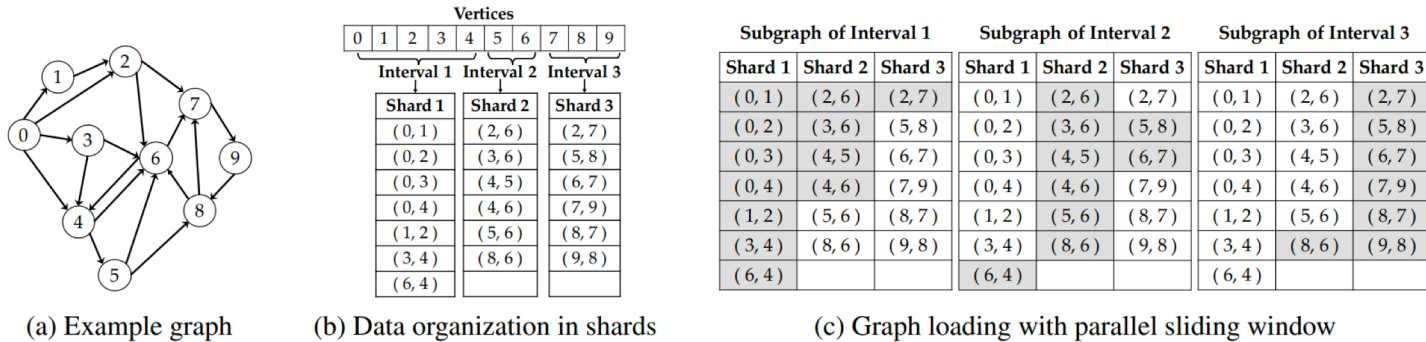
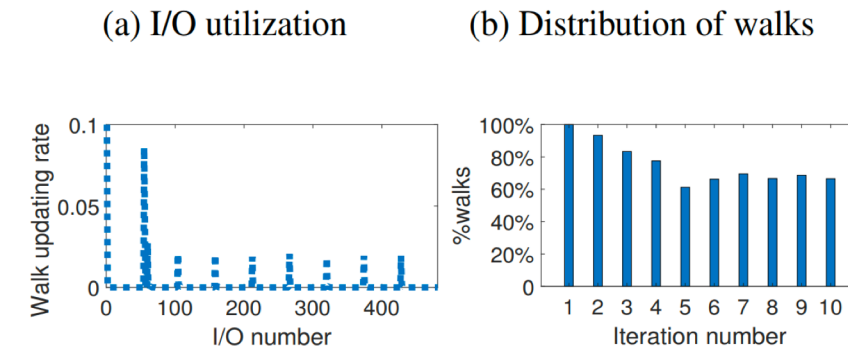
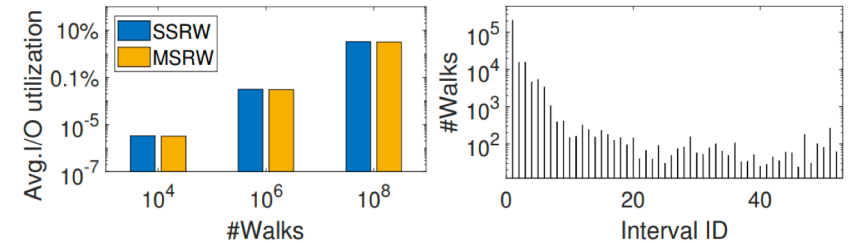


Figure 1: Storage and I/O model in GraphChi

- **Low I/O utilization** : Unaware of the unevenly scattered walk states, just sequentially loads all needed subgraphs into memory
- **Low walk updating rate**: Ensuring a synchronized analysis, all walks move exactly one step in each iteration
- **High memory cost**: Existing graph systems usually use massive dynamic arrays to record the walks. However, this indexing design requires large memory space



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

GraphWalker

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 graph system.

- **state-aware I/O model**
- **asynchronous walk updating scheme**
- **lightweight block-centric indexing scheme**

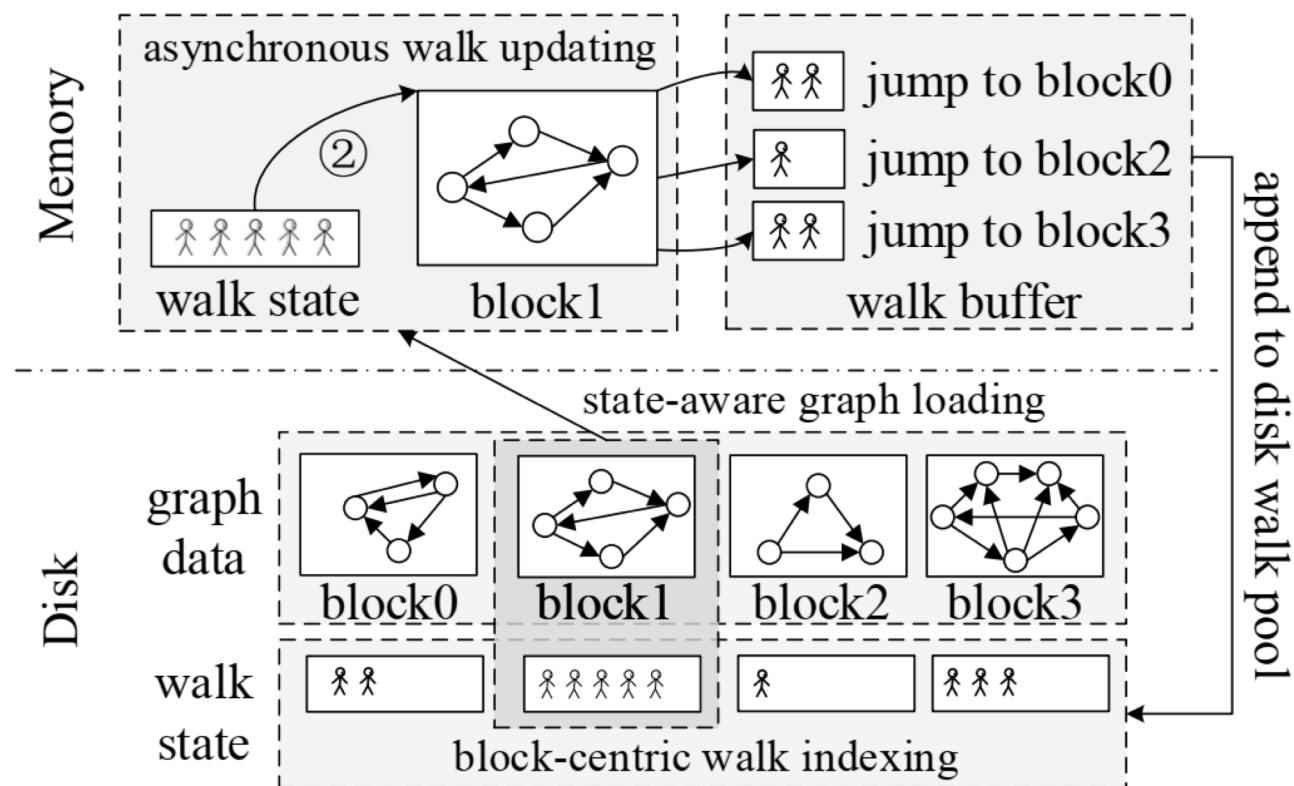
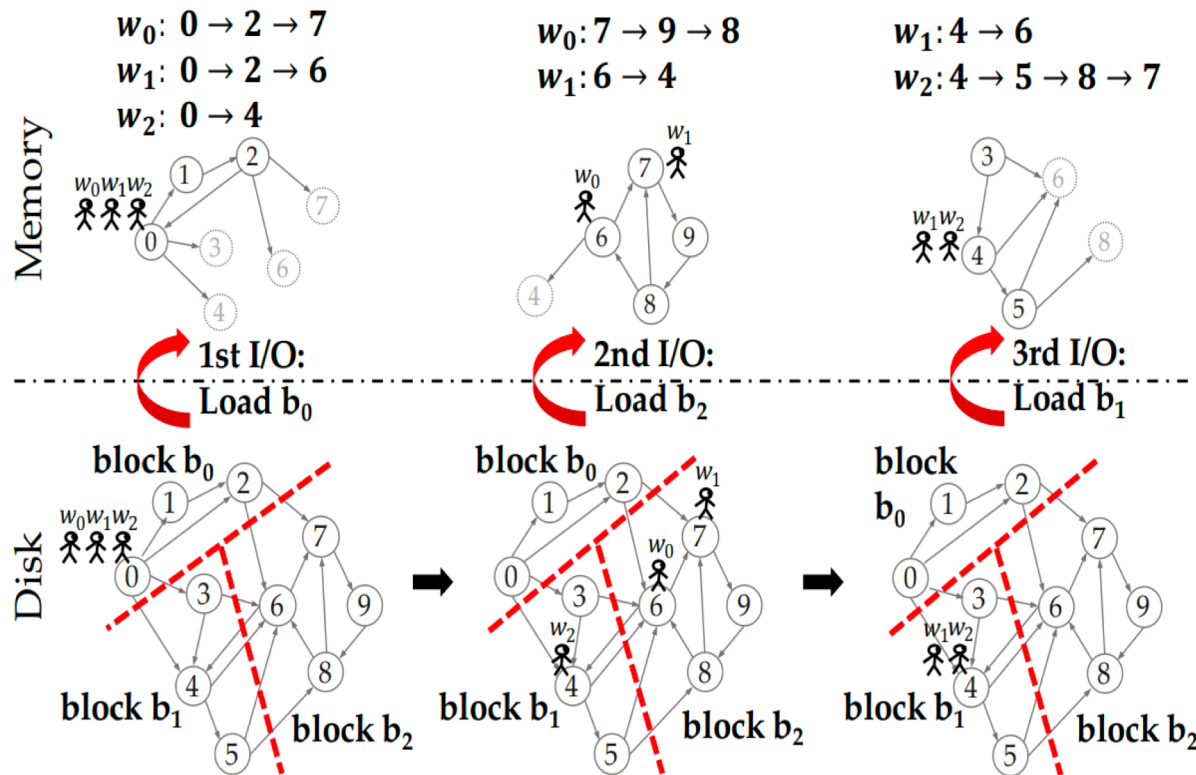


Figure 5: Overall design of GraphWalker

Design idea

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.



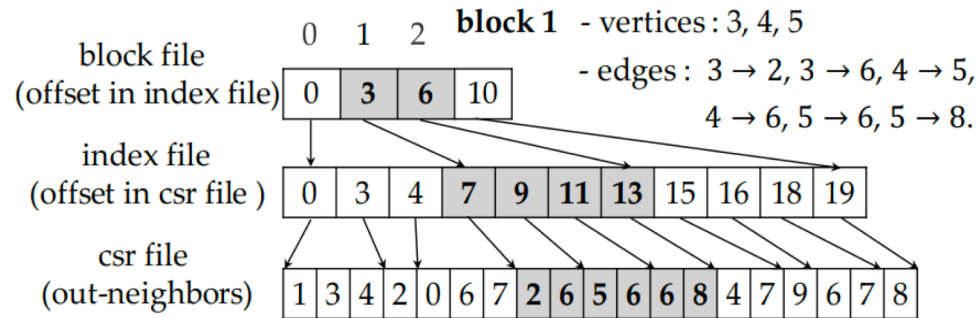
Suppose that we have to run three random walks which start at node 0 and have to move four steps.

- 1) in the first I/O, graph block b_0 is loaded into memory. With the loaded graph block b_0 , walk w_0 and w_1 move two steps, and w_2 moves only one step
- 2) 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.
- 3) both the remaining two walks are in block b_1 , so we load b_1 into memory, and all walks can be finished.

Note only 3 I/Os are required v.s. 12 I/Os in iteration-based model

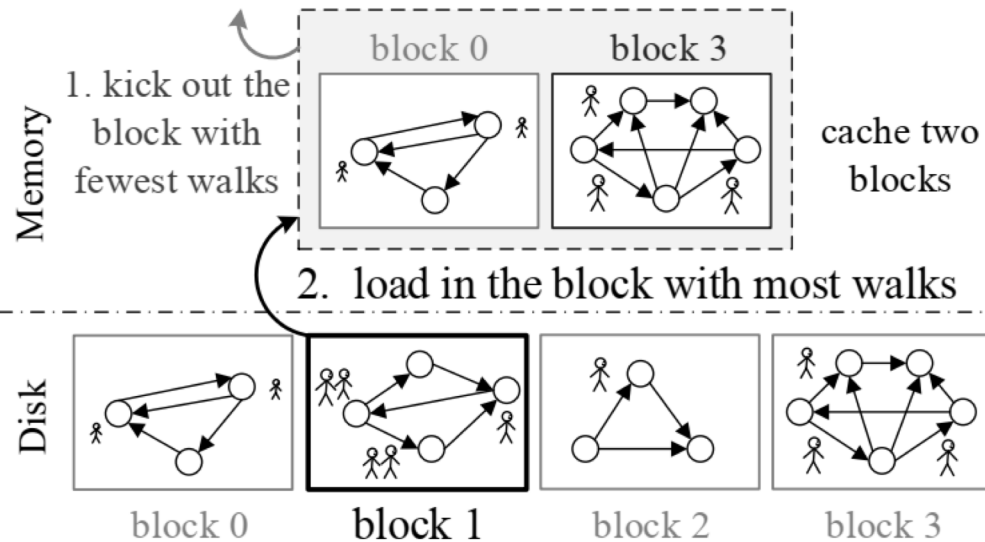
State aware graph loading

Data organization



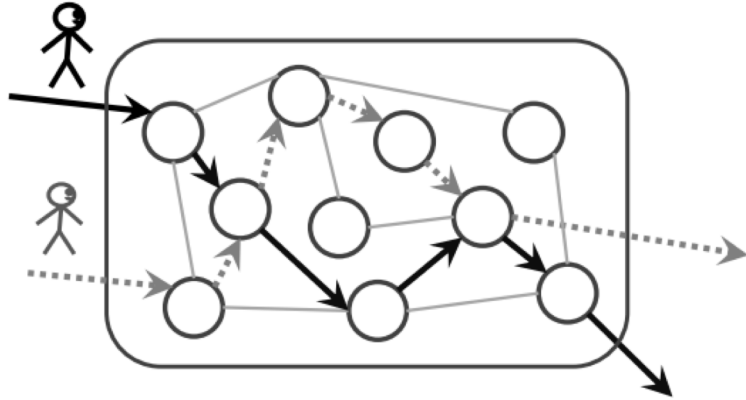
- A block file to record the vertices in ascending order
- An index file to record the beginning position of each vertex in the csr file.
- A Compressed Sparse Row (CSR) format, which sequentially stores the out neighbors of vertices

Graph loading with block caching



- Select a candidate block based on the state-aware model,
- To load this block, check whether it is cached in memory or not. If it is already in memory, then we directly access memory to perform analysis.
- Otherwise, load it from disk, and also evict out the block in memory containing the fewest walks if the cache is full.

Asynchronous walk updating



- 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

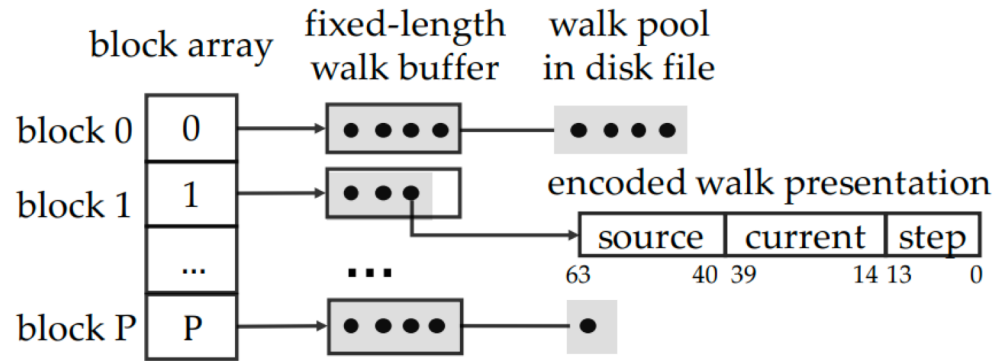
Asynchronous walk updating in parallel

Global Straggler problem

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 slowly as they may be trapped in some cold blocks which are not loaded into memory for a long time.

Solution: 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

Block-centric walk management



For each graph block, we use a walk pool to record the walks which are currently in the block. 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.

Figure 9: Block-centric walk 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 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 update the walk states.

Performance valuation

Datasets

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

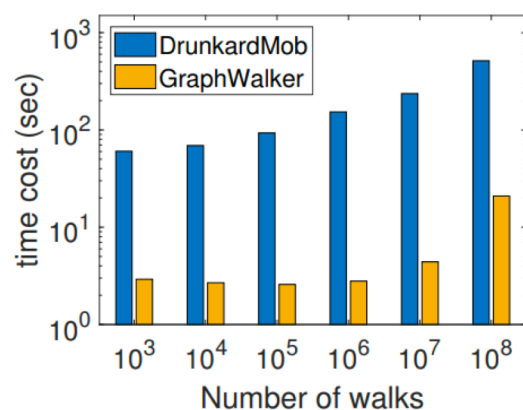
We validate the efficiency of GraphWalker by comparing it with DrunkardMob, the state-of-the-art single-machine system that is specially optimized for random walk.

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

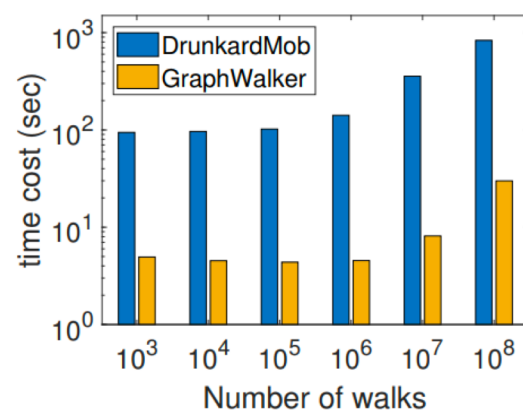
- *Random Walk Domination (RWD)* [27]. We start one walk of length six from each vertex in the graph to find a vertex set which has the maximum influence diffusion.
- *Graphlet Concentration (Graphlet)* [34, 35]. We use a special graphlet, triangle, as a study case. We randomly start 100 thousand random walks of length four to estimate the ratio of triangles in the graph.
- *Personalized PageRank (PPR)* [12]. 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.
- *SimRank (SR)* [19]. We start 2000 random walks of length 11 respectively from the query pair vertices to compute the expected meeting time,

Performance in entire design space

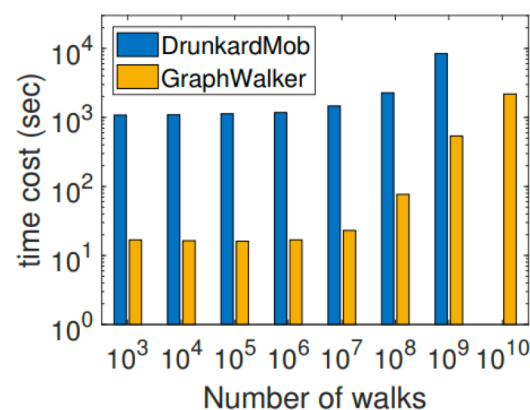
We first show the results by fixing the walk length to 10, but varying the number of walks from 10^3 to 10^{10} , as depicted in Figure 10. In general, GraphWalker achieves 16 x to 70 x speedup under all settings.



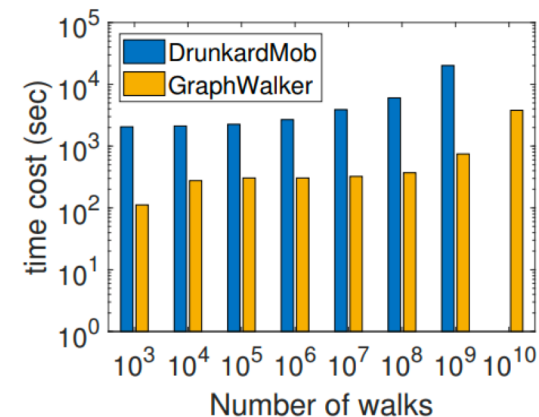
(a) Twitter



(b) Friendster



(c) YahooWeb



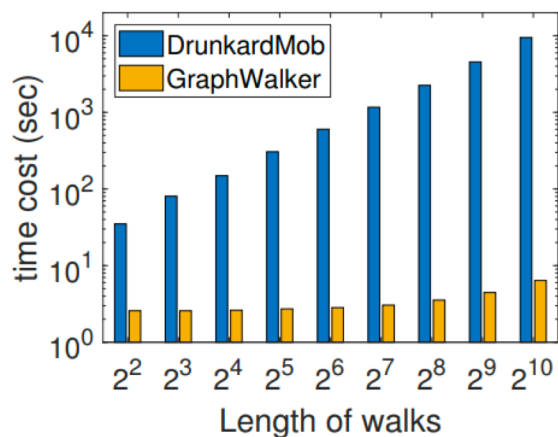
(d) Kron30

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

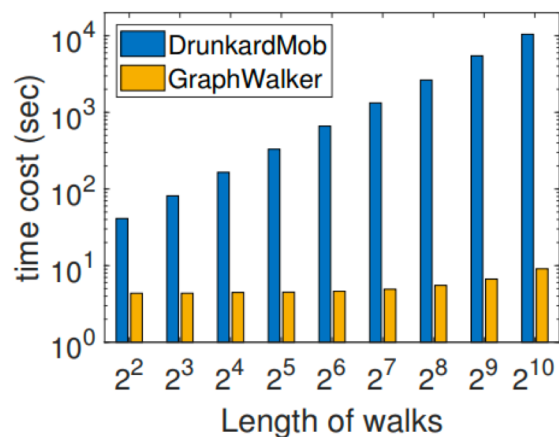
One attractive feature of GraphWalker we like to highlight is its scalability. We point out that even for running tens of billions of random walks on large graphs, GraphWalker can still finish within a reasonable time. However, DrunkardMob even fails to run 10^{10} walks on large graphs, due to the out-of-memory error

Performance in entire design space

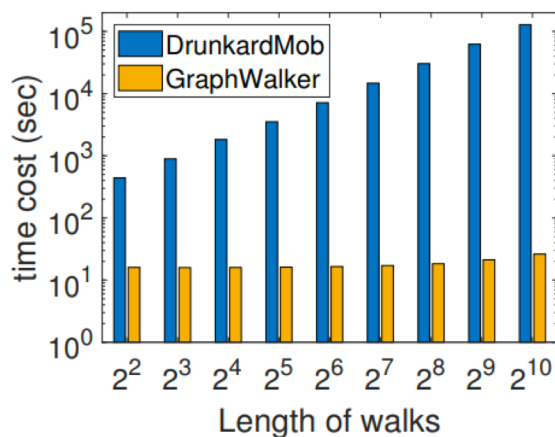
We also evaluate the performance by varying the walk length. Here we fix the number of walks as 10^5 and vary the length of each walk from 2^2 to 2^{10} . The results are shown in Figure 11. First, we can see that GraphWalker is always much faster than DrunkardMob, and it achieves even more than three orders of magnitude in the best case



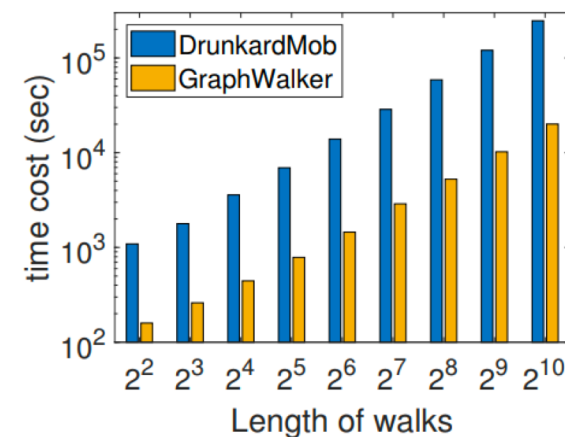
(a) Twitter



(b) Friendster



(c) YahooWeb

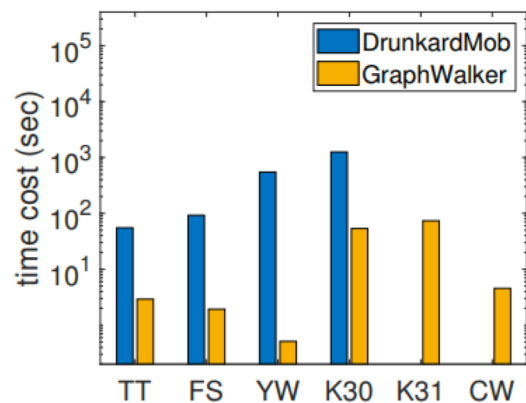


(d) Kron30

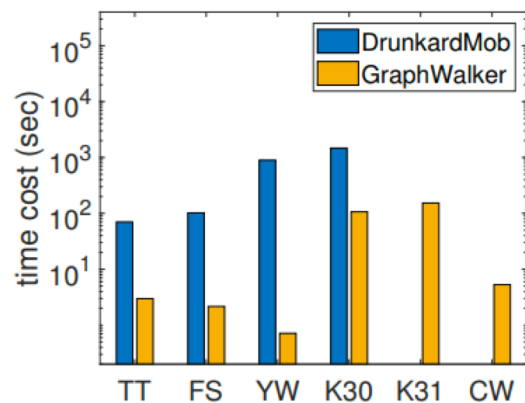
Figure 11: Performance of random walks with different walk lengths by fixing the number of walks as 10^5 .

Performance in random walk based algorithms

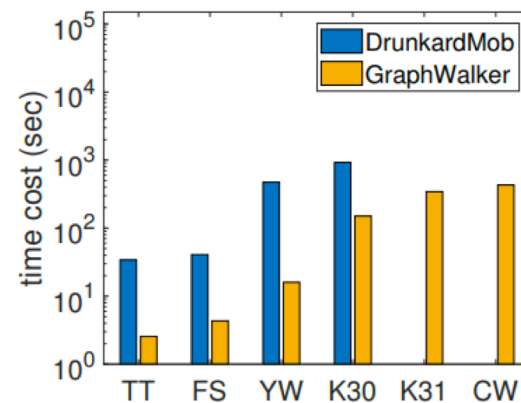
We now evaluate the performance of the four common random walk based algorithms. From Figure 12, we can see that GraphWalker achieves 9 x to 48 x speedup upon DrunkardMob.



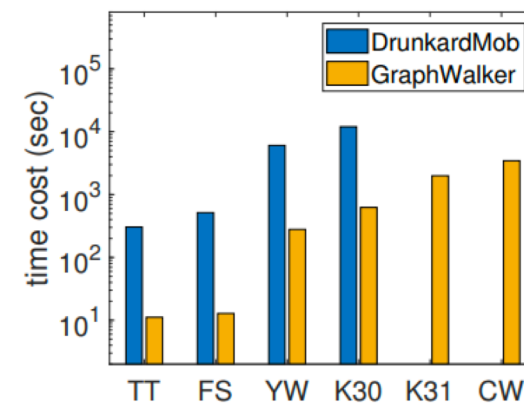
(a) Personalized PageRank



(b) SimRank



(c) Graphlet



(d) Random Walk Domination

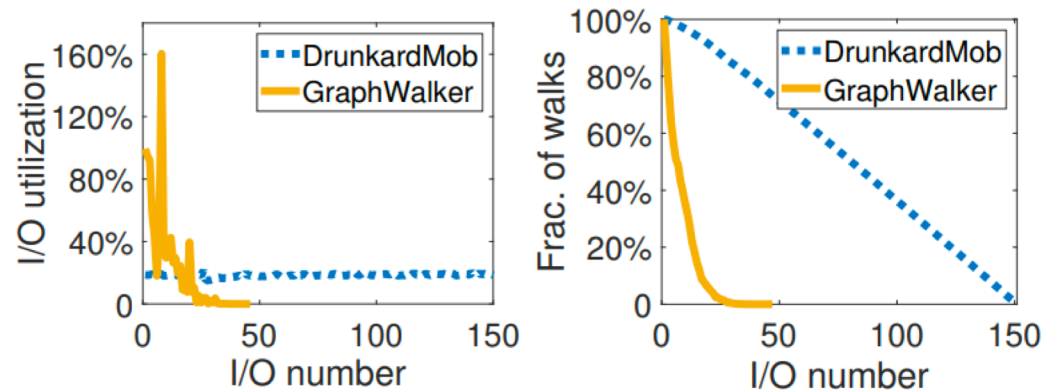
Figure 12: Performance of random walk based algorithms.

Micro-benchmark drilling down

I/O utilization: defined as the edge usage amount for updating walks divided by the total number of edges loaded by one I/O.

GraphWalker develops an asynchronous walk updating method to fully utilize the loaded graph data in memory

- (1) graph loading, loads graph blocks into memory with disk I/Os
- (2) walk updating, updates the walk states maintained in memory
- (3) walk persisting, includes to read walk states from disk into memory and write back updated states to disk for persistency. of executing each operation.



(a) I/O utilization

(b) Walk updating rate

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

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

Comparison with state-of-the-art systems

Single-machine graph systems.

First, GraphWalker consistently outperforms Graphene. It achieves up to 19 x speedup.

Second, compared with GraFSoft, when the number of walks is small, the improvement of GraphWalker is limited. However, the improvement of GraphWalker increases as the number of random walks gets larger

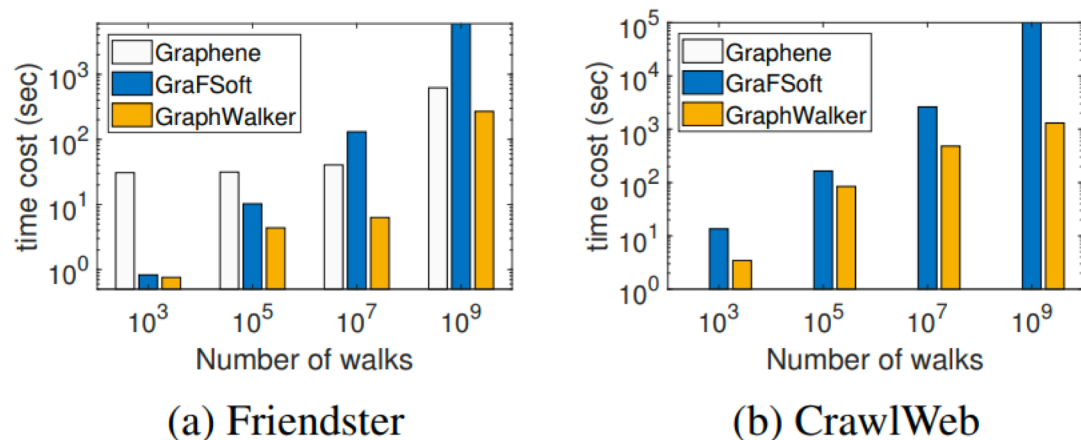


Figure 14: Comparison with Graphene and GraFSoft

Distributed random walk system.

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

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.

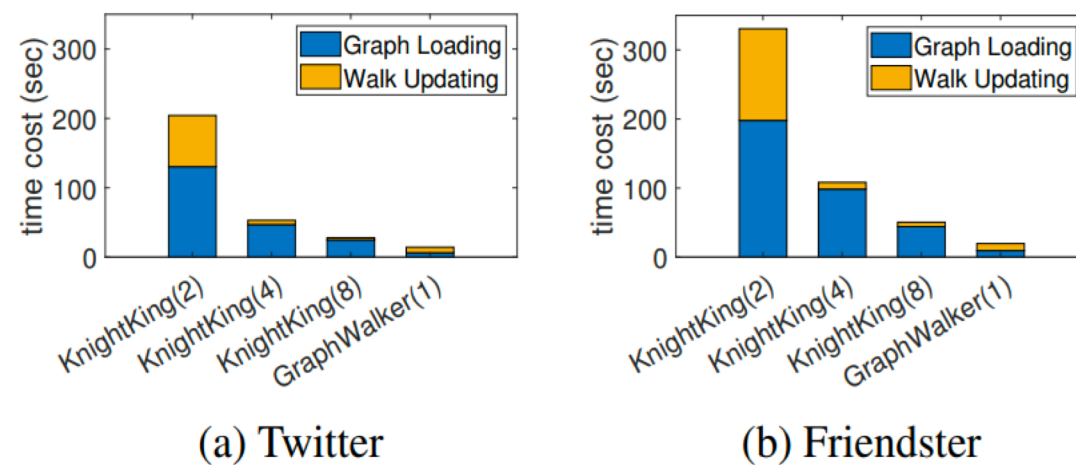


Figure 15: Comparison with KnightKing

Impact of system configuration

Performance on HDDs.

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 x to 135 x speedup under different settings

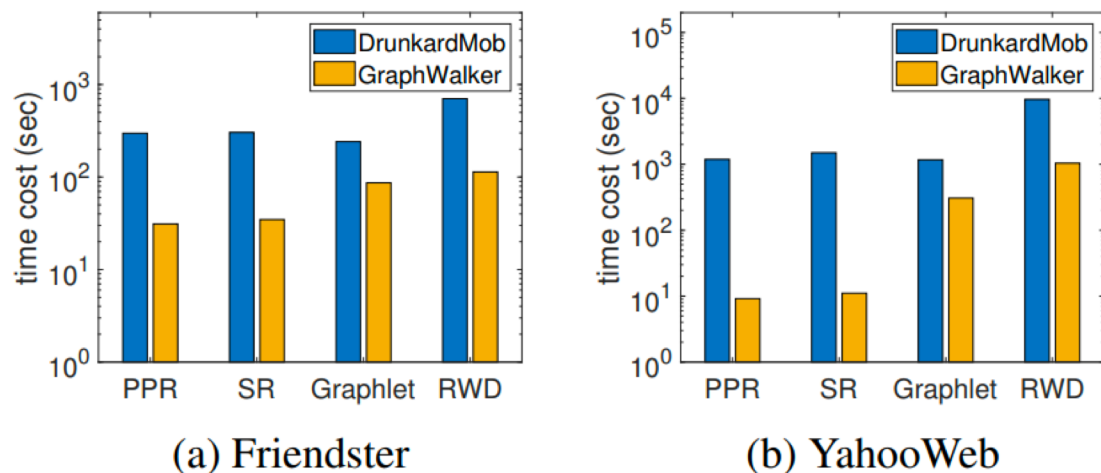


Figure 16: Performance on HDDs

Impact of block size.

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.

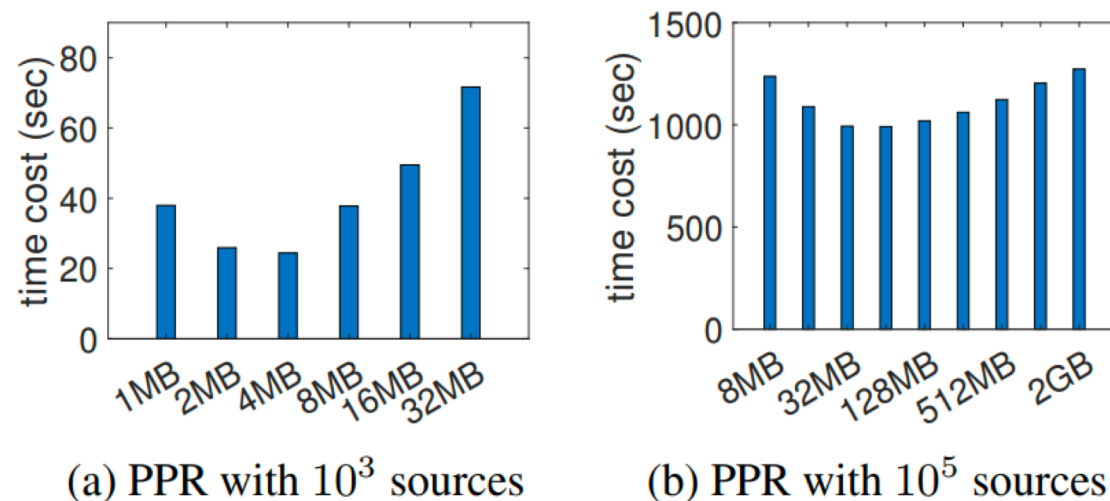


Figure 17: Impact of block size

Summary

GraphWalker is an I/O-efficient and resource friendly graph system.

- A novel **state-aware I/O model**, which leverages the state 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 utilization.
- An **asynchronous walk updating scheme** based on the re-entry method, which allows each walk to move as many steps as possible so as to fully utilize the loaded subgraph and greatly accelerate the progress of random walks.
- 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.