Random Walks on Huge Graphs at Cache Efficiency

Ke Yang*†‡
yangke14@mails.tsinghua.edu.cn

Xiaosong Ma†
xma@hbku.edu.qa

Saravanan Thirumuruganathan†
sthirumuruganathan@hbku.edu.qa

Kang Chen*‡
chenkang@tsinghua.edu.cn

Yongwei Wu*‡
wuyw@tsinghua.edu.cn

Abstract

Data-intensive applications dominated by random accesses to large working sets fail to utilize the computing power of modern processors. Graph random walk, an indispensable workhorse for many important graph processing and learning applications, is one prominent case of such applications. Existing graph random walk systems are currently unable to match the GPU-side node embedding training speed.

This work reveals that existing approaches fail to effectively utilize the modern CPU memory hierarchy, due to the widely held assumption that the inherent randomness in random walks and the skewed nature of graphs render most memory accesses random. We demonstrate that there is actually plenty of spatial and temporal locality to harvest, by careful partitioning, rearranging, and batching of operations. The resulting system, FlashMob, improves both cache and memory bandwidth utilization by making memory accesses more sequential and regular. We also found that a classical combinatorial optimization problem (and its exact pseudo-polynomial solution) can be applied to complex decision making, for accurate yet efficient data/task partitioning. Our comprehensive experiments over diverse graphs show that our system achieves an order of magnitude performance improvement over the fastest existing system. It processes a 58GB real graph at higher per-step speed than the existing system on a 600KB toy graph fitting in the L2 cache.

*Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China.
†Qatar Computing Research Institute, Hamad Bin Khalifa University.
‡Beijing HaiZhi XingTu Technology Co., Ltd.

A large part of this work was carried out during the first author's research internship at QCRI.

SOSP '21, October 26–28, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10…$15.00
https://doi.org/10.1145/3477132.3483575

CCS Concepts: • **Mathematics of computing** → **Probabilistic algorithms**; • **Computer systems organization** → **Multicore architectures**; • **General and reference** → **Performance**.

Keywords: graph computing, random walk, memory, cache

ACM Reference Format:
Ke Yang, Xiaosong Ma, Saravanan Thirumuruganathan, Kang Chen, and Yongwei Wu. 2021. Random Walks on Huge Graphs at Cache Efficiency. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–28, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3477132.3483575

1 Introduction

Modern computers have sophisticated memory hierarchies designed for data-intensive applications. Multiple levels of CPU cache, along with high DRAM bandwidth as well as features like hardware prefetching and memory row buffers, transparently help programs profit from temporal and spatial locality. The former allows data reuse. The latter facilitates large, sequential accesses, which tend to be much faster. Unfortunately, applications that heavily perform random accesses on a large working set obtain much lower benefits from state-of-the-art CPU hardware. They often spend the bulk of their execution time stalling for data, wasting expensive data center or server resources.

One prominent example of such applications is graph random walk [57], a graph workload that is becoming increasingly important. It has been heavily used in industry by companies such as Alibaba [3, 88], Facebook [4, 31], Google [2, 65, 66], LinkedIn [55], Tencent [82, 91], and Twitter [75]. Given an input graph, a random walk application issues a certain number of *walkers*, each walking among the vertices, by sampling one edge out of its current vertex according to a certain *transition probability* specification. An important and emerging application of random walks is graph embedding [8, 18, 33, 37], used for diverse applications such as node classification [35, 37], link prediction [35], recommendation [17, 27, 88], attribute prediction [95], and community detection [12, 90]. More generally, random walks are used for graph analytics (such as PageRank computation [43, 65]), sub-graph sampling [25, 47], aggregate estimation [5, 20, 62], ranking [53, 100], data integration [11, 83],

cardinality estimation [76] for query optimization, predicting drug interactions and functionality [14, 58], fake news mitigation [44], epidemics study [16], etc.

Many graph random walk executions today feed edge or path samples to graph embedding training, typically using stochastic gradient descent (SGD) methods. Therefore recent graph embedding frameworks (such as GraphVite [102] and Tencent's graph embedding system [91]) simultaneously perform graph random walk on CPUs and embedding training (plus downstream applications) on GPUs. The much faster compute power growth of GPUs relative to CPUs brings continuous pressure for the host-side random walk to keep up with the GPU-side embedding computation. This pressure is further intensified by the common adoption of multi-GPU nodes for machine learning applications, where a few GPUs could concurrently train independent graph embeddings using different hyper-parameters [2, 26]. In [91] the generated random walks are used 10 times to hide the overhead of the random walk engine in the pipeline.

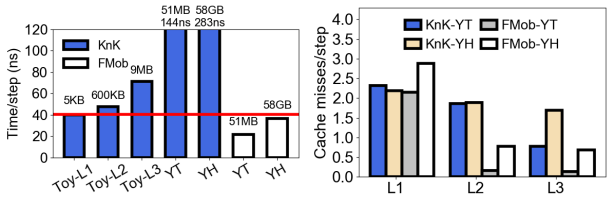
Graph random walk has been assumed to have little locality. The inherent randomness in its probabilistic operations over large, irregular graph datasets renders most memory accesses random. Existing systems process each walker sequentially and independently, sampling one edge on the fly and moving wherever it leads to in DRAM. With such low expectations, no coordination or scheduling is done among walkers/vertices. Most systems adopt routine random walk settings (such as having $|V|$ walkers each walking 80 steps within a *round* and repeating the process for 10 rounds), regardless of graph size or topology. In most cases, only one edge (the one sampled) stored in a cache line is used.

We argue that underneath the apparent random nature of walking large graphs, there is plenty of spatial and temporal locality to harvest, by *careful partitioning, re-arranging, and batching of operations*. This work proposes FlashMob, a new graph random walk design. It enables largely sequential memory accesses, to graph partitions strategically cut to be processed within different cache levels.

The result is an iterative graph random walk pipeline reminiscent of the MapReduce workflow [21]: walkers residing on a partition of vertices are processed together, with single-step transitions dumped as messages; the messages are processed in a shuffling stage to regroup the walkers by their new locations.

FlashMob's streaming processing of partitions containing similar-degree vertices subsequently enables multiple optimizations. For the small number of high-degree vertices, we harvest their *access density*, by batching walkers who happen to co-locate on these hot spots. For low-degree ones, we exploit their *regularity* in degree, adopting simplified data structures with direct indexing to reduce random memory accesses in their processing.

We found challenging questions like "How to partition the vertices" and "which cache level to fit a task" can be mapped



(a) Time **(b)** Cache miss count

Figure 1. Performance highlight: FlashMob achieves similar per-step time on a 58GB graph (YH) as KnightKing on a 600KB toy graph that fits into L2.

to a known multi-constraint optimization problem, which FlashMob solves efficiently by applying an existing dynamic programming algorithm.

To our best knowledge, FlashMob is the first graph random walk system that explicitly fits most of its computation into the CPU cache, with fast streaming from/to the DRAM. It brings an order of magnitude performance improvement compared with the best baseline we could find. To give a performance highlight, Figure 1a reports the per-walker-step average execution time (our main performance metrics, called *per-step time* for the rest of the paper) using the popular DeepWalk algorithm [66].

The first 5 bars in blue are by KnightKing [94], a state-of-the-art graph random walk engine, on three toy graphs sized to fit the data footprint entirely into the L1, L2, and L3 capacities respectively, plus two real-world graphs: YouTube (YT) and YahooWeb (YH). Clearly the per-step walk time steadily increases as the graph size grows, as more accesses go to the next level of memory hierarchy. The final two bars, in white, give FlashMob's speed for much larger graphs including the 51MB YT graph and 58GB YH graph. FlashMob's speed on the YH graph, the largest among the 5 real-world graphs in our evaluation, matches KnightKing's performance on a 600KB toy graph. Figure 1b gives per-step cache miss count breakdowns for KnightKing and FlashMob on YT and YH, which confirms the latter's significant reduction in cache misses, especially at the L2 and L3 levels.

Finally, though the paper focuses on graph random walks, insights and techniques here apply to a wider range of applications that rely on random sampling. For example, an important component of approximate graph mining systems (such as ASAP [41] and GraphSage [36]) performs neighborhood sampling that expands sampled subgraphs, which would also benefit from FlashMob's cache-friendly design.

2 Background

2.1 Graph Random Walk Basics

Given a graph $G = (V, E)$ and a starting vertex $u \in V$, a random walker w proceeds as follows. Each neighbor v of u is associated with a *transition probability* that determines the likelihood that v would be chosen as the next vertex. Typically, the transition probability $p(v|u)$ only depends on

u, in which case we have a *first-order* random walk. In a *higher-order* random walk, the probability is specified in the form of $p(v|u, t, s, \ldots)$, where s, t are the predecessors of u in the current walk, involving w's walk history in computing the out-going edges' transition probability. w samples an edge according to this probability and repeats until certain termination criteria are satisfied. The termination could be deterministic (after a given number of steps) or stochastic (walkers exiting with a fixed probability at each step).

While random walks are widely used, here we give more details on one application: *node embeddings*, today an essential component of graph learning. Given a graph $G = (V, E)$ and a dimensionality $d \ll |V|$, the node embedding problem [8, 18, 33, 37] seeks to represent each $v \in V$ as a d-dimensional vector $Emb(v)$ such that G's structural information is preserved. Intuitively, if a node pair u and v have "similar" neighborhoods, then their embeddings $Emb(u)$ and $Emb(v)$ are also close to each other. In contrast, two nodes with dissimilar neighborhoods will have embeddings farther apart. This is achieved by training a deep learning model over a collection of positive node pairs $\mathcal{P}$ and negative node pairs $\mathcal{N}$ to learn node embeddings such that node embeddings for $(u, v) \in \mathcal{P}$ are closer to each other while those for $(u, v) \in \mathcal{N}$ are farther from each other.

Typically, $\mathcal{N}$ is constructed by randomly picking node pairs (given the sparsity of real-world graphs, they are unlikely to be connected), while $\mathcal{P}$ is by random walks. Widely-used node embedding algorithms such as DeepWalk [66] and node2vec [35] differ in how they use random walks to measure neighborhood similarity, by giving different transition probability definitions. More specifically, DeepWalk executes a first-order uniform random walk. Node2vec, on the other hand, is a second-order algorithm with hyper-parameters to create a configurable interpolation between BFS and DFS. In typical runs, both algorithms adopt default parameters requiring 10 random walks, of length 40 and 80 respectively, starting from each node in the graph [35, 66].

2.2 Random Walk System Design

The current state-of-the-art approaches focus on algorithmic improvements, such as improved edge sampling (KnightKing [94]) or out-of-core walks on large graphs (GraphWalker [89]) that significantly reduces the memory requirement for complex random walk algorithms and enables in-memory processing of very large graphs. However, once the main walk task (sampling the next vertex) fits into memory, all existing systems happily pay the random access costs.

Without exception, existing systems process walkers one by one: during an iteration, all (active) walkers take turns to each sample and follow one edge from the adjacency list of its current vertex. Such common practice, while intuitive, incurs huge resource waste. Unlike graph processing tasks, graph random walk performs much more sparse processing, by choosing one edge among potentially many candidates. The majority of content from a sampled cache line (typically storing a dozen or more edges) is therefore discarded. This abuses the memory hierarchy in multiple ways: low utilization of cached data in the fast and private L2 cache, low data re-use rate in the shared L3 cache and increased total DRAM traffic. Our work aims at mitigating such waste.

Aside from random accesses, current systems casually perform pointer-chasing. GraphVite [102] finishes one walker's entire path before starting another, while KnightKing [94] (a distributed walk engine) moves a walker as much as possible before it leaves the local graph partition, as an optimization. The data dependency brought by such operations further lowers their memory access efficiency.

2.3 Cache Hierarchy in Recent Processors

Recently, there has been a number of architectural innovations targeting data center workloads that heavily use virtualization. While here we discuss Intel's current generation processors, processors from other major vendors also have similar characteristics.

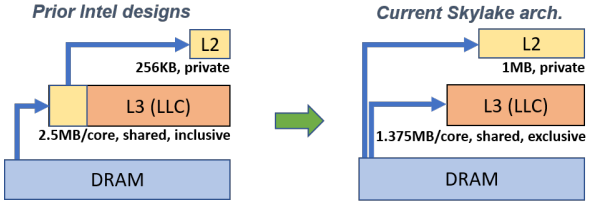


Figure 2. Changes in Intel processor cache design

As illustrated in Figure 2, recent Intel processors retain the layered cache hierarchy of modern multi-core processors: each core with its private L1 and L2 caches, while all cores within a socket share an LLC (Last Level Cache, typically the L3) that sits between the cores and the main memory. Prior Intel CPUs (*Broadwell* family and earlier) adopt an LLC an order of magnitude larger than the L2, with *inclusive* LLC management. This means all the data brought into the much smaller L2 cache also reside in the L3. With Intel's current Scalable Family (*Skylake*) processors, the relative size ratio between L3 and L2 is dramatically reduced: *e.g.*, a typical per-core cache configuration is 1MB L2 and 1.375MB L3 (total L3 size divided by the number of cores). The significantly larger L2 is further promoted by a new, exclusive L3 design: cache misses will bring data directly into L2 and not L3, with the latter used to hold data evicted from L2, allowing better overall cache capacity utilization and facilitating data sharing among cores. Also, with the non-inclusive L3 design, though the combined cache size actually shrinks from the previous Broadwell architecture (from 2.5MB to 1.375MB per core), more of such space is now on the faster L2 and holds a disjoint set of data as the shared L3.

While these design changes are optimized for virtualized and multithreaded workloads [40, 79], where a larger private L2 accommodates more core-private data and reduces inter-workload interference, they also bring fresh opportunities

for single-workload execution. By simultaneously executing random and streaming accesses (as well as cache-aware data organization and work partitioning), we allow the former to consciously retain their working set within the L2 (or even L1), and the latter to enjoy most of the shared L3 capacity and the memory bandwidth.

Location	L1C	L2C	L3C	LocalMem	RemoteMem
Sequential read	0.42ns	0.41ns	0.44ns	**0.76ns**	**1.51ns**
Random read	**0.77ns**	**0.95ns**	**2.60ns**	**18.35ns**	24.35ns
Pointer-chasing	**1.69ns**	**5.26ns**	19.26ns	116.90ns	194.26ns

Table 1. Load latency from memory hierarchy levels

On a server with the aforementioned architecture (Intel Xeon Gold 6126), we measured the latency of loading a single word from different locations along the memory hierarchy, with varied access patterns. Results in Table 1 confirm that (1) despite the random-access nature of this memory hardware, there is a big latency gap between sequential and random accesses; (2) sequential streaming brings affordable latencies even from remote memory across NUMA nodes, while the sequential-random performance gap grows fast as we go down the hierarchy; and (3) pointer chasing is expensive, whose cost renders accesses within the L3 cache slower than simple random accesses to the DRAM.

Existing systems assume fitting the graph into DRAM is the best one can hope for, running at latency levels largely from the most expensive cases in Table 1 (marked in red). In the next sections, we explain how FlashMob manages to avoid these high-cost accesses and stay mostly within the green territory, especially the cases highlighted in bold.

3 Approach Overview

The new architectural features and the inefficiencies found with existing systems motivate us to design FlashMob, a new solution to graph random walk. We first present a brief yet concrete workload characteristics report, followed by an overview of FlashMob's major innovations.

Graphs		<1%	1%~5%	5%~25%	25%~100%
YT	$\overline{D}$	338.4	38.0	8.5	1.2
	|E|	39.0%	21.9%	24.3%	14.9%
	|W|	39.0%	21.9%	24.3%	14.9%
TW	$\overline{D}$	3463.0	291.2	50.5	7.9
	|E|	49.1%	20.7%	17.9%	12.3%
	|W|	49.1%	20.6%	17.9%	12.3%
FS	$\overline{D}$	1027.6	296.4	90.8	6.6
	|E|	18.7%	26.9%	41.2%	13.2%
	|W|	18.7%	26.9%	41.2%	13.2%
UK	$\overline{D}$	3874.8	264.8	69.4	12.9
	|E|	46.4%	15.8%	20.8%	17.0%
	|W|	56.8%	12.9%	17.7%	12.6%
YH	$\overline{D}$	856.7	78.0	22.0	3.1
	|E|	46.5%	16.9%	23.8%	12.8%
	|W|	53.0%	14.7%	21.3%	10.9%

Table 2. DeepWalk statistics by degree groups

Random Walk Workload Characteristics. Table 2 summarizes sample profiling results from running the DeepWalk [66] algorithm on five real-world graphs with varying sizes (details in Table 4). The tests have $|V|$ walkers, each walking 80 steps, initially placed by uniformly sampling among all edges. For each graph, we group its vertices into 4 buckets based on their degree percentile.

For each group, Table 2 lists their average degree ($\overline{D}$) and share in two dimensions: total number of edges ($|E|$) and number of walkers stopping by ($|W|$). The results confirm the dramatic disparity in graph vertices' popularity in random walk (and graph sampling in general): the higher-degree vertices attract most of the traffic. In particular, the first group (the top 1% vertices with the highest degrees) occupies around half of all visits with three graphs (TW, UK, and YH).

For all graphs, vertices from the first two columns (top 5% in degree) possess 45.6% to 69.7% of visits. The vast majority of the vertices (bottom 75%, last column), meanwhile, have an average degree of 1.2 to 12.9 and attract under 15% of walk traffic across all five graphs. Finally, while the absolute average degree varies significantly among these graphs, for each degree-based vertex group, its share of total visit counts highly correlates with that of edge counts.

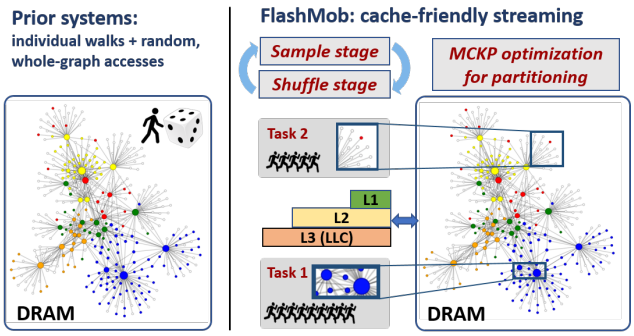


Figure 3. FlashMob vs. existing system design

Unlike existing systems, which handle all vertices uniformly, FlashMob's design is based on this highly skewed traffic pattern, as described below.

FlashMob Workflow and Innovations. Figure 3 depicts the FlashMob architecture overview. We highlight its major innovations while walking through its workflow.

Streaming processing of partitioned graph: Unlike existing random walk implementations, FlashMob does not follow walkers to visit the entire graph, even when there is ample memory space to host the latter. Instead, it sorts all vertices in descending order of degree and cuts them into many vertex partitions. Figure 3 shows two such partitions, one with a few high-degree vertices and the other with many low-degree ones. A thread will only work on one task at a time, to move *all walkers* currently on one partition by one step. This obviously will require the walk to be decoupled into stages: *sample* and *shuffle* in this case, introducing to graph random walk a streaming model reminiscent of MapReduce.

Cache-friendly walker batching: All existing solutions move walkers individually. FlashMob, recognizing the heavy traffic on popular vertices, aggressively batches co-located walkers,

as enabled by its shuffle process. Instead of randomly fetching an edge for each walker independently, it can *pre-sample* many edges, achieving both much cheaper sample generation and full utilization of cache lines bringing in such pre-sampled edges. Combining vertex partitioning with walker batching, FlashMob dramatically enhances memory access efficiency by (1) fitting the working set of each task in cache and (2) promoting sequential memory accesses. Essentially, it performs "out-of-core" processing *within CPU caches* and uses DRAM for fast data streaming.

Automatic policy/partition planning: The overall walk performance depends on the intricate interplay of many factors, such as the graph size and degree distribution, the size/speed of each cache level, and the relative walker "density" on the graph. Rather than hard-coding or leaving it to users to configure important parameters such as partition and sample policy setting (when to enable pre-sampling), through approximation FlashMob maps its optimization to the Multi-Choice KnapSack (MCKP) problem [78]. As shown in Figure 3, it performs a quick, one-time automatic configuration at the beginning of a run to cut its vertex array into partitions and assign each partition an appropriate sample policy.

4 System Design

4.1 Frequency-Aware Vertex Grouping

A key design change brought by FlashMob is to *partition the graph* and *batch the walkers* residing within each partition.

Vertex ordering. Table 2 clearly shows that random walks disproportionately visit high degree vertices, echoing earlier work on graph random walk [57, 94], as well as other graph sampling algorithms [49].

Considering the overall high correlation between vertex degree and their popularity in random walks, FlashMob arranges the vertices of the input graph in descending order of their degree. The overhead of this pre-processing is quite small, as to be shown in Sec 5.2.

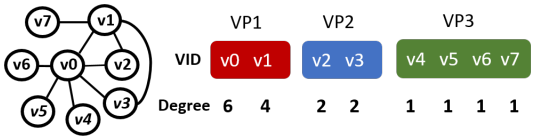


Figure 4. Sample vertex partitions

Variable-size vertex partitioning. The sorted vertex array is then cut into contiguous *vertex partitions* (*VPs* for short) of varying sizes. Figure 4 demonstrates a sample graph, whose 8 vertices are sorted by degree and cut into 3 VPs. The VPs form the basic task units, whose sizes are optimized based on both graph characteristics and system resource constraints (details in Section 4.4). With vertices of similar degrees grouped, FlashMob handles them with different strategies and gains performance from different sources, as to be seen in Section 4.2.

Walker Data Organization. FlashMob also partitions walker data so that during each walk iteration, a VP is processed with a contiguous chunk of walker data to access and update walker locations. To this end, FlashMob seeks to have simple and compact walker state storage, which doubles as "messages" to be shuffled between walk steps, and walkers' path history (details in Section 4.3).

For the ith iteration, FlashMob adopts a simple 1-D array $\mathtt{W}_i$, with length of $|W|$, where $\mathtt{W}_i\mathtt{[j]}$ stores the current location of walker w_j, in *vertex ID (VID)*. Additional walker metadata, if any, could be stored and shuffled alongside the walkers. During the shuffle stages, the walkers are rearranged so that those residing within the same vertex partition are stored contiguously. At the beginning of the execution, $\mathtt{W}_0$ is initialized by assigning each walker a vertex as its start point. Figure 5 gives a sample initial walker array.

4.2 Edge Sample Stage

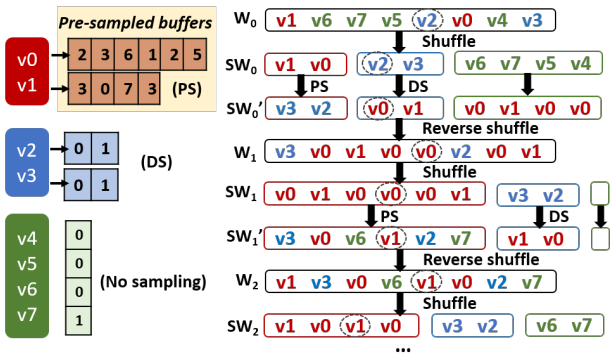


Figure 5. FlashMob walker movement, with the locations of the 5th walker circled

The edge sampling phase of FlashMob's random walk iteration performs the central task: finding one edge to move along for each walker. More specifically, for a given VP, FlashMob selects one sampling policy, either *pre-sampling (PS)* or *direct sampling (DS)* (defined below), and consequently adopts different data organization and access patterns.

Common to both policies, for the i-th walk iteration, a FlashMob thread is assigned one VP at a time, along with a contiguous chunk of the sorted walker array $\mathtt{SW}_i$, storing all the walkers currently within this partition. It scans through this chunk of SW, updating all walkers in order.

Once a walker w finds its outgoing edge from its current location v, either with PS or DS, FlashMob records it by overwriting its VID (*i.e.*, v) in $\mathtt{SW}_i$ with its next stop. At the end of one sample stage (and after a "reverse-shuffle" process to be described later), the $\mathtt{SW}_i$ becomes $\mathtt{W}_{i+1}$, ready as the input to the next shuffle stage. In Figure 5, $\mathtt{W}_1$ and $\mathtt{W}_2$ store the locations of all walkers, in the original *walker ID (WID)* order, after they make the 1st and 2nd steps, respectively.

In addition to having these bandwidth-aware in-place updates, walker state accesses perform a single sequential scan, leaving most of the cache space to edge data.

Pre-sampling (PS) This policy is designed for more frequently visited vertices, maximizing cache utilization by batching walkers on the same vertex. The main idea is to sample in advance many edges, which are consumed sequentially by the many co-located walkers. For a VP adopting the PS policy, each vertex allocates a *pre-sampled edge buffer*, where walkers retrieve edge samples. As mentioned earlier, a vertex's frequency of being visited is strongly correlated to its degree, so we set the size of the pre-sampled buffer of vertex v to $d(v)$. The thread processing the sample task of a given VP is in charge of refilling the pre-sampled edge buffers when they are empty. In Figure 5, only the first VP adopts PS, where $v0$ and $v1$ have edge buffers of size 6 and 4.

With PS, one essentially decouples the edge sample *production* and *consumption* phases. While this generates one extra round of edge sample storing/retrieving operations, by batching similar operations, the overall memory accesses themselves are significantly more efficient.

In the production step, FlashMob throws the dice consecutively, to refill its pre-sampled edge buffer with the edge transition probability. Note that this is done for one vertex at a time, during which its edges (each on average sampled once) could reside in the cache to allow fast random reads, with one sequential write stream to the buffer being refilled.

In the consumption step, the filled buffer will feed $d(v)$ walkers departing from v. For example, in Figure 5, walkers on $v0$ would sequentially follow edge samples to $v2$, $v3$, $v6$, etc. Though it appears that this buffer occupies exactly the same space as v's adjacency list, with contiguously stored, ready-to-use edge samples, at any given time there is only one *active* cache line accessed per vertex.

Direct sampling (DS) Intuitively, the benefit of pre-sampling diminishes with lower-degree vertices. In the extreme cases, for vertices with only one edge (who constitute a proportion between 3.5% and 49.3% of vertices among our 5 real-world graphs), like VP3 in Figure 5, there is no need for edge sampling at all. For those with a degree of 2 (who make up another 5.0%-14.7% of vertices), like VP2 in Figure 5, storing and sampling from their two edges saves both space and time, as to be shown with profiling results later.

Therefore, FlashMob provides the option of DS, where a walker throws a dice on the spot to select an outgoing edge from the (often short) adjacency list. In Figure 5, whenever a walker is within VP2, it randomly samples from the two outgoing edges of a vertex. *E.g.*, the 5th walker from its initial location ($v2$) selects $v0$ as the next stop.

Besides compact edge storage, DS allows FlashMob to exploit other sources of optimization. Real-world graphs' long tails produce many partitions of uniform-degree vertices (like VP2 and VP3 here). While high-degree partitions have to adopt the traditional CSR [70] graph representation, where sampling v's adjacency list requires one random access to read v's degree first, low-degree partitions allow simpler indexing. Here FlashMob could easily keep track of the one or a few distinct degree values for a partition, enabling it to quickly access and sample using simple arithmetic.

Memory access patterns and partition sizing Besides PS/DS selection, another dimension of FlashMob's decision making is the size of the VPs. Meanwhile, the sensitivity of the sample stage performance to the VP size heavily depends on a partition's degree distribution and sampling policy. To help understand such relationship, Table 3 summarizes the major random walk memory access patterns, comparing our proposed PS and DS policies with the naive direct sampling approach used in prior random walk implementations.

Approach/Operation		Access pattern
Common	Read walker state	Single-stream seq. read
	Update walker state	Single-stream seq. write
PS	Edge buffer refill	Single-adj-list random read + single-stream seq. write
	Sample retrieval	Multi-vertex random read + multi-stream seq. read
DS	Cache-aware partitioned direct sampling	Multi-adjacency-list random read
Prior systems	Whole graph/subgraph direct sampling	Random read, potentially w. pointer chasing

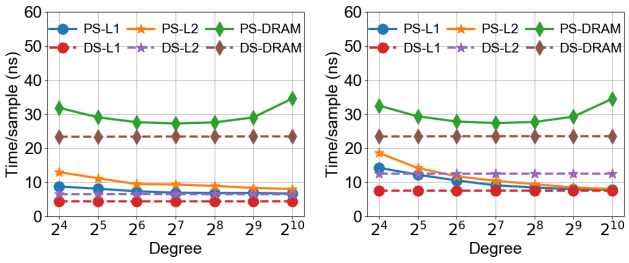
Table 3. Major memory access patterns in edge sampling

Common to all approaches are the accesses involved in reading and updating the walker state: the current- and next-step locations of the walkers are sequentially retrieved / stored in large arrays. As in existing systems, each FlashMob thread processes a subset of walkers. The major difference is its cache-friendly edge sampling that also partitions the graph to fit the task in cache.

The random walk implementation in prior systems needs to randomly access one vertex and randomly selects one of its edges, either within the entire graph [34, 35, 66, 102] or within an entire sub-graph staged to the local memory in the case of out-of-core/distributed processing [51, 82, 89, 94], often with extra speed bumps brought by pointer chasing. With DS, FlashMob does not change the direct sampling semantics or operation complexity but limits the scope to the adjacency lists within the current VP.

With PS, FlashMob actually adds operation complexity: pre-sampling performs random read within a single adjacency list and refills a single pre-sampled edge buffer with sequential write. It then reads back such buffers in the actual walk, with multi-stream sequential read (one per vertex) plus a single random read to "seek" to the proper buffer position for each walker on its current vertex. More memory accesses are performed, but with further reduced scope and promotion of sequential scanning, each access is much faster. As to be seen later, reducing waste from discarding unused cache line content also saves overall DRAM traffic.

The effectiveness of FlashMob's cache-friendly edge sampling is mainly determined by two factors. First, whether



(a) 1 walker/edge **(b)** 0.25 walker/edge

Figure 6. The per-step sample time for different sampling policies and typical VP task sizes (categorized by active memory used, not graph size)

(and at which level) the random accessed working set fits into cache, which depends mostly on the vertex partition size. Note that although PS and DS both involve random reads, PS has a smaller working set (one vertex's data for pre-sampling, then space to hold per-vertex pointers, plus one active cache line per vertex for edge samples). DS, in contrast, needs to fit in cache all *edges* within a VP. Therefore, to fit into the same cache level (*e.g.*, L2), PS allows much larger partitions than DS does with high-degree vertices.

The other factor concerns data reuse rate, which leads us to define *walker density* (*density* for short), as the average number of walkers per edge on a graph. For a VP, this is simply the total number of walkers divided by the total number of edges possessed by its member vertices. The higher the density, the better chances for random accesses to hit cached data, which is more pertinent to DS than to PS.

Figure 6 gives sample sensitivity results, on synthetic VPs possessing a uniform degree, ranging from 1024 to 16. The figures give the sample-stage performance at the density of 1 (Figure 6a) and 0.25 (Figure 6b) walker/edge, under PS (solid lines) and DS (dashed), each with VPs sized to fit the working sets into L1, L2, L3, and DRAM (set at 8×L3 capacity), respectively. A few observations can be made from these figures:

1. Both policies benefit from fitting the working set into faster caches.
2. The higher the degree is, the faster PS works, as higher degree vertices attract more walkers, bringing higher utilization of sequentially read cache lines. All the DS lines, meanwhile, are insensitive to vertex degree.
3. When data fits into caches, both policies benefit from higher walker density, which enhances cache utilization. Otherwise they both stay insensitive to walker density.
4. Regarding specific strategies, DS-L1 performs the best as it involves fewer accesses, which however would require cutting a large graph into a huge number of VPs. PS-L1 follows closely, especially with higher degree vertices, where cache lines for both sample production and consumption get better utilized. PS-DRAM is significantly slower than any other combinations, as when there are more streams to fit one active cache line from each into the cache, it pays the extra operations without benefiting from sequential streaming of pre-sampled edges.

These figures illustrate FlashMob's profiling data, reusable across different graphs, allow comprehensive partitioning optimization by considering the complex tradeoffs here in the sample stage and in shuffling (to be discussed next).

4.3 Shuffle Stage

After each sample stage, walkers get dispersed from each VP. In the subsequent shuffle stage, all threads work in parallel to rearrange walkers, so that those now within the same VP are again stored contiguously. With higher-order walks, additional per-walker data needed by edge sampling (*e.g.*, immediate walk history for node2vec) are shuffled together with walkers' current location.

Scalable walker-to-partition shuffling. As depicted in Figure 5, the overall result is another 1-D array of the same size, $\mathtt{SW}_i$, storing a permutation of $\mathtt{W}_i$, with array elements ordered by their VP affiliation. Note that walkers within each VP are not further grouped by vertex, hence the multi-stream sequential or in-partition random access patterns as discussed previously.

Since FlashMob adds this extra stage to facilitate cache-local, mutually-independent edge sampling, it is critical that the shuffling is done efficiently. Fast shuffling itself, in turn, relies on cache-friendly design. Again we find the L2 cache size provides a nice balance between capacity and speed. Here, chunks of $\mathtt{W}_i$ are scanned twice, first to count the number of walkers walking into the destination partitions (bins), then after aggregating counting results, to write new walker locations into appropriate locations in $\mathtt{SW}_i$.

Therefore, the L2 cache size determines the number of VPs to be accommodated by a single level of shuffle, *i.e.*, the number of concurrent sequential write streams to $\mathtt{SW}_i$. If the number of VPs goes over this limit, FlashMob is forced to add a level of shuffle, which adds to the overall overhead. This forms a major constraint on VP sizing, as to be formally discussed in Section 4.4.

Compact walker state storage. Like in the sample stage, threads work on disjoint array areas, simplifying synchronization and eliminating the needs for locks. Meanwhile, FlashMob aligns per-partition walker data to cache lines to avoid false sharing. More importantly, care is taken to minimize the message footprint. Rather than having explicit key-value pairs like $<w, v>$, the $\mathtt{W}$ and $\mathtt{SW}$ arrays only store VIDs (see Figure 5). Walker identities, meanwhile, are *implicitly* carried by their consistent ordering in the $\mathtt{W}$ arrays.

For each vertex partition, the associated elements in $\mathtt{SW}_i$ remain in the same order as encountered by a linear scan of $\mathtt{W}_i$. Therefore after all the walkers make one step in the sample stage and overwrite $\mathtt{SW}_i$ to be $\mathtt{SW}_i'$ (no data copy), by scanning $\mathtt{W}_i$ again we could find the shuffled location of each

walker in W_{i+1} and retrieve its content there. This produces W_{i+1}, input to the $(i+1)$-th shuffle stage, which contains the walkers' updated placement after the ith iteration while preserving the original walker ordering as in W_0. In Figure 5, *e.g.*, the 6 walkers on the first VP will have their new locations (stored in SW'_1) sequentially written to the W_2 positions where a scan of W_1 would find vertices in the same VP.

By doing this, FlashMob trades off cheap computation (with streaming memory access) to trim the walker array footprint by half. It saves main memory bandwidth in both sample and shuffle stages, as well as a non-trivial amount of DRAM space. The latter in turn allows FlashMob to accommodate more walkers in each round of random walk, maximizing walker density for better data reuse.

Random walk paths output. At the end of an n-step random walk, the $(n+1)$ W_i arrays store the entire walk history of all walkers. By transposing these arrays we can get $|w|$ 1-D arrays, each carrying the $(n+1)$ VIDs forming a walker's output path. Alternatively, users could choose to stream the sampled edges (<W_i[j], W_{i+1}[j]>) to the GPU performing graph embedding training.

Throughout FlashMob's workflow, a design guideline repeatedly followed is to perform sequential scans of regular, compact data structures (1-D arrays in most cases). This may increase total DRAM traffic in exchange for speed, as modern processors handle sequential accesses very well, with the help of hardware prefetching. Nevertheless, as to be shown in Section 5, such increase is more than offset by DRAM traffic savings from our cache-efficient edge sampling.

4.4 Walk Optimization with Dynamic Programming

With major FlashMob operations discussed in Section 4.2-4.3, we propose a principled approach for partitioning the vertices and assigning sampling policies, by formulating it as a combinatorial optimization problem and obtaining an efficient and optimal dynamic programming solution.

Problem definition. An input graph has vertices ordered in descending degree, with vertex IDs from 0 to $(|V|-1)$, VID 0 being the vertex with the highest degree. For cache-friendly edge sampling, FlashMob needs to cut the vertices into contiguous VPs, potentially of different sizes.

The main tradeoff lies between the VP sizes and counts: smaller VPs fit into faster caches, but the benefit heavily depends on factors such as the average degree within the VP and the walk density; more VPs raise the shuffle overhead, either by not fitting into cache or increasing levels of shuffle.

Problem simplification and mapping. We make two approximations to simplify our problem and reduce the solution search space, as FlashMob is designed to accommodate a wide range of graph sizes. First, we *group* the sorted list of vertices into G groups where G is a hyper-parameter (in our experiments, G is set between 64 and 128). All groups, except the last one, are equally sized, containing a power-of-2 number of vertices for easy indexing. Since the vertices are sorted by degree, the average degree of the group will not vary drastically from the degrees of its individual members. Member vertices of a group will be again cut into equal-size VPs of power-of-2 sizes, while vertex partition sizes may vary across groups. Second, the sampling policy is to be decided *at partition level*: vertices in each resulting VP are assigned to use either PS or DS.

The problem can now be reduced to a classical combinatorial optimization problem called Multiple-Choice Knapsack Problem (MCKP) [78]. With MCKP, a set of items, each with a profit and weight, are partitioned into *classes*. The goal is to select *exactly one item* out of each class of items, to maximize the total profit while satisfying the total weight limit.

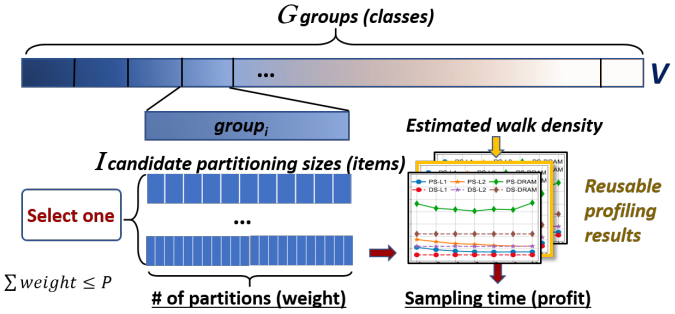


Figure 7. FlashMob's MCKP problem formulation, demonstrated by zooming into one group's optimization. Though not depicted, each candidate partitioning size will look up both PS and DS profiling results for the generated partitions.

Figure 7 shows the MCKP problem formulation. Our G groups map to G classes. For each class, the candidate "items" are the 2-D combinations of VP sizes and the per-VP sampling policy. For example, an item "(PS, DS, PS, DS)" specifies that the group has 4 uniform-size partitions, each with the given sampling policy. As VP sizes are required to be a power of 2, there are quite limited candidate sizes within a group.

For each candidate VP size (item), we define its *profit* as the negative value of its total sampling cost: the sum of the sampling time of all VPs within a group. The latter, in turn, comes from the lower between the PS and DS times for each individual VP. Its *weight*, on the other hand, is the number of VPs generated within a group using this VP size.

FlashMob's shuffle stage comes into play as the hyper-parameter P, giving the total weight limit. On a given machine, FlashMob sets P as the total number of VPs allowed to fit each outer-level shuffle task into the L2 cache (2048 on our test platform). For each group, FlashMob assesses the option of adding an internal, additional level of shuffle, whose cost is added to this candidate item's sampling time in formulating its total profit while increasing this group's weight to -1 in calculating the MCKP total weight.

Offline profiling for profit calculation. FlashMob conducts preliminary profiling to quantify a certain VP size-policy combination's sampling cost, *i.e.*, the profit value of the corresponding candidate item. This is done by running

micro-benchmarks collecting performance curves similar to those in Figure 6. We sample the relevant parameter space in terms of VP size (in vertex count), average degree, and walk density, to measure and record the per-VP sampling time under PS and DS, respectively.

One key insight here is that *the profiling itself is machine-dependent yet graph-independent*: with the VP size, degree, and density parameters set, under FlashMob's streaming model, each data point's execution is not affected at all by the graph topology, making such offline profiling a one-time effort reusable across different graphs. Its absolute cost is also quite modest: with our rather fine-grained sampling at 5% degree increment, across the space applicable to our largest graph, it costs 258 seconds, about 28% of FlashMob's DeepWalk run time on the same graph.

At runtime, based on the graph size and degree distribution, FlashMob decides the number of walkers it could hold in DRAM for each round of walk. This allows it to estimate the walker density for each VP and look up the appropriate offline profiling data points (see Figure 7), feeding FlashMob with all the MCKP parameters.

Solution via dynamic programming. Finally, FlashMob solves the fully instantiated optimization problem at the beginning of a random walk execution. MKCP is known to be NP-complete. However, it can be solved by a pseudo-polynomial dynamic programming (DP) algorithm [24, 48] with a time complexity of $O(CPI)$ and space complexity of $O(CP)$, which we implement in FlashMob. Here C is the number of MCKP classes, P is the total weight limit, and I is the maximum number of items to choose from. With $C, P, I \ll |V|$, the DP algorithm execution itself has negligible overhead. FlashMob's vertex partitioning, including generating candidate policies and solving the MCKP problem, takes 0.34 seconds on our largest graph. The DP algorithm alone requires just 0.01 seconds.

Note that with this DP-based optimization, the aforementioned new cache architecture automatically generates impact. Recall that the exclusive L2 is much closer in speed to L1 than to L3 (Table 1). Also, the private L2 is much larger than before, while per-core L3 size is smaller, making fitting data into L2 both fast and scalable to more threads, by avoiding L3 pollution brought by inclusive L2. These are implicitly reflected in the profiling curves, leading the MCKP algorithm to often favor L2-size VPs, then chop leftover lower-degree vertices into larger chunks (skipping the L3 category) to satisfy the weight constraint (results in Figure 10).

4.5 Cross-socket Walk

Common servers today possess the NUMA (Non-Uniform Memory Access) architecture with multiple sockets, where CPU cores on each socket have faster accesses to the "local" memory. With single-socket sampling policy and VP sizing optimized using dynamic programming, finally, we take one step back and examine cross-socket walks. The unique nature of updating many *independent* walkers gives us more flexibility in work decomposition. To this end, FlashMob investigates two distribution modes, as described below.

With the first mode, *FlashMob-P* ("P" for *graph partitioning*), VPs are distributed across sockets, as well as walker arrays. *E.g.*, on a 2-socket server, threads on socket 0 will always sample for the first half of VPs and shuffle the first half of walkers. Its only remote memory accesses occur in the sample stage when its VPs need to process walkers assigned to socket 1 that currently reside on socket 0 VPs. With walkers already shuffled, such remote accesses are strictly streaming-only, avoiding the super-expensive remote random accesses (Table 3). As the shuffle stage performs more scans than the sample stage, keeping its accesses local reduces overall remote memory traffic.

The second mode, *FlashMob-R* ("R" for *graph replication*), avoids remote memory accesses altogether when possible. When the entire graph, along with auxiliary data such as the pre-sampled edge buffers, fits in single-socket DRAM, one could simply replicate such graph data at each socket and run multiple independent instances of random walk simultaneously. For graphs of moderate sizes, this eliminates all remote data accesses and cross-socket synchronization. A hidden caveat, however, is that by replicating the graph itself, FlashMob-R has less memory for the walker arrays, leading to reduced walk density and lower reuse rate of cached data.

Existing solutions are happy with fitting a graph into DRAM. When unable to do so, out-of-core systems [51, 89] use DRAM as a large cache to host parts of graph data. Our evaluation results comparing the above two modes (Section 5.4) demonstrate that FlashMob sustains its in-cache processing efficiency performing cross-socket graph walk with NUMA, which shows strong promise for its future extension to walk disk-resident graphs at cache speed.

5 Evaluation

5.1 Experimental Setup

Test platform. All experiments use a Dell PowerEdge R740 server running Ubuntu 20.04.1 LTS. It has two 2.60GHz Xeon Gold 6126 processors, each with 12 cores and 296GB DRAM. Each core has a private 32KB L1 and 1MB L2 caches, while all cores within a socket share a 19.75MB L3 cache (LLC).

Graph Datasets. We test with 5 real-world graph datasets commonly used in graph systems (Table 4), including three social networks (YouTube [64], Twitter [50], and Friendster [93]), and two web graphs (UK-Union [7] and YahooWeb [92]). Their degree distribution information was given earlier in Table 2.

Systems for Comparison. We compare with two state-of-the-art systems, both released in 2019. GraphVite [102] is the first high-performance CPU-GPU hybrid node embedding system for shared-memory environments, using CPU for

Graphs	|V|	|E|	CSR Size
YouTube (YT) [64]	1.14M	4.95M	50.8MB
Twitter (TW) [50]	41.65M	1.47B	11.4GB
Friendster (FS) [93]	65.61M	1.81B	14.2GB
UK-Union (UK) [7]	131.81M	5.51B	42.5GB
YahooWeb (YH) [92]	720.24M	6.64B	57.5GB

Table 4. Graphs used (0-degree vertices removed)

edge sampling by random walks and GPU for training node embeddings, reporting a speedup of up to 50× over major CPU-based node embedding systems [35, 66, 81]. Here we compare FlashMob with GraphVite's random walk component. KnightKing [94] is a general-purpose random walk engine delivering orders of magnitude speedup over prior systems. While it was originally designed for distributed environments, KnightKing's latest release has been optimized for single-node executions.

Random Walk Algorithms. We evaluate FlashMob with two popular node embedding algorithms: (1) DeepWalk [66], a first-order walk with static transition probability that selects nodes uniformly at random, and (2) node2vec [35], a second-order walk with dynamic transition probability that interpolates between BFS and DFS.

Common practice in graph random walk evaluation executes 10 episodes, each with $|V|$ walkers walking 80 steps [35, 66, 94]. Here we adopt the same tradition and walk $10|V|$ walkers in total, each for 80 steps, though our number of walkers per episode is configured at runtime based on DRAM capacity. Results reported are averaged over 5 runs, with error bars omitted due to small variance.

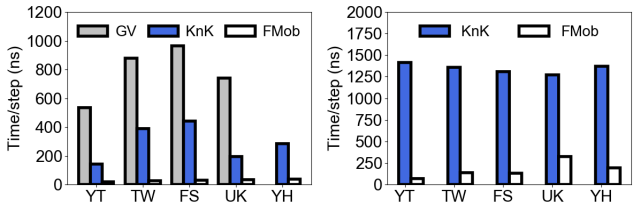
5.2 Overall Performance

Pre-processing overhead FlashMob requires certain pre-processing, whose overhead is rather insignificant. More specifically, sorting vertices by their degree on YH, our largest graph, takes 7.7 seconds using the $O(|V|)$-complexity counting sort [71], which can be further reduced by a multi-threaded implementation. FlashMob's vertex partitioning, including generating candidate policies and solving the MCKP problem, takes 0.34 seconds on YH. Overall, the total pre-processing overhead occupies 0.04% – 0.7% of the random walk time, across all five graphs evaluated. For the rest of our performance discussion, this pre-processing time is excluded.

Also, following common practice [32, 42, 94], the random walk time measurement does not include common tasks shared by all graph processing systems, such as loading graph data from disk and creating CSR.

Random walk performance Figure 8 gives the overall walk performance on our 5 real-world graphs, in per-step time, comparing FlashMob with the two baseline systems.

Figure 8a shows results for DeepWalk, where KnightKing is 2.2-3.8 times faster than GraphVite, due to its more efficient edge accesses. FlashMob, meanwhile, brings a speedup of 5.4-13.7× over KnightKing, finishing a walk step in under 40ns.



(a) DeepWalk **(b)** node2vec

Figure 8. Overall speed. DeepWalk with FlashMob (short bars): 21.5, 29.0, 32.4, 36.1, and 36.7ns/step, respectively.

For smaller graphs, it is able to achieve even faster speed, by fitting more sampling tasks into faster caches (more details below). Also, except for the smallest graph (YT), KnightKing delivers quite similar performance on other graphs, except for the UK outlier that we will explain next. FlashMob, in contrast, manages to harvest additional optimizations for smaller graphs, gradually lowering its per-step time from 37ns on YH to 21ns on YT.

Figure 8b gives node2vec results. Here we omit GraphVite as it significantly lags behind and would heavily skew the data presentation. The 2nd-order walk makes edge sampling much more complex. Still, FlashMob achieves a 3.9-19.9× speedup over KnightKing. The smaller profit margin is mainly due to the lower access locality as node2vec involves a connectivity check between a walker's sampled destination (candidate for its next stop) and its previous stop. Though FlashMob again batches such lookups, computation is no longer constrained within a single VP. For the rest of the paper, we focus on DeepWalk that is representative of the widely used first order random walks.

		KnK-FS	FMob-FS	KnK-UK	FMob-UK
L1-hit|miss/step		76.6 | 3.0	21.1 | 3.5	66.6 | 2.0	31.4 | 2.9
L2-hit|miss/step		0.1 | 2.9	2.8 | 0.7	0.6 | 1.5	2.1 | 0.8
L3-hit|miss/step		0.1 | 2.8	0.3 | 0.4	0.2 | 1.1	0.2 | 0.7
L1-bound	time	16.8ns	2.0ns	9.3ns	2.1ns
	pct.	3.8%	6.1%	4.8%	5.2%
L2-bound	time	5.3ns	0.7ns	2.0ns	0.6ns
	pct.	1.2%	2.2%	1.0%	1.4%
L3-bound	time	60.5ns	1.4ns	24.3ns	1.2ns
	pct.	13.7%	4.3%	12.5%	2.9%
DRAM-bound	time	307.1ns	12.1ns	115.7ns	18.4ns
	pct.	69.5%	37.4%	59.5%	44.8%
Total data-bound	time	389.7ns	16.2ns	151.3ns	22.3ns
	pct.	88.2%	50.0%	77.8%	54.3%
Avg. DRAM BW		15.7GB/s	51.1GB/s	11.7GB/s	57.1GB/s
DRAM traffic/step		310.4B	74.0B	101.8B	92.1B

Table 5. Memory hierarchy profiling case studies

Now we zoom into two test cases to take a closer look at differences in memory access behavior between KnightKing and FlashMob. Table 5 gives profiling details for the FS and UK graph runs, where FlashMob delivers the highest and lowest speedup over KnightKing, respectively.

The first rows list cache hit/miss count per step at each cache level measured via perf [56]. As expected, FlashMob reports significantly lower misses/step in both L2 and L3 than KnightKing's. Most importantly, the miss counts show L2 "catches" most of the L1 misses. KnightKing on FS, in contrast, has very similar miss counts across all cache levels and very low L2/L3 hits, indicating "straight misses" to DRAM. Its much higher L1 hit count is actually due to computation: it uses the Mersenne Twister [63] random number generator. FlashMob adopts the simpler xorshift* algorithm [61], reducing related computation time by more than 5×.

However, this cost (around 20ns/step) occupies a tiny fraction of KnightKing's total run time as it is dominantly data-bound. Changing to xorshift* only speeds up KnightKing by 4% and 9% on FS and UK, respectively. The next rows in Table 5 also confirm this, by the amount and percentage of per-step execution time bound on different memory hierarchy levels, measured with the Intel VTune Profiler [39]. FlashMob cuts the L3- and DRAM-bound time by up to 43.2× and 25.4 × respectively from KnightKing, producing a 24.1× saving in total data-bound time. As a result, only half of its execution time is data-bound, delivering much higher CPU efficiency than KnightKing.

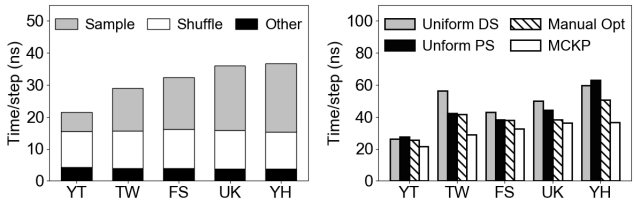
Meanwhile, its faster walk speed and streamlined sequential accesses achieve higher overall memory bandwidth (total DRAM traffic divided by total execution time): 3.3-4.9× over KnightKing. Yet with its two-stage processing and multiple scans added for shuffling, its DRAM access volume per step is only 1/4 of KnightKing's for the FS graph, thanks to more efficient use of full cache lines storing pre-sampled edges.

The last observation holds for 4 out of the 5 graphs, with UK being the only exception. Our memory traffic profiling results help explain KnightKing's better performance on the UK graph (hence FlashMob's smaller improvement). The UK graph encompasses stronger locality and lower "walker mobility". *E.g.*, its estimated diameter is much larger (147 vs. FS's 32 with only twice as many vertices). With walkers initialized by uniform distribution on vertices, neighboring walkers are likely to remain "close" in their walks, leading to better data reuse. This also explains KnightKing's better cache hit/miss counts on UK as compared to FS.

In addition, though not included in the table, we evaluated the impact of adopting simpler, regular graph data structures for low-degree partitions using DS. Our results show that overall, compared to using standard CSR, FlashMob reduces L2/L3 misses by 33%/30% with UK and 13%/20% with FS. This is noteworthy as only a small portion of walkers visit these lower-degree vertices (especially with FS, where 80.9% of walkers are in VPs adopting PS).

5.3 Effectiveness of DP-based Optimization

We evaluate FlashMob's DP algorithm for automatic vertex partitioning and sampling policy optimization by giving the breakdown of total walk time across FlashMob stages in the

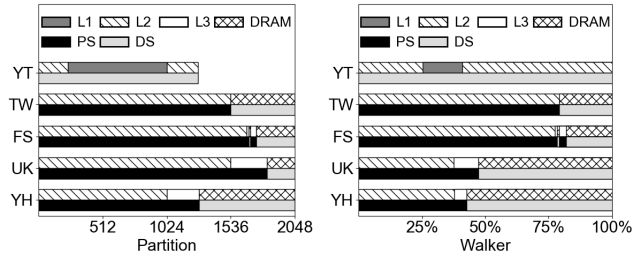


(a) Time breakdown across stages **(b)** DP vs. alternative approaches

Figure 9. DP-algorithm optimization results

DP-identified solution (Figure 9a). There the "Other" category includes all costs outside of the sample and shuffle stages, such as initialization and final output. The time split indicates the non-trivial challenges our optimization faces: partitioning and shuffling enable fast sampling by fitting working sets within cache, which in turn renders the relative cost of shuffling itself quite comparable with sampling.

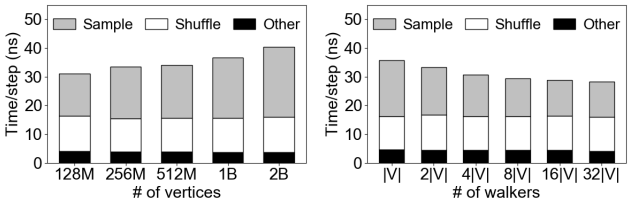
Figure 9b compares the DP-identified solution with several alternatives. We test two uniform partitioning strategies that cut the graph into 2048 equal-sized VPs (to allow single-level shuffle), adopting pre-sampling (PS) and direct sampling (DS), respectively. In addition, we include "Manual Opt", our best-effort partitioning and sample policy optimization, adopted in our prototype before we identified the MCKP mapping. Results show that the DP-based MCKP solution delivers significantly better overall performance than the uniform strategies, confirming the margin of improvement brought by adaptively sizing data and tasks. As expected, it also significantly beats our manual optimization, which is based on the heuristic of adopting PS for low-walker-density or high-degree vertices and DS for the others.



(a) By partitions **(b)** Weighted by walkers

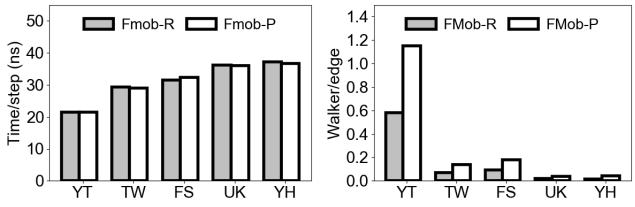
Figure 10. DP-identified solution for FlashMob auto-configuration. For each graph, the top bar gives the "VP size" decisions along the sorted vertex array, while the bottom one gives the "sampling policy" decisions. Figure 10a has each VP occupying equal bar length. Figure 10b shows the percentage of walker-steps on these VP size and sampling policy combinations.

Figure 10 illustrates the DP-identified solutions for each graph. First, the DP algorithm chooses to stay with single-level shuffle (limiting the total number of partitions to 2048).



(a) Growing $|V|$ w. synthetic graphs **(b)** Growing # of walkers on TW

Figure 11. FMob speed w.r.t. graph size and walker density



(a) Walk speed **(b)** Walker density

Figure 12. Walk w. NUMA: graph partitioning vs. replication

This might change for larger graphs. Second, for the smallest YT graph, which affords to use DS all the way, DP does not use up the total partition number allowance, as further partitioning lower-degree vertices to fit into L1 is found by profiling to hurt performance. Finally, the overall distribution of VP sizes and policies is quite intuitive: the high-degree vertices on the left of the bars are cut into smaller (mostly L2-size) VPs and assigned to use SP; the lowest degree vertices are usually using DP, and fitting into L3 does not help. MCKP's preference of L2 over L3 here reflects the impact of the current large private L2 cache design introduced in Section 2.3. Meanwhile, the exact places to cut in the vertex degree spectrum do not appear to follow easy patterns, making the DP algorithm especially helpful.

For the larger graphs, YH and UK, DP can fit some of its higher-degree vertices into L2-sized VPs, but not all. With each L3- and DRAM-sized VPs containing much more vertices each, when weighted by walkers the L2-sized VP recedes in its share, illustrating the DP algorithm's bias toward the highest-degree vertices.

5.4 Scalability

We now assess FlashMob's scalability by scaling up the graph size and walker density. The former (Figure 11a) is done by generating synthetic graphs using the degree distribution of YH, the largest real-world graph we tested. As expected, as $|V|$ grows (eventually to a 168GB graph), the sampling cost steadily rises as VPs get larger and more VPs are forced to adopt DS (again the DP algorithm chooses to stay with 1-level shuffle due to the considerable shuffling cost).

The latter (Figure 11b) is by simply doubling the total number of walkers (as multiples of V) on the TW graph. One sees that clearly higher density allows more efficient edge sampling (32.6% reduction in per-step sampling cost with $8|V|$ than with $|V|$ walkers). The benefit, however, levels off at that point, indicating that we do not need 100s of walkers per vertex for peak performance. Note that with traditional random walk systems, even $8|V|$ on a medium-sized graph is often not possible as it substantially blows up DRAM consumption. FlashMob's streaming approach, on the other hand, paves the way for future out-of-core processing while using DRAM to increase the walker density.

Finally, in Figure 12 we compare the two NUMA execution modes discussed in Section 4.5, tested on our dual-socket server, each running 12 threads. Across the 5 graphs, FlashMob-P and FlashMob-R actually show quite similar performance, as seen from Figure 12a. Recall that FlashMob-R avoids remote memory accesses altogether, while FlashMob-P saves DRAM space (by storing one rather than two copies of the graph) to accommodate more walkers simultaneously in each round. The latter is confirmed by the walker density comparison in Figure 12b: FlashMob-P almost doubles the density from FlashMob-R, promoting data reuse.

The VTune profiler reports that FlashMob-P only has 0.0023 and 0.0011 memory accesses per step that incur L3 cache misses leading to remote memory access on FS and UK, respectively. It proves that streaming data from remote memory does not slow down our cache-efficiency random walk, echoing the profiling results in Table 1. This further lends confidence for our future work extending FlashMob to out-of-core processing by streaming data from the disk. For example, FlashMob completes the 80-step DeepWalk in 881s. A larger graph streamed through the DRAM 80 times (assuming walk path results streamed to GPUs for consumption) would consume an I/O bandwidth of 5GB/s, below the capability of today's commodity NVMe SSDs.

6 Related Work

Streaming methods for graph systems. DrunkardMob [51] implemented out-of-core random walk on top of GraphChi [52]. Recent systems such as GraphWalker [89] mainly focus on optimizing I/O utilization and thus are orthogonal to our work. GraphChi [52], X-Stream [69], and GridGraph [101] are out-of-core graph analytic systems that process cache-friendly partitions in a streaming manner. However, typical graph engine tasks have vertices traverse *all* their edges, creating mostly sequential accesses, while graph random walks perform much sparser computation. To our knowledge, FlashMob is the first system to apply partitioning to *speedup graph sampling* and the first to *explicitly fit such computation into CPU caches* when possible.

Exploiting architectural features for graph computation. There have been extensive investigations on how graph analytics is impacted by novel hardware such as NVM [23, 29, 60], GPUs [15, 28, 45, 46, 72, 73], and SSDs [38].

FlashMob follows this tradition by targeting the effective use of the current-generation Intel cache hierarchy. In general, our work is motivated by similar inefficiencies found in graph random walk implementations and addresses the problem with cache-aware system designs.

A number of existing frameworks [6, 13, 77, 97, 99] perform cache-aware processing of specific graph analytics tasks. However, we are not aware of any prior work that improves upon graph sampling using cache locality.

Speeding up random walks. There have been increasing interest in performance optimization of random walks. Unfortunately, the randomness inherent in random walks obviates many of the optimizations in common graph analytics frameworks [13, 32, 59, 77, 99]. Hence, many prior studies are tailored towards accelerating specific applications of random walk such as PageRank [65] or SimRank [43].

There has been a paucity in application-agnostic approaches accelerating arbitrary random walk approaches. Classical techniques such as inverse transform sampling [22] and alias table [85] seek to perform pre-processing and/or use additional storage for faster edge sampling. Recent systems [74, 94] proposed algorithmic techniques based on rejection sampling to speed up a *single* random walk step. A number of recent production platforms for deep learning on graphs (*e.g.*, Plato [82], Euler [3], and AGL [96]) leverage these approaches for accelerating random walks. These optimizations have significantly reduced the (amortized) computation complexity but not not address the inefficiency brought by the predominantly random memory accesses.

Other efforts have focused on reducing computation or enhancing parallelism. *E.g.*, Spark-Node2Vec addresses the bottleneck of edge sampling for high-degree vertices by only considering a subset of 30 neighbors [34] or approximating transition probabilities [98]. A recent random walk system, NextDoor [42] uses intelligent scheduling and caching strategies to accelerate graph sampling on GPUs. However, GPU memories are much smaller than that of CPUs and the GPU-based systems perform best when the graph completely fits in the GPU memory, otherwise experiencing a steep performance drop. For example, on DeepWalk, Nextdoor is reported to be 436× faster than KnighKing on the smallest graph with 50K vertices, 26.1 × −38.9× faster on the 3 medium-size graphs with 3M-5.8M vertices, and 3× slower on the largest graph (FS), on which FlashMob is 13.6× faster than KnightKing. Again, *all prior systems* dominantly perform random, whole-graph memory accesses.

We argue that since GPUs are highly optimized for linear-algebraic operations, efficient CPU-side random walks complement their strength by supplying their learning computation with fast edge sampling, both improving overall resource utilization and shortening end-to-end computation. Recent systems such as GraphVite [102] adopt such CPU-sampling and GPU-training paradigm.

Finally, a very recent work, ThunderRW [80], also addresses the irregular memory access patterns of random walk, using a step interleaving technique to hide the memory access latency, which is orthogonal to our approach.

Node embeddings. An overview of algorithms for learning embeddings can be found in [8, 18, 33, 37]. Broadly, there are three types of approaches. Matrix factorization based approaches [9, 67, 68, 90] factorize a proximity matrix M where each entry $M[u, v]$ quantifies the proximity between vertices $u, v \in V$. Alternatively, one could use deep learning models such as autoencoders [10, 86], LSTM [84] and GANs [19, 87] to learn the node embeddings. Both of these approaches are expensive and do not scale to large graphs. Random walk based approaches [35, 66, 81] are widely used as they are flexible, easily parallelizable, and can scale to large graphs. GraphAny2Vec [30] and GraphVite [102] are recent examples of following this approach, whose focus is on building systems to compute node embedding on large graphs. FlashMob's optimization techniques are orthogonal to their efforts and could be used to accelerate the above existing systems.

7 Conclusion and Future Work

In this work, we debunk the long-held assumption that random walks on large graphs need to settle with random DRAM accesses. Our proposed system, FlashMob, manages to exploit hidden spatial and temporal locality from the random walk semantics, through careful partitioning, rearranging, and batching of operations. Our results reveal that modern servers' caches enable efficient "out-of-cache" processing even for data-driven programs with random and irregular computation. We also found that with the large disparity between sequential and random DRAM accesses, it is worthwhile to add access volume by performing more scans, which may end up saving both time and actual DRAM traffic.

Besides extending FlashMob to walk disk-resident graphs, we are also exploring other avenues involving sampling, where random accesses could be converted to sequential ones without affecting their statistical guarantees.

Acknowledgments

We thank the anonymous reviewers for their valuable comments and suggestions, and in particular our shepherd Phillip Gibbons for his detailed guidance. We thank the excellent support we have received from the QCRI local cloud, with high-end processors and profiling tools. We specifically thank Ahmed Tawfik, Anurag Shrivastava, and Mahmood Gajem for their excellent support. We are grateful to our graph data sources, including the Stanford SNAP Datasets [54], the Laboratory for Web Algorithmics [1], and the Yahoo Webscope Program [92]. This work has been partially supported by National Key Research & Development Program of China (2018YFB1003505) and Natural Science Foundation of China (61877035).

References

[1] [n.d.]. Laboratory for Web Algorithmics. http://law.di.unimi.it/datasets.php.

[2] Sami Abu-El-Haija, Bryan Perozzi, Rami Al-Rfou, and Alexander A Alemi. 2018. Watch your step: Learning node embeddings via graph attention. In *Advances in Neural Information Processing Systems*. 9180–9190.

[3] Alibaba. 2020. Euler. https://github.com/alibaba/euler

[4] Lars Backstrom and Jure Leskovec. 2011. Supervised random walks: predicting and recommending links in social networks. In *WSDM*. 635–644.

[5] Ziv Bar-Yossef, Alexander Berg, Steve Chien, Jittat Fakcharoenphol, and Dror Weitz. 2000. Approximating aggregate queries about web pages via random walks. In *VLDB*. 535–544.

[6] Scott Beamer, Krste Asanović, and David Patterson. 2017. Reducing pagerank communication via propagation blocking. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 820–831.

[7] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A large time-aware web graph. *SIGIR Forum* 42, 2 (2008), 33–38.

[8] Hongyun Cai, Vincent W Zheng, and Kevin Chen-Chuan Chang. 2018. A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Transactions on Knowledge and Data Engineering* 30, 9 (2018), 1616–1637.

[9] Shaosheng Cao, Wei Lu, and Qiongkai Xu. 2015. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM international on conference on information and knowledge management*. 891–900.

[10] Shaosheng Cao, Wei Lu, and Qiongkai Xu. 2016. Deep neural networks for learning graph representations.. In *AAAI*, Vol. 16. 1145–1152.

[11] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. 2020. Creating embeddings of heterogeneous relational datasets for data integration tasks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1335–1349.

[12] Sandro Cavallari, Vincent W Zheng, Hongyun Cai, Kevin Chen-Chuan Chang, and Erik Cambria. 2017. Learning community embedding with community detection and node embedding on graphs. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. 377–386.

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

[14] Xing Chen, MingXi Liu, and GuiYing Yan. 2012. Drug–target interaction prediction by random walk on the heterogeneous network. *Molecular BioSystems* 8, 7 (2012), 1970–1978.

[15] Yan-Hao Chen, Ari B. Hayes, Chi Zhang, Timothy Salmon, and Eddy Z. Zhang. 2018. Locality-aware software throttling for sparse matrix operation on GPUs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 413–426.

[16] Robert M Christley, GL Pinchbeck, Roger G Bowers, Damian Clancy, Nigel P French, Rachel Bennett, and Joanne Turner. 2005. Infection in social networks: using network analysis to identify high-risk individuals. *American journal of epidemiology* 162, 10 (2005), 1024–1031.

[17] Colin Cooper, Sang Hyuk Lee, Tomasz Radzik, and Yiannis Siantos. 2014. Random walks in recommender systems: exact computation and simulations. In *Proceedings of the 23rd International Conference on World Wide Web*. 811–816.

[18] Peng Cui, Xiao Wang, Jian Pei, and Wenwu Zhu. 2018. A survey on network embedding. *IEEE Transactions on Knowledge and Data Engineering* 31, 5 (2018), 833–852.

[19] Quanyu Dai, Qiang Li, Jian Tang, and Dan Wang. 2018. Adversarial network embedding. In *32nd AAAI Conference on Artificial Intelligence*. 2167–2174.

[20] Arjun Dasgupta, Gautam Das, and Heikki Mannila. 2007. A random walk approach to sampling hidden databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 629–640.

[21] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[22] Luc Devroye. 2006. Nonuniform random variate generation. *Handbooks in operations research and management science* 13 (2006), 83–121.

[23] Laxman Dhulipala, Charles McGuffey, Hongbo Kang, Yan Gu, Guy E. Blelloch, Phillip B. Gibbons, and Julian Shun. 2020. Sage: parallel semi-asymmetric graph algorithms for NVRAMs. 13, 9 (2020), 1598–1613.

[24] Krzysztof Dudziński and Stanisław Walukiewicz. 1987. Exact methods for the knapsack problem and its generalizations. *European Journal of Operational Research* 28, 1 (1987), 3–21.

[25] Peter Ebbes, Zan Huang, Arvind Rangaswamy, et al. 2010. *Subgraph sampling methods for social networks: The good, the bad, and the ugly*. Technical Report.

[26] Alessandro Epasto and Bryan Perozzi. 2019. Is a single embedding enough? Learning node representations that capture multiple social contexts. In *The World Wide Web Conference*. 394–404.

[27] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph neural networks for social recommendation. In *The World Wide Web Conference*. 417–426.

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

[29] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. 2020. Single machine graph analytics on massive datasets using Intel optane DC persistent memory. In *Proceedings of the VLDB Endowment*, Vol. 13. 1304–1318.

[30] Gurbinder Gill, Roshan Dathathri, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, and Olli Saarikivi. 2021. Distributed training of embeddings using graph analytics. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 973–983.

[31] Minas Gjoka, Maciej Kurant, Carter T Butts, and Athina Markopoulou. 2010. Walking in facebook: A case study of unbiased sampling of osns. In *IEEE INFOCOM*. IEEE, 1–9.

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

[33] Palash Goyal and Emilio Ferrara. 2018. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems* 151 (2018), 78–94.

[34] Aditya Grover and Jure Leskovec. 2016. Node2vec on Spark. https://github.com/aditya-grover/node2vec.

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

[36] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 1025–1035.

[37] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation learning on graphs: Methods and applications. *IEEE Data(base) Engineering Bulletin* 40 (2017), 52–74.

[38] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC.

In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 77–85.

[39] Intel. 2009. VTune Performance Analyzer. https://software.intel.com/content/www/us/en/develop/home.html.

[40] Intel. 2020. Second Generation Intel Xeon Scalable Processors. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/2nd-gen-xeon-scalable-processors-brief-Feb-2020-2.pdf.

[41] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. 2018. ASAP: Fast, approximate graph pattern mining at scale. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 745–761.

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

[43] Glen Jeh and Jennifer Widom. 2002. SimRank: A measure of structural-context similarity. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 538–543.

[44] Jinyuan Jia, Binghui Wang, and Neil Zhenqiang Gong. 2017. Random walk based fake account detection in online social networks. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 273–284.

[45] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. 2017. A distributed multi-GPU system for fast graph processing. *Proceedings of the VLDB Endowment* 11, 3 (2017), 297–310.

[46] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with ROC. *Proceedings of Machine Learning and Systems* 2 (2020), 187–198.

[47] Nadav Kashtan, Shalev Itzkovitz, Ron Milo, and Uri Alon. 2004. Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. *Bioinformatics* 20, 11 (2004), 1746–1758.

[48] Hans Kellerer, Ulrich Pferschy, and David Pisinger. 2004. The multiple-choice knapsack problem. In *Knapsack Problems*. Springer, 317–347.

[49] Maciej Kurant, Athina Markopoulou, and Patrick Thiran. 2010. On the bias of BFS (breadth first search). In *2010 22nd International Teletraffic Congress (lTC 22)*. IEEE, 1–8.

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

[51] Aapo Kyrola. 2013. Drunkardmob: Billions of random walks on just a PC. In *Proceedings of the 7th ACM conference on Recommender systems*. 257–264.

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

[53] Sangkeun Lee, Sang-il Song, Minsuk Kahng, Dongjoo Lee, and Sanggoo Lee. 2011. Random walk based entity ranking on graph for multidimensional recommendation. In *Proceedings of the fifth ACM conference on Recommender systems*. 93–100.

[54] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[55] LinkedIn. 2017. Random Walks on Large Scale Graphs with Apache Spark. https://www.slideshare.net/databricks/random-walks-on-large-scale-graphs-with-apache-spark-with-min-shen, Last accessed on 2020-12-10.

[56] Linux. 2009. perf. https://perf.wiki.kernel.org/.

[57] László Lovász et al. 1993. Random walks on graphs: A survey. *Combinatorics, Paul erdos is eighty* 2, 1 (1993), 1–46.

[58] Kathy Macropol, Tolga Can, and Ambuj K Singh. 2009. RRW: Repeated random walks on genome-scale protein networks for local cluster discovery. *BMC bioinformatics* 10, 1 (2009), 283.

[59] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[60] Jasmina Malicevic, Subramanya Dulloor, Narayanan Sundaram, Nadathur Satish, Jeff Jackson, and Willy Zwaenepoel. 2015. Exploiting NVM in large-scale graph analytics. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*. 1–9.

[61] George Marsaglia et al. 2003. Xorshift rngs. *Journal of Statistical Software* 8, 14 (2003), 1–6.

[62] Laurent Massoulié, Erwan Le Merrer, Anne-Marie Kermarrec, and Ayalvadi Ganesh. 2006. Peer counting and sampling in overlay networks: random walk methods. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*. 123–132.

[63] Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8, 1 (1998), 3–30.

[64] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. 29–42.

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

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

[67] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Chi Wang, Kuansan Wang, and Jie Tang. 2019. NetSMF: Large-scale network embedding as sparse matrix factorization. In *The World Wide Web Conference*. 1509–1520.

[68] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. 2018. Network embedding as matrix factorization: Unifying DeepWalk, LINE, PTE, and node2vec. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. 459–467.

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

[70] Yousef Saad. 2003. *Iterative methods for sparse linear systems*. SIAM.

[71] Harold Herbert Seward. 1954. *Information sorting in the application of electronic digital computers to business operations*. Ph.D. Dissertation. Massachusetts Institute of Technology. Department of Electrical Engineering.

[72] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Accelerating dynamic graph analytics on GPUs. 11, 1 (2017), 107–120.

[73] Mo Sha, Yuchen Li, and Kian-Lee Tan. 2019. GPU-based graph traversal on compressed graphs. In *Proceedings of the 2019 International Conference on Management of Data*. 775–792.

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

[75] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. 2016. GraphJet: Real-time content recommendations at Twitter. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1281–1292.

[76] Suraj Shetiya, Saravanan Thirumuruganathan, Nick Koudas, and Gautam Das. 2020. Astrid: Accurate selectivity estimation for string predicates using deep learning. *Proceedings of the VLDB Endowment* 14, 4 (2020), 471–484.

[77] Julian Shun and Guy E Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, Vol. 48. ACM, 135–146.

[78] Prabhakant Sinha and Andris A Zoltners. 1979. The multiple-choice knapsack problem. *Operations Research* 27, 3 (1979), 503–515.

[79] Don Soltis, Irma Esmer, Adi Yoaz, and Sailesh Kottapalli. 2017. The New Intel Xeon Processor Scalable Family (Formerly Skylake-SP). In *IEEE Hot Chips 32 Symposium*.

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

[81] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. LINE: Large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*. 1067–1077.

[82] Tencent. 2019. Plato. https://github.com/Tencent/plato

[83] Saravanan Thirumuruganathan, Nan Tang, Mourad Ouzzani, and AnHai Doan. 2020. Data curation with deep learning.. In *EDBT*. 277–286.

[84] Ke Tu, Peng Cui, Xiao Wang, Philip S Yu, and Wenwu Zhu. 2018. Deep recursive network embedding with regular equivalence. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2357–2366.

[85] Alastair J Walker. 1977. An efficient method for generating discrete random variables with general distributions. *ACM Transactions on Mathematical Software (TOMS)* 3, 3 (1977), 253–256.

[86] Daixin Wang, Peng Cui, and Wenwu Zhu. 2016. Structural deep network embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 1225–1234.

[87] Hongwei Wang, Jia Wang, Jialin Wang, Miao Zhao, Weinan Zhang, Fuzheng Zhang, Xing Xie, and Minyi Guo. 2018. GraphGAN: Graph representation learning with generative adversarial nets. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 32. 2508–2515.

[88] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale commodity embedding for e-commerce recommendation in Alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 839–848.

[89] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John C. S. Lui. 2020. GraphWalker: An I/O-efficient and resource-friendly graph analytic system for fast and scalable random walks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 559–571.

[90] Xiao Wang, Peng Cui, Jing Wang, Jian Pei, Wenwu Zhu, and Shiqiang Yang. 2017. Community preserving network embedding.. In *AAAI*, Vol. 17. 203–209.

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

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

[93] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213.

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

[95] Renchi Yang, Jieming Shi, Xiaokui Xiao, Yin Yang, Juncheng Liu, and Sourav S. Bhowmick. 2020. Scaling attributed network embedding to massive graphs. In *Proceedings of the VLDB Endowment*, Vol. 14. 37–49.

[96] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. 2020. AGL: A scalable system for industrial-purpose graph machine learning. 13, 12 (2020), 3125–3137.

[97] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. 2017. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 293–302.

[98] Dongyan Zhou, Songjie Niu, and Shimin Chen. 2018. Efficient graph computation for node2vec. *arXiv preprint arXiv:1805.00280* (2018).

[99] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *the Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 301–316.

[100] Xiaojin Zhu, Andrew B Goldberg, Jurgen Van Gael, and David Andrzejewski. 2007. Improving diversity in ranking using absorbing random walks. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*. 97–104.

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

[102] Zhaocheng Zhu, Shizhen Xu, Jian Tang, and Meng Qu. 2019. GraphVite: A high-performance CPU-GPU hybrid system for node embedding. In *The World Wide Web Conference*. 2494–2504.